‘Always On Time’ Delivery

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, don’t 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.

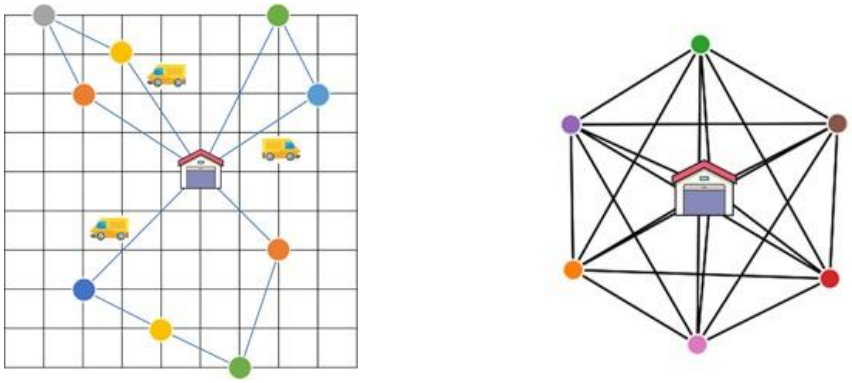


Figure 1: Visualization of the problem’s underlying structure

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.

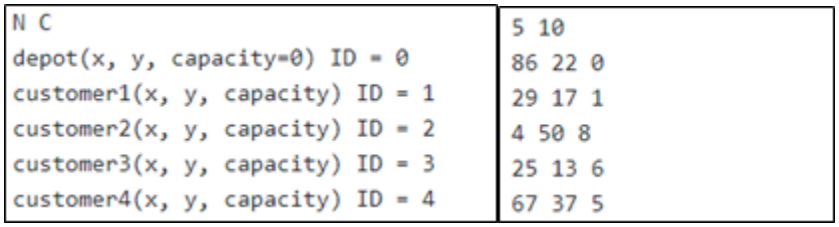
Basic Requirements

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 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 the Fig. 1 above.

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 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 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 second customer at (4, 50) with demand size 8, etc.

Noted 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.

Output

There are total of 3 outputs you have to generate, we define each output as a **simulation**:

1. Let's start with a simple one. 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 as '***Basic Simulation***'.

2. 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 looks for the shortest distance between current location and all next possible location to select the next location to go. A simple explanation will be given in the Github link provided in 'Resource' section. We name this approach as '***Greedy Simulation***'.

3. Now, this is the most important part of this simulation. We want to search for the **best tour that we could search of 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 of 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 as '***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                Greedy Simulation               MCTS Simulation
Tour                            Tour                            Tour
Tour Cost: 346.2483982181608    Tour Cost: 420.98628823988776   Tour Cost: 346.2483982181608
Vehicle 1                       Vehicle 1                       Vehicle 1
0 -> 4 -> 0                     0 -> 4 -> 1 -> 0                0 -> 3 -> 1 -> 0
Capacity: 5                     Capacity: 6                     Capacity: 7
Cost: 48.41487374764082         Cost: 124.36813598466804        Cost: 124.53609229104424
Vehicle 2                       Vehicle 2                       Vehicle 2
0 -> 2 -> 0                     0 -> 3 -> 0                     0 -> 4 -> 0
Capacity: 8                     Capacity: 6                     Capacity: 5
Cost: 173.29743217947575        Cost: 123.32072007574396        Cost: 48.4148737476408
Vehicle 3                       Vehicle 3                       Vehicle 3
0 -> 3 -> 1 -> 0                0 -> 2 -> 0                     0 -> 2 -> 0
Capacity: 7                     Capacity: 8                     Capacity: 8
Cost: 124.53609229104421        Cost: 173.29743217947575        Cost: 173.29743217947575
```

Figure 3: Example output based on the input file in Fig. 2. You may output your simulations one after another instead of side by side. It is put as side by side here just to save spaces.

Monte Carlo Tree 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 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.

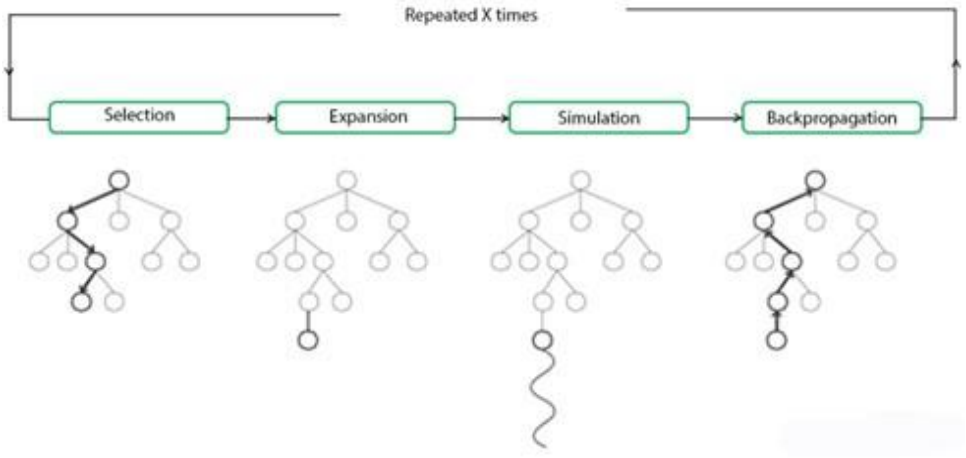


Figure 4: The Monte Carlo Tree Search process.

- **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 backpropagates 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.

For your knowledge, 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).

Resources

I uploaded some of the sample inputs on my personal GitHub repository, where you can access it through this link: **Delivery-Instances (GitHub Repo)**. You might want to test your program using the sample inputs provided there or you can simply generate inputs randomly for your own sake. The sample inputs provided there follow the input format explained earlier and if your program is receiving a different input format then you might need to change those samples’ format after downloading them as well.

Inside the repository you will also find the implementation of MCTS, in pseudo-codes form, and I believe you guys have the ability to figure out the complete implementation.

If you have any questions please feel free to raise it in the GitHub repo by creating a new issue. I will check the repo from time to time and try to answer your questions. If you are unfamiliar with GitHub, **here** is a documentation guide on how to create a new issue in a GitHub repo.

Extra: I might be putting more and more resources regarding the problems (some possible new ideas for extra features of this problem or explanations about MCTS as well) for your reading. If you are interested, can try checking the GitHub repo from time to time.

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 in 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 pickups 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 customer

In real life, we not only have to minimize the time and fuel (represented by distance) used by couriers, 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 algorithm you might want to consider:

1. Best First Search
2. A* Search
3. Genetic Algorithm
4. Hybrid Search I (MCTS + GA)
5. Hybrid Search II (MCTS + ML)
6. 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 some times if you animate your simulation), you can continue to search the next best move from the next location so that you don't have to waste so much time initially for searching the best whole route for every vehicle before they depart.

END OF THE QUESTION