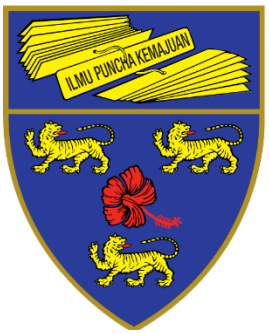

UNIVERSITY OF MALAYA

WIA1002: Data Structure

Group Assignment Technical Report

Always On Time Delivery

Prepared by

sadcat.com

Occurrence 7

Dr. Mohd Hairul Nizam bin Md Nasir

June 14, 2021

Contents

Question

Project 2: Always on Time Delivery

A. Introduction

Your friend's delivery company 'Never on Time Sdn. Bhd.' is receiving tons of complaints from customers as they feel that the delivery process is far too slow. Delivery men in your friend's company are always lost in the middle of their delivery route, and do not know where to deliver the parcel and which road they should take to shorten the delivery time. Sometimes they feel angry and exhausted when they lose their direction, and they eventually take it out on the parcels which causes more complaints from customers. Your friend tried out many ways to solve the problem but to no avail. Hence, you are your friend's last hope to save his company.

B. Problem Statement

In this assignment, you as a professional engineer are requested to simulate the delivery process and planning in a country to help your friend shorten their delivery time.

i) Basic Requirements

Let us start with definitions of all terms we are going to use in this problem context. A customer is an entity that has a certain demand and therefore requires the presence of a vehicle, a unit that can move between customers and the depot, a unit that initially possesses the demands of the customers. All vehicles are capacitated so that they can only contain goods (the customer's demands) up to a certain maximum capacity. Moving a vehicle between the depot and the customers comes with a certain cost. A route is a sequence of visited customers by a certain vehicle, starting and ending at a depot while a tour is a list of routes of all vehicles to fulfil all customers' demands.

You can imagine the underlying structure of this problem as a complete undirected graph G(V, E). A simple visualization is given in Fig. 1 below.

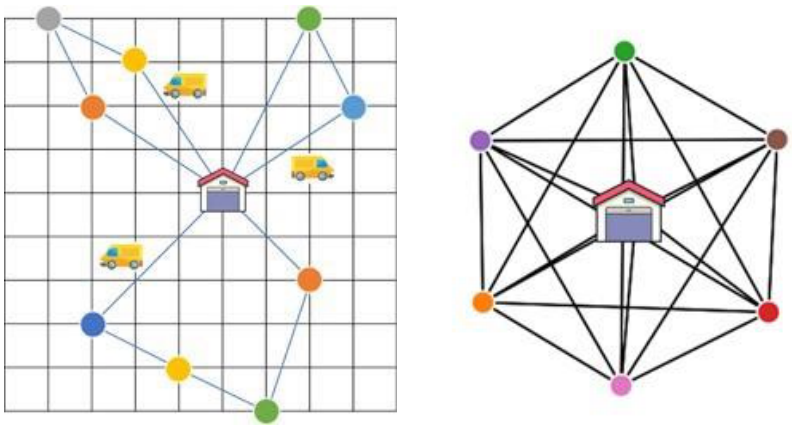

Figure 1: Visualization of the problem's underlying structure.

ii) Input

Given a text file, where the first row indicates the number of customers (including depot), N and maximum capacity of all vehicles, C. After this, starting from the second row onwards are the N rows of information. In particular, every row of information contains 3 data, which are x-coordinate, y-coordinate and lastly demand size of a customer. The second row represents the depot, and its demand size is always 0 while the rest of the rows show customer information. An example of input is given below.

N C	5 10
depot(x, y, capacity=0) ID = 0	86 22 0
customer1(x, y, capacity) ID = 1	29 17 1
customer2(x, y, capacity) ID = 2	4 50 8
customer3(x, y, capacity) ID = 3	25 13 6
customer4(x, y, capacity) ID = 4	67 37 5

Figure 2: The format of the input file and the example input.

In Fig. 2, the left side of the figure shows the input file format, while the right side of the figure shows an example of input text file. In this example, N=5 indicates that there are 4 customers and a depot, and the maximum capacity for all vehicles you use is 10.

It is then followed by 5 rows of information. Second row of the example input indicates location of the depot in x and y coordinates, (86, 22) and demand size is 0. The next 4 rows (viz., third row to sixth row) show locations and demand size of every customer. For instance, the first customer is located at (29, 17) with demand size of 1. In order for us to identify the depot and every customer, we simply give each one of them an ID according to their sequence in the text file. For example, the depot located at (86, 22)

is assigned an ID of 0, then the first customer located at (29, 17) with demand size 1 is assigned an ID of 1, followed by ID 2 for the second customer at (4, 50) with demand size 8, etc.

Note that Fig. 2 is only an example input, you may adjust input format for your own needs.

Given any input of this format, your task is to find out the best tour with lowest cost (a tour's cost is simply the summation of all routes' costs while a route's cost is simply the total Euclidean distance travelled by the vehicle using that route). Note that this also means that your program needs to find out what is the best number of vehicles to be used as well in order to achieve the lowest cost.

iii) Output

There are a total of 3 outputs you have to generate; we define each output as a simulation.

In the first simulation, you are requested to find the **best tour** for a given case with small N using **Breadth-First Search** / **Depth-First Search** traversal implementation which you will learn in the class. We name this approach '**Basic Simulation**'.

Secondly, you are requested to implement a **Greedy Search** that is able to find a **good solution** given a case with small or large N. Greedy Search means we simply look for the best option in the next move when we travel through the whole graph. For illustration, in this context the vehicle will always look for the shortest distance between current location and all next possible locations to select the next location to go. A simple explanation will be given in the GitHub link provided in the 'Resource' section. We call this approach '**Greedy Simulation**'.

Now, this is the most important part of this simulation. We want to search for the **best tour that we could search for in a limited computation time**. This means that if we are given enough time, then we will provide the best tour, else we will just provide the best tour we could find if we are given a limited time with large N. Thus, we are going to implement another searching algorithm, which is known as **Monte Carlo Tree**

Search. A brief explanation of this algorithm is given in the next section. We name this approach '**MCTS Simulation**'.

For every simulation, you need to show the **tour's cost**, followed by every used vehicle's information which includes the **route taken**, **vehicle's used capacity** and **cost of the route taken**. Sample output is given below with the case of the sample input given in the previous section.

```
Basic Simulation
Tour
Tour Cost: 346.2483982181608
Vehicle 1
0 -> 4 -> 0
Capacity: 5
Cost: 48.41487374764082
Vehicle 2
0 -> 2 -> 0
Capacity: 8
Cost: 173.29743217947575
Vehicle 3
0 -> 3 -> 1 -> 0
Capacity: 7
Cost: 124.53609229104421
```

```
Greedy Simulation
Tour
Tour Cost: 420.98628823988776
Vehicle 1
0 -> 4 -> 1 -> 0
Capacity: 6
Cost: 124.36813598466804
Vehicle 2
0 -> 3 -> 0
Capacity: 6
Cost: 123.32072007574396
Vehicle 3
0 -> 2 -> 0
Capacity: 8
Cost: 173.29743217947575
```

```
MCTS Simulation
Tour
Tour Cost: 346.2483982181608
Vehicle 1
0 -> 3 -> 1 -> 0
Capacity: 7
Cost: 124.53609229104424
Vehicle 2
0 -> 4 -> 0
Capacity: 5
Cost: 48.4148737476408
Vehicle 3
0 -> 2 -> 0
Capacity: 8
Cost: 173.29743217947575
```

Figure 3: Example output based on the input file in Figure 2. Put them side by side to save spaces.

iv) Monte Carlo Search

In MCTS, nodes are the building blocks of the search tree. These nodes are formed based on the outcome of a number of simulations. In general, MCTS selects next moves based on certain strategies until the end (leaf node) and based on the evaluation of the status in leaf node, we backpropagate the evaluation result to the root node to update the strategy so that we can select a better move in the next loop. The process of Monte Carlo Tree Search can be broken down into four distinct steps, selection, expansion, simulation, and backpropagation as shown in the figure below.

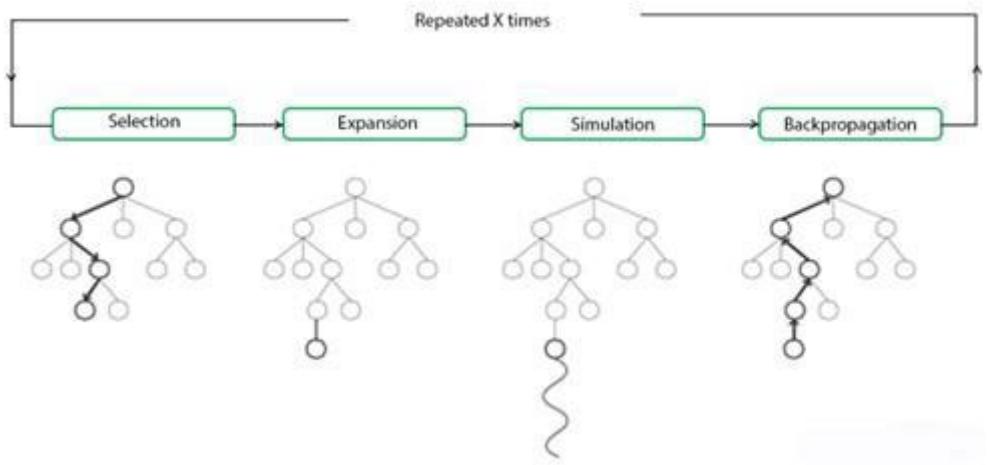

Figure 4: The Monte Carlo Search Tree process

Selection

In this process, the MCTS algorithm traverses the current tree from the root node using a specific strategy. The strategy we are going to use to solve our problem is policy adaptation. It is a bit long to be explained here and out of the scope of this Data Structure course thus the full logic of selection will be provided. Based on the policy, we select the 'best child' from all possible child nodes and enter the expansion phase.

Expansion

In this process, a new child node is added to the tree to that node which was optimally reached during the selection process.

Simulation

In this process, a simulation is performed by choosing moves or strategies until a result or predefined state is achieved. We can choose those moves randomly, to perform a random simulation, but for better efficiency, we choose the moves according to policy.

Backpropagation

After determining the value of the newly added node, the remaining tree must be updated. So, the backpropagation process is performed, where it back propagates from the new node to the root node. During the process, the policy we use to select moves is updated according to the cost of the result.

MCTS was indeed successfully implemented in many real-life applications such as

board games including Chess and Go (AlphaGo), protein folding problems, chemical design applications, planning and logistics, building structural design, interplanetary flights planning and is currently intended as one of the best approaches in Artificial General Intelligence for games (AGI).

v) Extra Features

Graphic User Interface (GUI)

Build your simulator with a nice-looking GUI. You can either simulate it in a graph or in a graph with a graphic map background. Your program should simulate the process of delivery (movement of vehicles between locations with respect to time).

Random Parcel Pick Up Spawning

During the parcel delivery process, there might be customers requesting parcel pick up from their home to be delivered to other places. Thus, in order to minimize cost, we have to update the path used by couriers (vehicles) whenever there is a new request for parcel pickup from a new location.

Pickup and Delivery

In this case, parcels are not initially located at the depot, instead parcels are on the customer's site. For every customer, you need to send a vehicle to their location to pick up the parcel (demand) and send it to another location specified by the customer (demand is released at the destination).

There are few things you might need to be careful of: first, a delivery point cannot come before its respective pickup point when you find the best route for your couriers. Secondly, all vehicles departing from the depot have 0 used capacity (as parcels are not inside the depot anymore). When a vehicle reaches a pickup point, it decrements the available capacity in the vehicle as it picks up the parcel. When a vehicle reaches a delivery point with its respective parcel, it releases the parcel and thus increments back the available capacity of the vehicle.

Heterogeneous Vehicle Capacity

In basic requirements, we assume that every vehicle shares the same capacity, C. In fact, we might have different types of vehicles that have different capacity (e.g., a lorry

can deliver more loads than a van). In order to produce a simulation closer to real life, you might need to consider adding this feature.

Time constraint from consumer

In real life, we not only have to minimize the time and fuel (represented by distance) used by couriers, but we also have to consider the expected arrival time of every parcel to their owners. We should not deliver the parcel later than its expected arrival time as it would result in bad customer reviews. In your simulation, you might want to add this feature as well.

For every customer, you might need to add a time window [t1, t2] to specify the time range we can deliver a parcel to a customer. If we arrive at the customer location before t1, then we have to wait for it and do nothing else, and if we deliver the parcel after t2, then we will receive a penalty which is undesired. Thus, your tour should strictly follow the time window for every customer and at the same time minimize the cost.

Traffic Conditions

Traffic conditions for every road keep changing due to many reasons which we can't predict (accidents, peak hours, holiday seasons etc.). In your simulation, you can also simulate this by assigning flexible time (that may change from time to time) to every road (connection between nodes). When there are traffic condition changes, the path taken by every courier that is affected should make some changes as well.

Site-Dependent Customer

Not every type of vehicle can serve every type of customer because of site-dependent restrictions. For example, customers located in very narrow streets cannot be served by a very big truck, and customers with very high demands require large vehicles. So associated with each customer is a set of feasible vehicles but not all.

Extra Algorithm Implementation

You can implement other searching algorithms to search for a best-known path for the delivery process. Here are some of the possible searching algorithms you might want to consider:

-Best First Search
-A* Search
-Genetic Algorithm
-Hybrid Search I (MCTS + GA)
-Hybrid Search II (MCTS + ML)
-Your custom searching algorithm

Parallelism (Threading)

You might want to apply parallel programming for the MCTS algorithm. For MCTS, simulations made can be run parallelly to reduce time usage greatly so that we can get a better result with shorter time for large N.

You can also apply parallel programming by parallelising the process of vehicle movement and route searching. You can try to search a best next move first using MCTS, then while vehicle is moving to the next location (which the process should take sometimes if you animate your simulation), you can continue to search the next best move from the next location so that you do not have to waste so much time initially for searching the best whole route for every vehicle before they depart.

Approaches

Before we begin programming, we first discussed the algorithms that will be used to fulfil the requirements of this project. To help with the discussion, we referred to the sample output provided in the question. From there, we determined that we need to consider the following:

- The capacity of each vehicle.
- The cost of each route traversed.
- The predicted path traversed according to each ID assigned.
- The tour cost for each vehicle.

To further understand these requirements, we made a general flowchart on how the program would run in Figure 5.

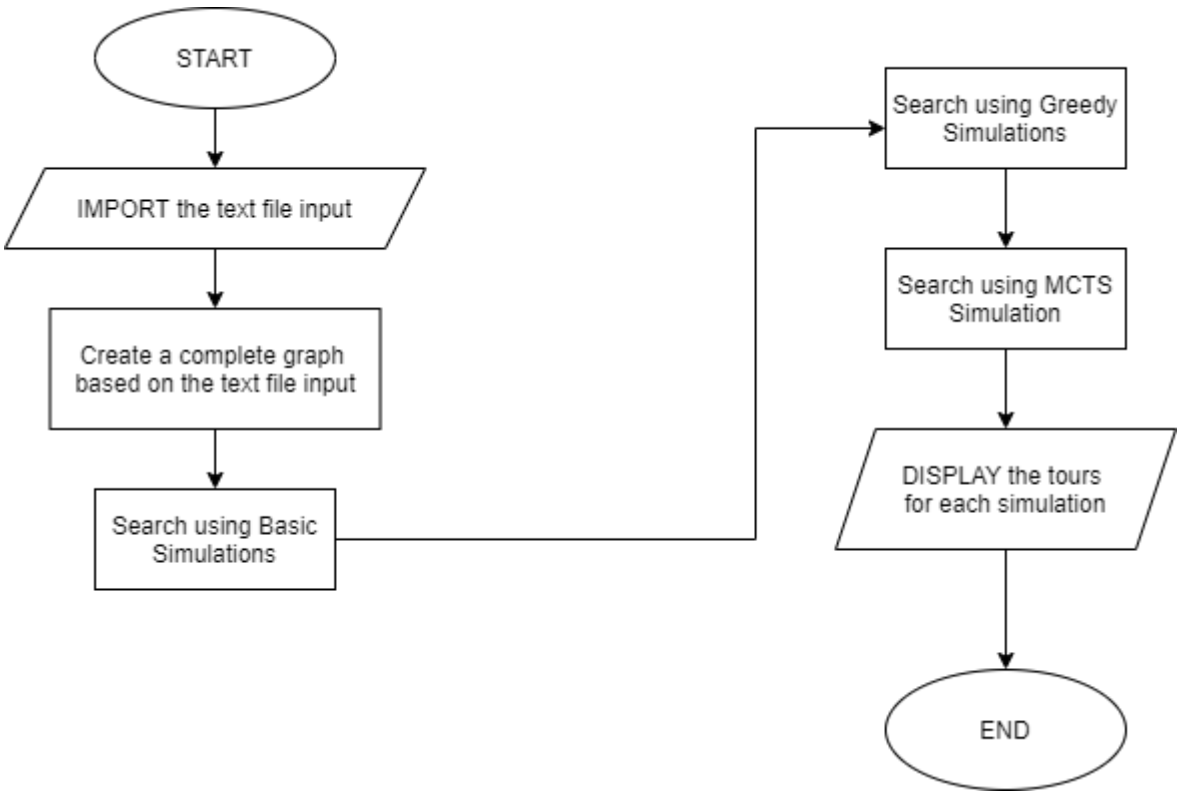

Figure 5: The general flowchart of the program.

Basic Simulation

Starting with the basic simulation, we first compared the total capacity for each path, using 10 as the maximum capacity. With the maximum capacity of 10, two out of three vehicles will only be able to deliver to one node only; Vehicle 1 going to ID 4, Vehicle 2 going to ID 2, and Vehicle 3 to ID 1 and 3. This is as shown in the example output in the introduction. We also tried comparing the costs for each route at the same time, but the outcome was the same, so we concluded that for basic simulation, this will be the best possible route for each vehicle. Using this concept, we tried to implement different algorithms for the basic simulation and were able to use the Blind Depth-First Search and Depth-First Search algorithm.

For Blind DFS, the program will accept an input file from the user. It will then traverse through the ID of the customers, starting with the last ID until the first ID. The route that will be taken for each vehicle is determined by comparing the amount of capacity left for them and the demand of the current order considered. For example, if Vehicle 1 currently has capacity of 4 and the current demand is 3, the route for the current order will be added into Vehicle 1 and the amount of capacity will be deducted. If the next demand has capacity of 2, which is more than the capacity left in Vehicle 1, it will be assigned to the next vehicle.

DFS is almost identical to Blind DFS with a slight difference before searching begins. Instead of instantly traversing through the ID of the customers, DFS constructs a tree of possible paths which is then used to go through each move.

Greedy Simulation

For the greedy simulation, we managed to implement multiple algorithms. They are:

- A* Search
- Best First Search
- Best Path Search
- Dijkstra Algorithm

All of these algorithms have a similarity whereby they keep updating the visited vertices and determine the right choice of route for each vehicle given their respective situations in terms of visited vertices, the cost for each path, and the demand of each order. Visited vertices will be ignored by the next vehicle, allowing for conserved capacity. The breakdown of the comparisons are as the following:

- A* Search: Heuristics (demand) & step cost (total distance)
- Dijkstra : Step cost (Total distance) only.
- Best-First : Heuristics (demand) only.
- Best-Path : Cost (distance) only

MCTS Simulation

For this simulation, we implemented the NRPA variant of the MCTS algorithm. As explained earlier, the MCTS algorithm consists of four key steps; Selection, Expansion, Simulation, and Backpropagate. In NRPA-MCTS, the program will construct a rollout tree by applying 'policies' to search for the best move from the node of one level to another node from the next level. Once finished with the rollout tree, the program will carry out the basic MCTS algorithm to finalize and discover the best tour for each vehicle. This algorithm will possibly yield the best results of all simulations, due to the learning process that keeps the best move updated by learning from past iterations. However, due to the same reason, the NRPA-MCTS algorithm will take a longer time, since it compares for every possibility that can be considered in finding the best tour. The policy used in this algorithm compares between two distances and chooses the shorter.

Extra Features

Using JavaFX, we were able to develop a Graphical User Interface (GUI) for the program. This GUI will allow for better understanding of each vehicle movement through visualization, as the nodes of each ID will change colour once it is traversed through by a delivery vehicle.

We also implemented the Heterogeneous Vehicle Capacity feature by adding lorries to the basic vehicles. The lorries have double the capacity of the regular delivery vehicles, allowing for longer routes and more demands to be fulfilled. However, this extra feature is not implemented in Revised DFS and NRPA-MCTS algorithms as it caused major bugs and disrupted the parallelism process.

In addition, we implemented site-dependent customer, where there are two cases:

i. The customer may be in a narrow area, which causes difficulties for lorries to be dispatched there. This means that only regular delivery vehicles can access through the area.

ii. The customer may be in a wider area (¬narrow), where lorries and regular deliver vehicles can fulfil the customer's demands there.

In this program, to determine whether the customer is in a narrow area or not, we used the inequality of a circle with the centre (x_c, y_c) and radius r to state that the interior of the circle refers to the narrow area.

$$(x - x_c)^2 + (y - y_c)^2 \leq r^2$$

If the customer at (x, y) satisfies this inequality, then the customer is in a narrow area, else it is in a wide area. We plotted the coordinates of the customers (x, y) and the circle to illustrate this feature in Figure 6.

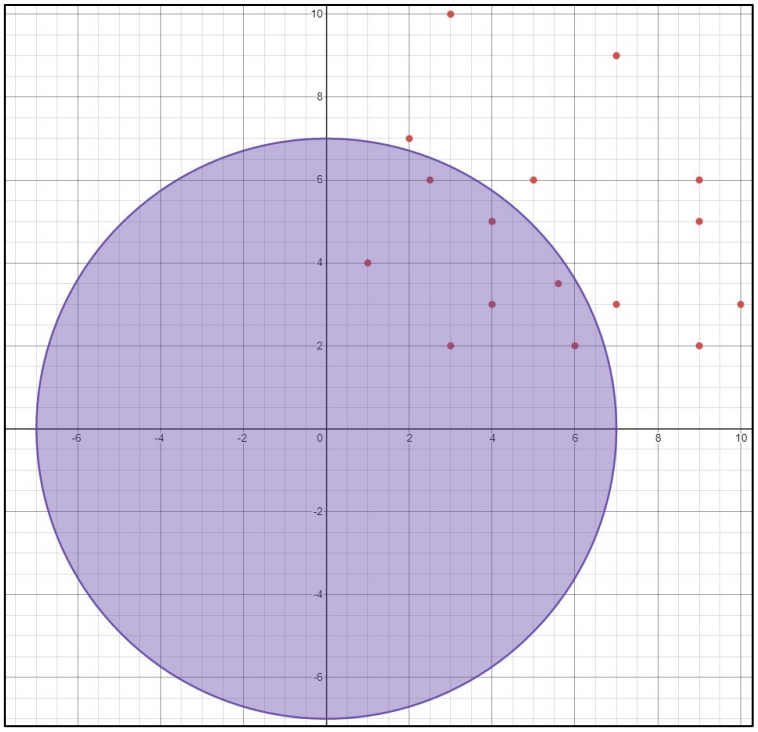

Figure 6: The plotted coordinates of the customers (*red dots*) and the narrow area (*purple shade*)

This feature is not implemented in Revised DFS and NRPA-MCTS algorithms as the heterogeneous vehicle capacity feature is not implemented too. We concluded that both of these extra features are dependent.

Pseudocode

Blind Depth-First Search

1. Given a graph, G.
2. Create an empty array list for all visited vertices except depot.
3. Initialize the number of lorries and vehicles dispatched as 0.
4. While all vertices are not visited:
    a. Check for the type of vehicle used.
    b. Set dT as 0 (dT is total distance travelled).
    c. Set totalCap = 0 (total capacity taken for current vehicle)
    d. Set tempD= 0 (distance for each trip)
    e. Go through every vertex:

        i. If the current vehicle is a lorry, and the destination is a narrow area, continue the loop.
        ii. If capacity of current vehicle >= demand AND destination not visited yet:
        iii. Choose this path.
        iv. Go to the vertex based on the path.
        v. Update tempD.
        vi. Add destination to visited vertices.
        vii. Update dT, totalCap, and current available Capacity.

Revised Depth-First Search

1. Given a graph, G.
2. Generate tree (to limit capacity)
3. Set distance = 0.
4. Set capacity = 0.
5. Generate an array of nodes.
6. Find the distance for every path by computing distance between two nodes.
7. Construct paths for the tour.
8. Compare the distance between same visited nodes using the graph:

a. If the new path of visited nodes has less distance than the old path of same visited nodes:

b. Replace old path.

9. Add all paths into a list of paths, tagged with an ID.

10. Search for the best tour in the list of paths.

A* Search

1. Given a graph, G.
2. While all vertices are not visited:
    a. Set dT as 0. (dT is the total distance travelled)
    b. Let cV: the expected path cost each vertex take.
    c. Set cV = {+inf, +inf, +inf, ...}
    d. Send a vehicle.
    e. Start from the depot (ID 0)
    f. Let currV: current Vertex

    g. Loop through all other vertices:
        i. Let currE: current Edge.
        ii. Loop through all edges/path connected from currV:

            1. if the demand is sufficient AND the destination is not visited AND dT + the path's distance + the heuristic< cV:
            2. NOTE: heuristic = the destination's demand size
            3. choose this path.
            4. update cV with the new expected total cost value
            5. keep the chosen path's demand size to update dTs

    h. Add the chosen vertex to go into the "visited" list.
    i. Update the total distance travelled by the vehicle. (dT = dV - chosen_demand)
    j. Deduct the capacity (send the package)
    k. Go to the chosen vertex using the chosen path.

Best First Search

1. Given a graph, G.
2. While all vertices are not visited:
    a. Set dT as 0. (dT is total distance travelled)
    b. Let hV: the expected demand (heuristic function) each vertex takes.
    c. Set hV = {0, 0, 0, …}
    d. Send a vehicle.
    e. Start from depot (ID 0)
    f. Let currV: current Vertex

    g. Loop through all other vertices:
        i. Let currE: current Edge.
        ii. Loop through all edges/path connected from currV:

            1. if the demand is sufficient AND the destination is not visited AND the destination's demand > hV AND the destination is not the depot:
            2. Choose this path.
            3. Update hV with the new expected demand value.
            4. Keep the chosen path's distance.

        iii. if the capacity is insufficient OR all vertices are visited:
            1. Choose the path to the depot.
            2. Update hV with the new heuristic value
            3. Keep the chosen path's distance.
        iv. Add the chosen vertex to go into the "visited" list.
        v. Update the total distance travelled by the vehicle. (From saved chosen path's distance)
        vi. Deduct the capacity (send the package).
        vii. Go to the chosen vertex using the chosen path, if possible.

Best Path Search

1. Given a graph, G.
2. While all vertices are not visited:
    a. Set dT as 0. (dT is total distance travelled.)
    b. Let cV: the expected path cost each vertex take.

c. Set cV = {+inf, +inf, +inf, ...}
d. Send a vehicle.
e. Start from the depot (ID 0)
f. Let currV: current Vertex.
g. Loop through all other vertices:

    i. Let currE: current Edge.
    ii. Loop through all edges/path connected from currV:

        1. if the demand is sufficient AND the destination is not visited AND dT + the path's distance + the heuristic< cV:
        2. choose this path.
        3. update cV with the new expected total cost value
        4. keep the chosen path's demand size to update dTs

    iii. Add the chosen vertex to go into the "visited" list.
    iv. Update the total distance travelled by the vehicle.
    v. Deduct the capacity (send the package)
    vi. Go to the chosen vertex using the chosen path.

Dijkstra Search

1. Given a graph, G.
2. While all vertices are not visited:
    a. Set dT as 0. (dT is total distance travelled.)
    b. Let dV: expected path distance/weight each vertex takes.
    c. Set dV = {+inf, +inf, +inf, …}
    d. Send a vehicle.
    e. Start from the depot (ID 0)
    f. Let currV: current Vertex

    g. Loop through all other vertices:
        i. Let currE: current Edge.
        ii. Loop through all edges/path connected from currV:

1. If the demand is sufficient AND the destination is not visited AND dT + the path's distance < dV:
2. Choose this path.
3. Update dV with the new expected total distance value.

iii. Add the chosen vertex to go into the "visited" list.
iv. Update the total distance travelled by the vehicle.
v. Go to the chosen vertex.

NRPA-MCTS

1. Given a graph, G.
2. Let level = 3. (level: tree level of rollout tree)
3. Let iterations = 100 (iterations: number of simulations to backpropagate and update policies of each node for each level.)\
4. If level = 0, perform rollout.
5. While level of rollout tree != 0:
    a. Update the policy at current level with current globalPolicy (globalPolicy is the reference for move selection when performing rollout.)
    b. Loop through all iterations:

        i. Construct a new tour, new_tour with the level set as (level - 1) using the current iteration.
        ii. If the total distance for this tour < total distance for current best tour:
        iii. Update the best tour for the level with this tour (adapt)

    c. Update the globalPolicy with the current policy (which will be used by next level)

6. Return the best_tour.

Class and Methods

In this section, we listed all the classes and methods we implemented for this program.

Basic Simulation

Blind Depth-First Search
Variables -G : Map ❖ Map of the delivery -N : int ❖ Number of vehicles -C : int ❖ Capacity of each vehicle -lorries : int ❖ Number of lorries
Methods +run(Map G, int N, int C, int numberOfLorries) : void ❖ Runs the simulation. +search() : void ❖ Starts the search algorithm using parameters given in run.

Revised Depth-First Search
Variables -G : Map ❖ Map of the delivery -C : int ❖ Capacity of each vehicle -tree : List<String> ❖ List of all possible trees -pathMap : HashMap<String, Path>

-pathList : List<Path>

- List of all paths discovered.

-route : List<String>

- List of all routes constructed.

-tourDistance : double

- Distance for each tour

-start : long

- Simulation start time

-end : long

- Simulation end time

-time : long

- Total time taken to find best tour.

-maxTime: long

- Maximum time for simulation to run

Methods

+run(Map G, int C) : String

- Run simulation

+generateTree(int capacity, int vertexID, String currentList) : void

- Generates tree for algorithm

+search() : void

- Starts search algorithm

+computeDistance (Vertex v1, Vertex v2) : double

- Compute the distance for every path

+arrayPathChecker(Path from, Path dest) : boolean

- Find path between two vertices

+pathGraph(Path currentPath) : void

- Creates graph from paths

+bestTour(int pathID, String visitedPath, String visitedNodes, double distance) : void

- ❖ Finds best tour among paths

+haveIntegerInString(Integer[] nodes, String visited) : boolean

- ❖ Checks visited nodes

+StringContainAllNodes(String visited) : boolean

+printAllEdge() : void

- ❖ Displays all edges for the current path.

Greedy Simulation

A* Search
Variables -G : Map ❖ Map of the delivery -C : int ❖ Capacity of each vehicle -lorries : int ❖ Number of lorries used -tourCost : double ❖ Total cost of the tour
Methods +run(Map G, int C, int numberOfLorries) : void ❖ Runs simulation using the parameters given +search() : String ❖ Starts the search algorithm, returns the vertices visited, capacity and vehicle used, and the capacity for each tour.

BestFirst
Variables -G : Map ❖ Map of the delivery -C : int ❖ Capacity of each vehicle -lorries : int ❖ Number of lorries used -tourCost : double ❖ Total tour cost
Methods

+run(Map G, int C, int numberOfLorries) : void

- ❖ Runs the simulation
- ❖ Prints the tourCost, result, and time taken for the simulation to complete.

+search() : String

- ❖ Starts the search algorithm
- ❖ Returns a String of the Vehicle used, sequence of vertices visited, capacity used and the cost of tour.

BestPath

Variables

-G : Map

-C : int

-lorries : int

-tourCost : double

Methods

+ run(Map G, int C, int numberOfLorries) : void

+search() : String

Dijkstra

Variables

-G : Map

- ❖ Map of the delivery

-C : int

- ❖ Total capacity of each vehicle

-lorries : int

- ❖ Number of lorries used

-tourCost : double

- ❖ Total tour cost

Methods

+run(Map G, int C, int numberOfLorries) : void

- Run the simulation
- Returns the tourCost, search result, and the time taken to complete simulation.

+search() : String

- Starts the search algorithm
- Returns a String of vehicle used, sequence of vertices visited, capacity used and cost of tour.

MCTS Simulation

NRPA

Variables

-G : Map

- Map of the delivery

-N : int

- ID of each order

-C : int

- Order capacity

-tourCost : double

- Total cost of the tour

-policy : double[][][]

- Policy for next move from each level

-globalPolicy : double[][]

- Reference when deciding next move

-bestTour : Tour

- Best next move for each level

-ALPHA : double

- Hyperparameter.

Methods

+run(Map G, int N, int C, int level, int iterations, double ALPHA) : String

- Runs the simulation.
- Fills each row of first level with 0.
- Returns the search result and its total cost.

+search(int level, int iterations) : Tour

- Starts the NRPA algorithm.
- Returns bestTour of each level to construct rollout tree

+adapt(Tour currentTour, int level) : void

- Updates the policy for the current level using current tour's result.

+rollout() : Tour

- Carries out basic MCTS algorithm on the rollout tree.

+select_next_move(Vertex currentStop, List<Integer> possibleSuccessor) : Vertex

❖ Finds the best move between current vertex and possible successor in rollout tree +tourDistanceCalculator(Tour currentTour) : void ❖ Calculates the total distance of rollout tour.

Map and Map Components

Map
Variables -vertexArrayList : ArrayList<Vertex> ❖ holds all vertices in an arraylist
+Map() : void ❖ construct a new arraylist of vertices +size() : int ❖ gets the size of the vertexArrayList +addVertex(Vertex vertex) : void ❖ adds a vertex to the arrayList +addEdge(int ID_source, int ID_dest) : void ❖ adds an edge between two vertices +hasEdge(int ID_source, int ID_dest) : boolean ❖ checks if there is an edge between two vertices +getHead() : Vertex ❖ gets the first vertex in the arraylist +getVertex(int idx) : Vertex ❖ gets a specific vertex from the arraylist +getLast() : Vertex ❖ gets the last vertex from the arraylist +unvisitAll() : void ❖ resets the status of all vertices in the arraylist +isAllVisited() : boolean ❖ checks if all vertices has been visited +completeConnect() : void ❖ construct a complete graph from the vertexArrayList

+printConnection() : void

- ❖ show all connections of between vertices

+computeDistance(Vertex v1, Vertex v2) : double

- ❖ calculates distance between two vertices

Path

Variables

-nodes : Integer[]

- ❖ array of all nodes

-distance : double

- ❖ distance between each nodes

-capacity : int

- ❖ capacity of each nodes

-pathList : List<Path>

- ❖ arraylist of paths

-ID : int

- ❖ ID of each nodes

Methods

+Path(Integer[] nodes, double distance, int capacity) : void

- ❖ Construct a new path

+getPathList() : List<Path>

- ❖ get list of path

+getDistance() : double

- ❖ get distance of path

+getCapacity : int

- ❖ get capacity of node

+toString() : String

+getNodes() : Integer[]

- ❖ get nodes from array

+getID() : int

- ❖ get ID of node

+setID(int ID) : void

- set or modify ID of node

+compareTo(Path o) : int

- compare between two paths

Edge

Variables

-destination : Vertex

- holds destination vertex

-dist : double

- distance of the edge

Methods

+Edge(Vertex destination, double dist) : void

- construct a new edge

+toString() : String

- converts output to String

+compareTo(Edge o) : int

- compare the distance of two edges

Tour

Variables

-totalDistance : double

- total distance of the tour

-route : List<List<Vertex>>

- list of the routes, uses the list of vertex

Methods

+ Tour() : void

- constructs a list of tours

+getTotalDistance() : double

- retrieve the total distance of the tour

+setTotalDistance(double totalDistance) : void

- set or modify the total distance of the tour

+getRoute() : List<List<Vertex>>

- get the route of the tour

+setRoute(List<List<Vertex>> route) : void

- set or modify the route of a tour

+toString() : String

- convert output to String

+distanceRoute(List<Vertex> vertexList) : double

- calculates the distance of a route

Vertex

Variables

-coordinateX : double

- holds x-axis part of the vertex coordinate

-coordinateY : double

- holds y-axis part of vertex coordinate

-capacity : int

- capacity (demand) of the vertex

-ID : int

- ID of the vertex

-visited : boolean

- status of the vertex, visited or not

-narrowArea : boolean

- status of the vertex area, narrow or not

-EdgeList : ArrayList<Edge>

- ArrayList of Edges

Methods

+Vertex() : void

- ❖ construct an empty vertex

+Vertex(double coordinateX, double coordinateY, int capacity, int ID) : void

- ❖ construct a vertex with details provided from parameters

+visit() : Vertex

- ❖ change status of vertex to visited, return the vertex

+unvisit() : void

- ❖ change the status of vertex to not visited

+isVisited() : boolean

- ❖ check if vertex is visited

+setNarrowArea(boolean narrowArea) : void

- ❖ modify the narrowArea status of a vertex

+isNarrowArea() : boolean

- ❖ checks if a vertex has narrowArea status

+toString() : String

- ❖ converts output to String

Sample Output

These are the results/outputs that we obtained in the Command Line Interface (CLI) and GUI for the text file inputs *n5-c20.txt* and *n19-c38.txt* respectively.

Command Line Interface (CLI)

```
Printing map... [================================================] 100.0%
Number of lorries = 0
C = 10
{ID 0} [NARROW: false]
Vertex ID0 [|-> {ID 1} [NARROW: false] d=57.21887800367987|| h=1, |-> {ID 2} [NARROW: false] d=86.64871608973787|| h=8, |-> {ID 3} [NARROW: false] d=61.66036003787198|| h=6, |-> {ID 4} [NARROW: false] d=24.207436873820411| h=5]
Vertex ID1 [|-> {ID 0} [NARROW: false] d=57.21887800367987|| h=0, |-> {ID 2} [NARROW: false] d=41.400483088968905|| h=8, |-> {ID 3} [NARROW: false] d=5.656854249492381|| h=6, |-> {ID 4} [NARROW: false] d=42.941821107167776|| h=5]
Vertex ID2 [|-> {ID 0} [NARROW: false] d=86.64871608973787|| h=0, |-> {ID 1} [NARROW: false] d=41.400483088968905|| h=1, |-> {ID 3} [NARROW: false] d=42.5440947723653|| h=6, |-> {ID 4} [NARROW: false] d=64.32728814430156|| h=5]
Vertex ID3 [|-> {ID 0} [NARROW: false] d=61.66036003787198|| h=0, |-> {ID 1} [NARROW: false] d=5.656854249492381|| h=1, |-> {ID 2} [NARROW: false] d=42.5440947723653|| h=8, |-> {ID 4} [NARROW: false] d=48.373546489791295|| h=5]
Vertex ID4 [|-> {ID 0} [NARROW: false] d=24.207436873820411| h=0, |-> {ID 1} [NARROW: false] d=42.941821107167776|| h=1, |-> {ID 2} [NARROW: false] d=64.32728814430156|| h=8, |-> {ID 3} [NARROW: false] d=48.373546489791295|| h=6]
Starting up simulations...  [================================================] 100.0%
Searching using Basic Simulations...  [================================================] 100.0%

---Revised Depth-First Search---
0 2 6
1 5 6
2 3 5
Tour cost : 346.2483982181608
Vehicle 1
{ID 0} --> {ID 2} --> {ID 0}
Capacity : 8
Cost : 173.29743217947575
Vehicle 2
{ID 0} --> {ID 1} --> {ID 3} --> {ID 0}
Capacity : 7
Cost : 124.53609229104424
Vehicle 3
{ID 0} --> {ID 4} --> {ID 0}
Capacity : 5
Cost : 48.41487374764082
Execution time: 1.9999999999999998E-5ms


---Blind Depth-First Search---

Tour Cost: 420.98628823988776
--------------------
Vehicle 1
{ID 0} [NARROW: false]  --> {ID 4} [NARROW: false]  --> {ID 1} [NARROW: false]  --> {ID 0} [NARROW: false]
Capacity: 6
Cost: 124.36813598466804
--------------------
Vehicle 2
{ID 0} [NARROW: false]  --> {ID 3} [NARROW: false]  --> {ID 0} [NARROW: false]
Capacity: 6
Cost: 123.32072007574396
--------------------
Vehicle 3
{ID 0} [NARROW: false]  --> {ID 2} [NARROW: false]  --> {ID 0} [NARROW: false]
Capacity: 8
Cost: 173.29743217947575

Execution time: 0.5384ms
Searching using Greedy Simulation...  [================================================] 100.0%
---Dijkstra's Search---

Tour Cost: 346.2483982181608
--------------------
Vehicle 1
{ID 0} [NARROW: false]  --> {ID 4} [NARROW: false]  --> {ID 0} [NARROW: false]
Capacity: 5
Cost: 48.41487374764082
--------------------
Vehicle 2
{ID 0} [NARROW: false]  --> {ID 1} [NARROW: false]  --> {ID 3} [NARROW: false]  --> {ID 0} [NARROW: false]
Capacity: 7
Cost: 124.53609229104424
--------------------
Vehicle 3
{ID 0} [NARROW: false]  --> {ID 2} [NARROW: false]  --> {ID 0} [NARROW: false]
Capacity: 8
Cost: 173.29743217947575

Execution time: 0.4714ms

---A* Search---

Tour Cost: 346.2483982181608
--------------------
Vehicle 1
{ID 0} [NARROW: false]  --> {ID 4} [NARROW: false]  --> {ID 0} [NARROW: false]
Capacity: 5
Cost: 48.41487374764082
--------------------
Vehicle 2
{ID 0} [NARROW: false]  --> {ID 1} [NARROW: false]  --> {ID 3} [NARROW: false]  --> {ID 0} [NARROW: false]
Capacity: 7
Cost: 124.53609229104424
--------------------
Vehicle 3
{ID 0} [NARROW: false]  --> {ID 2} [NARROW: false]  --> {ID 0} [NARROW: false]
Capacity: 8
Cost: 173.29743217947575

Execution time: 0.425ms
```

```
---Best-First Search---

Tour Cost: 357.0036710057714
Vehicle 1
{ID 0} [NARROW: false]  --> {ID 2} [NARROW: false]  --> {ID 1} [NARROW: false]  --> {ID 0} [NARROW: false]
Capacity: 9
Cost: 185.26807718238663
Vehicle 2
{ID 0} [NARROW: false]  --> {ID 3} [NARROW: false]  --> {ID 0} [NARROW: false]
Capacity: 6
Cost: 123.32072007574396
Vehicle 3
{ID 0} [NARROW: false]  --> {ID 4} [NARROW: false]  --> {ID 0} [NARROW: false]
Capacity: 5
Cost: 48.41487374764082

Execution time: 0.6622ms

---Best-Path Search---

Tour Cost: 346.2483982181608
Vehicle 1
{ID 0} [NARROW: false]  --> {ID 4} [NARROW: false]  --> {ID 0} [NARROW: false]
Capacity: 5
Cost: 48.41487374764082
Vehicle 2
{ID 0} [NARROW: false]  --> {ID 1} [NARROW: false]  --> {ID 3} [NARROW: false]  --> {ID 0} [NARROW: false]
Capacity: 7
Cost: 124.53609229104424
Vehicle 3
{ID 0} [NARROW: false]  --> {ID 2} [NARROW: false]  --> {ID 0} [NARROW: false]
Capacity: 8
Cost: 173.29743217947575

Execution time: 0.2132ms

Searching using MCTS Algorithm...  [================================================] 100.0%
-----NRPA-MCTS-----

Total Cost: 346.2483982181608
Tour
--------------------
Vehicle 1
[{ID 0} [NARROW: false] , {ID 3} [NARROW: false] , {ID 1} [NARROW: false] , {ID 0} [NARROW: false] ]
Cost: 124.53609229104424
--------------------
Vehicle 2
[{ID 0} [NARROW: false] , {ID 4} [NARROW: false] , {ID 0} [NARROW: false] ]
Cost: 48.41487374764082
--------------------
Vehicle 3
[{ID 0} [NARROW: false] , {ID 2} [NARROW: false] , {ID 0} [NARROW: false] ]
Cost: 173.29743217947575


Process finished with exit code 0
```

Graphical User Interface (GUI)