


2021

VEHICLE ROUTING PROBLEM WITH TIME WINDOWS

PLOTWISE INTERVIEW TEST

OMID GHOLAMI
GHOLAMI.O@YAHOO.COM



Contents

Abstract	2
1. Introduction	2
2. Selected Approach	2
3. Modeling the problem	3
4. A brief overview of the algorithm.	5
4.1 Neighborhood search.	6
5 Implementation	6

Vehicle Routing Problem with Time Windows

Abstract

In this report, I am going to review the Vehicle Routing Problem (VRP) with time windows and explain a method used by me to tackle the problem. Vehicle Routing Problems with Time Windows (VRPTW) can be defined as choosing routes for a limited number of vehicles to serve a group of customers in the allocated time windows. Each vehicle has a limited capacity. It starts from the depot and terminates at the depot. Each customer should be served exactly once. In our problem, we assume that the distances of the delivery points are symmetrical. Additionally, we allow the delivery man to wait at a delivery point, if necessary, for a time window to open. In literature, different approaches have been presented in the last three decades, which I will briefly review and talk about the pros and cons of each one.

1. Introduction

The vehicle routing problem is a combinatorial optimization that is classified as an NP-hard problem. It means that finding an optimal solution for a large-size problem could be impossible and infeasible. But there are many exact algorithms introduced to tackle the problem. These algorithms try to solve the problem optimally using techniques such as Branch and bound, Branch and Cut, or Numerical methods (For this case Mixed-Integer Linear Programming). Although, these algorithms can provide the best results the use of exact optimization methods may be difficult for solving VRP problems in an acceptable CPU time when the problem involves real-world data sets that are very large.

The second type is metaheuristics (or better say the evolutionary ones) such as Ant Colony, Genetic Algorithms, Simulated Annealing, and Tabu Search. In this category, we build one or more solutions for the problem, and by using some evolutionary techniques we try to improve the result. Metaheuristics sample a subset of solutions that is otherwise too large to be completely enumerated or otherwise explored. Metaheuristics do not guarantee that a globally optimal solution can be found on some class of problems, and usually are faster than numerical methods but the result may not be optimal as the algorithm can be stuck in local optimal.

The last type of algorithm is heuristics which tries to solve the problem using one or more metrics more often in a greedy manner and build the solution regarding those metrics. This is achieved by trading optimality, completeness, accuracy, or precision for speed. This type of algorithm is fast but the result could be far from optimal solution considering our objective function. Also, such algorithms may work well for one objective function but not good if you consider other metrics. So, a developed algorithm can't be used for different scenarios.

2. Selected Approach

We know that the exact methods are not a good choice for vehicle routing problems in real life as the number of customers is high which makes it infeasible. Also, usually, we should come with a solution so fast (in real-time scenarios an interval of some seconds to a minute is acceptable). Also, in this problem, as we have a time constraint, the number of constraints increases a lot which makes it impossible for a solver to come with a solution in a feasible time for a numerical solver. Figure 1, gives an idea about how

big could be a solution space for VRPTW to VRP. We consider that most permutations should be rejected due to the infeasibility of the time window.

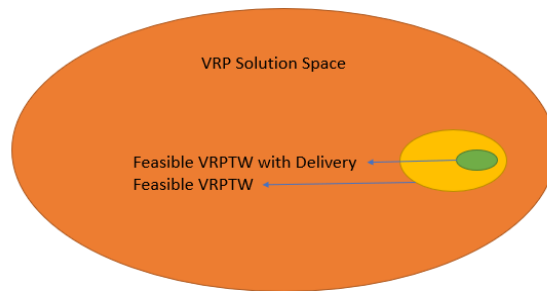


Figure 1: Feasible VRTW Solution.

On the other hand, I am not a fan of metaheuristics for this problem. I believe that evolutionary operators will generate many infeasible solutions as we have a time window in our problem. So the computational time will increase and the chance to be stuck inside a local optimum is high.

As a solution for the problem, I am planning for a two-stage algorithm which in the first stage tries to solve the problem **heuristically** and then by using **neighborhood search** tries to improve the solution. In this approach we are fast enough to have a solution in hand, then we can control how long we want to let the neighborhood search elaborate the answer. It means that we can stop if the solution is satisfactory enough or the due date has been reached. Also, we will not be drawn in infeasible solution space which I think is large as we have time window constrain (Figure 1). Also, we have a new constraint in the problem defined by Plotwise which *each delivery vehicle can only have one pickup task* maximum, which also reduces the feasibility space. Simply saying I prefer to struggle in the green area.

3. Modeling the problem

In VRP, we model each delivery point by assigning an x and y coordination to it. Then in a 2D space, we try to calculate the distance between every two points. The distance between two points would be the length of the path connecting them and the shortest path is a straight line. In a 2 dimensional space, the distance between point $A=(X1, Y1)$ and $B=(X2, Y2)$ is given by the Pythagorean theorem: $d=\sqrt{(x2-x1)^2+(y2-y1)^2}$.

In a vehicle routing problem with time windows, we have a time constraint that should be satisfied. It means that each delivery should happen in the time window of ready time and the due date. If the delivery vehicle arrives later it fails to deliver. If it arrives earlier the vehicle should wait until the ready time. Waiting time for a delivery vehicle is a cost for us that we could use that time to serve other customers. I did not like to generate the solution then check if it is feasible time window wise but I was keen to add this time window in a way to our 2D model of the problem that helps me to find a better solution and not choosing the point randomly (that can cause a big number of permutations).

The idea which worked for me was to see ready time as Z dimension. In this case, I consider each unit of time as one unit of distance. This assumption gives me the possibility to model my deliveries in a 3D space as $\{x, y, z\}$, which z is the ready time for that delivery task (the one to one unit conversion can change based on the problem in real life, even it can vary between different points).

Let's see how the problem has been formulated in a 3D space. In figure 2, you can find three points $A=(X1, Y1)$, $B=(X2, Y2)$, and $C=(X2, Y2)$. Two deliveries (B and C) are in the same position $\{x2=x3, y2=y3\}$. But they have different ready times ($z2=0$, and $z3=10$). So if we consider a 2D distance they are the same place, but by considering the ready time, the third one is far from the second one and we should wait for the third one. When I am modeling the problem in a 3D space, I can interpret this waiting time as a distance. So, by computing the distances in a 3D space ($d=\sqrt{(x2-x1)^2+(y2-y1)^2+(z2-z1)^2}$) I take into account the waiting time.

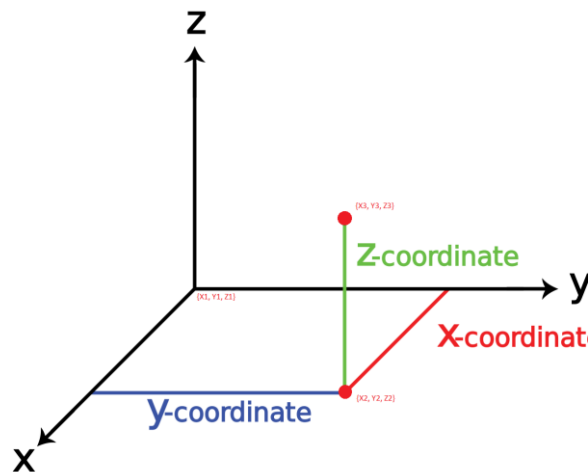
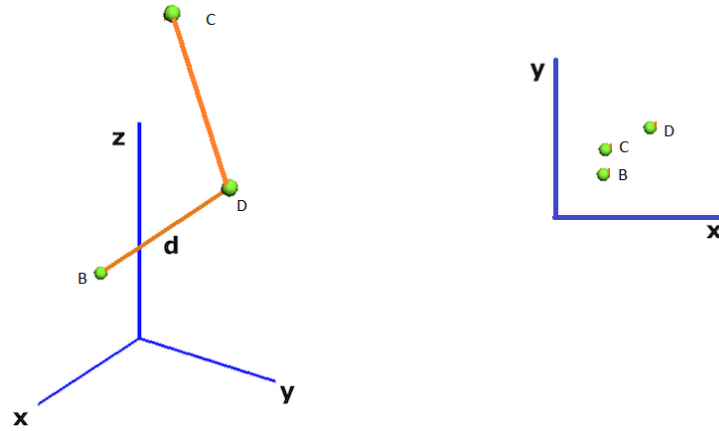


Figure 2: Two deliveries in the same place with different ready times.

This means that my new model considers the third point farther than the second one (doesn't matter if I should drive to a new place or wait in the same place, I could use this time to serve other deliveries). That means, there could be another point (D) which in 2D space is far from B and C but in a 3D space, it is closer to B compared to C. That is because we had to wait a lot of time at B to serve C, but now we can go to D and then come back for serving D.

As an example consider 4 points. $A=\{0,0\}$, $B=\{2,2\}$, $C=\{2,3\}$, $D=\{4,4\}$ with ready time of 0, 2, 10, and 5 respectively. In 2D space, if we use a greedy algorithm that serves based on the nearest job first we have to make a tour as A, B, C, D, A (A is the depot). But even B and C are near each other the ready time gap between them is large and that makes it possible to serve other deliveries and come back. This gap time modeled using the Z dimension, and the new positioning in a 3D space is $A=\{0,0,0\}$, $B=\{2,2,2\}$, $C=\{2,3,10\}$, $D=\{4,4,5\}$. In a 3D space, the heuristic algorithm chooses A, B, D, C, A. See Figure 2, C is closer to B than D in a 2D space but it is not the case when we consider ready time (Z dimension).



4. A brief overview of the algorithm.

The entry point of the algorithm is TSPManager class. This manager accepts an address of a benchmark file to read the input data. The manager has one public method `runSolvers()` which is the starting point of the algorithm. After reading the data (`getVehiclesData()`, `getDeliveryData()`), the algorithm marks some of those delivery points as **PickUp** points as Plotwise bring that in problem explanation. I considered 10% of the delivery tasks as a pickup point. Also, the first delivery point has been marked as the depot.

In the next step, we create a `HeuristicTSP` object and call the only public method which is `execute()` method. As each pickup point should be served by one vehicle we start with the same number of the vehicles as pickup points. We try to create a feasible solution and if we were not successful we add one new vehicle (The number of needed vehicles may reduce later in the neighborhood search).

The heuristic algorithm (`distributeDeliveries()`) works as follows:

- Sort all tasks based on their 3D distance to the depot. It means that we consider their x and y condonation and ready time.
- Assign first n task to n available vehicles.
- Loop: find the next (list is sorted 3D) unassigned task ($task_i$).
- Try to assign the task to the nearest vehicle. The nearest vehicle is a vehicle whose last delivery job is closer to the task in a 3D space.
- Try to assign $task_i$ to that vehicle (`assignDeliveryToVehicle`), if feasible go to the start of the loop, if not, mark that vehicle as tabu (this vehicle should not be selected next time if we fail to assign $task_i$ in this loop).
- If we could not find any vehicle for $task_i$ this assignment is infeasible, we should add one more vehicle and start again.
- Else, mark $task_i$ as done and continue with the next task.

The assignment task (`assignDeliveryToVehicle`) considers all possible variants and selects the feasible one with minimum cost. For example, if we have $\{t_1, t_2, t_3\}$ and we want to add t_4 , the algorithm considers 4 points of insertion (X_i) and chooses the one with minimum cost ($\{X_1, t_1, X_2, t_2, X_3, t_3, X_4\}$). The cost is the total time traveled by vehicle to serve all delivery tasks.

4.1. Neighborhood search.

The result provided by the heuristic algorithm now can be passed to the neighborhood algorithm as the initial solution for neighborhood search. In the literature, you can find different crossover and mutation methods but in my case the idea of neighborhood search is simple: try to transfer each delivery task_i from vehicle_m to vehicle_n and assign it to the best place in that delivery task (**assignDeliveryToVehicle**). This neighborhood strategy applies to mixes two strategies: the first one is finding the best vehicle to shift the task and the second one is finding the place in a task to handle the job. If the assignment is feasible and the cost of these two new routes is less than the old delivery plan, then accept this transfer:

$$(\text{cost}(\text{vehicle}_{m_old}) + \text{cost}(\text{vehicle}_{n_old}) < \text{cost}(\text{vehicle}_{m_new}) + \text{cost}(\text{vehicle}_{n_new})).$$

The algorithm continues transferring tasks until no new shift happens in the algorithm (in the large datasets maybe we can stop the algorithm by some strategies such as due date or percentage of improvement). One big benefit of neighborhood search is that we can easily change the cost function based on the customer demands or even use a mix of objective functions. In my algorithm, two cost functions were introduced:

- Total distance traveled by all vehicles.
- Total time traveled by all vehicles.

Results are as follows using benchmark C1_2_1:

- The result from the **heuristic algorithm**:
Total Traveled Time: 445:55 (hh:mm), Total Distance Traveled: 5906 km, Number Of Used Vehicles: 29.
- The result from the **neighborhood search** algorithm with optimization of **TIME TRAVELLED**:
Total Time Traveled: 427:41 (hh:mm), Total Distance Traveled: 6124 km, Number Of Used Vehicles: 27.
- The result from the **neighborhood search** algorithm with optimization of **DISTANCE TRAVELLED**:
Total Time Traveled: 457:20 (hh:mm), Total Distance Traveled: 4437 km, Number Of Used Vehicles: 27.

As you can see both the “TIME TRAVELLED” and “DISTANCE TRAVELLED” objectives have been satisfied using local search and the number of needed vehicles reduced by 2.

5. Implementation

The algorithm is written in C++ using Microsoft visual studio 2019 environment and compiler. You can find two projects in the solution folder, “**Plotwise**” which is the algorithm implementations (run it using F5), and “**Test**” which is a unit test provided for the application (you can run the test using test explorer windows in VS). For the test part, I could support more scenarios and test cases but I made it short due to lack of time. The benchmark file is in the “benchmark” folder.

- This document and the developed algorithm are written by Omid Gholami as a job interview test for Plotwise company, Netherlands.