Contents

[Introduction: 1](#_Toc57749063)

[Procedure: 2](#_Toc57749064)

[Structs used in the code: 2](#_Toc57749065)

[Task struct: 2](#_Toc57749066)

[Node struct: 2](#_Toc57749067)

[Files style: 3](#_Toc57749068)

[The coordinates file: 3](#_Toc57749069)

[The tasks file: 3](#_Toc57749070)

[Reading from files: 3](#_Toc57749071)

[MRV function: 4](#_Toc57749072)

[Backtracking function: 5](#_Toc57749073)

[Discussion ad simulation results: 7](#_Toc57749074)

[Sample test 1: 7](#_Toc57749075)

[Simulation results: 7](#_Toc57749076)

[Conclusion: 8](#_Toc57749077)

# 

# Introduction:

The Single Vehicle Pickup and Delivery Problem with Time Windows (PDPTW):

The single vehicle PDPTW deals with a number of customer requests that must be satisfied by one vehicle with a known capacity. The route of the vehicle starts with a central depot which isn’t a pickup or a delivery location. A request must be collected from a pickup location before being dropped off at a corresponding delivery location, and every pickup and delivery request is associated with a specific time window during which it must be served. If the vehicle arrives earlier than the beginning of the designated time window interval, it must wait until the service time begins. All requests must be served in a way that it doesn’t violate capacity and time windows constraints.

The time window for a delivery location is the same as its pickup location and a simple location can be visited only once and it can be either a pickup or delivery or both observing that it can’t be a pickup for more than one location and can’t be a delivery for more than one location too, so in the case of being both pickup and delivery the location X is concerned with maximum of 2 other location the delivery that its pickup is at X and pickup that must deliver to X and in this case all three locations must have the same time window. In addition, for simplicity all pickups are assumed to contain the same demand which is one, when taking pickups from multiple locations without delivering previous ones the load of the car must not exceed its capacity.

Our formalization for solving this problem:

Variables: we assumed that variables are the locations which represented using nodes, locations are given (x, y) coordinates and the time to travel between location to other is simply the direct distance between them.

Domains: the domain for each variable (location) is the time that is available from arriving to the location and the end of its time window.

Constraints: all locations must be served without violating the capacity and the time window. For time window, travel time to the next location is taken into consideration so we need to take a path that doesn’t exceed the end of the time window.

# Procedure:

First of all, we will begin with defining two structs which will contain the details for each node and for each task.

## Structs used in the code:

### Task struct:

Attributes:

1. Picked at: which identify the time in which the task is being picked up.
2. Delivered at: which identify the time in which the task is being delivered.
3. Destination: which identify the value of the nod where the task is to be delivered.
4. Source: which identify the value of the nod where the task is to be picked up.
5. Start: which identify the starting time of the time window.
6. End: which identify the ending time of the time window.
7. Id: a unique id for each task.

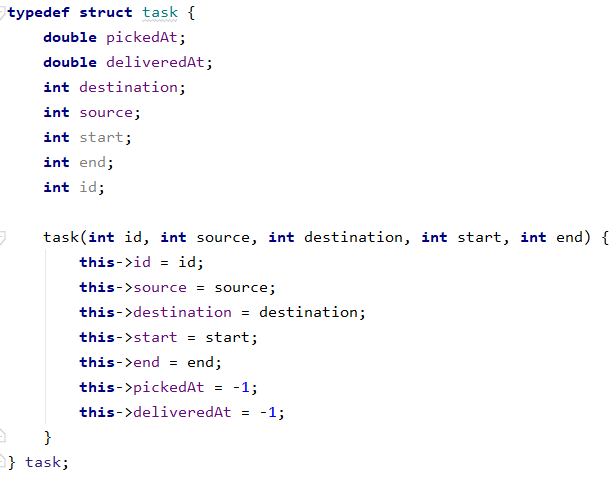


Figure : Task Struct

### Node struct:

Attributes:

1. Pickup and delivery: two Boolean values indicates if the current node is either delivery or pick up or the both.
2. X and Y: two double values which identify the location of the node with x and y coordinates.
3. Id: a unique id for each node.
4. Start: which identify the starting time of the time window.
5. End: which identify the ending time of the time window
6. taskD: contains the Id of the task in which the current node is the delivery node in it.
7. taskP: contains the Id of the task in which the current node is the pickup node in it.

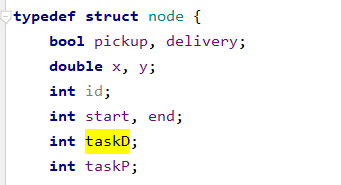


Figure : Node Struct

## Files style:

### The coordinates file:

the file is designed as follows:

the first line contains an integer n (the number of locations)

each line of the next n lines contains x and y the x and y coordinates of the location.

### The tasks file:

the file is designed as follows:

the first line contains an integer m (the maximum capacity of the vehicle)

each line of the next lines contains the source and the delivery and the time window for each task.

## Reading from files:

Raining process is done as follows:

## MRV function:

As the MRV algorithm states that we should choose the next variable depending on the having the least number of values.

In our problem the variable that has the least number of the values is the one that the travel time to it plus the current time is the nearest to the end time of its time window.

So now we need to sort all the unvisited children of the location depending on the difference between its end time and the time after reaching it.

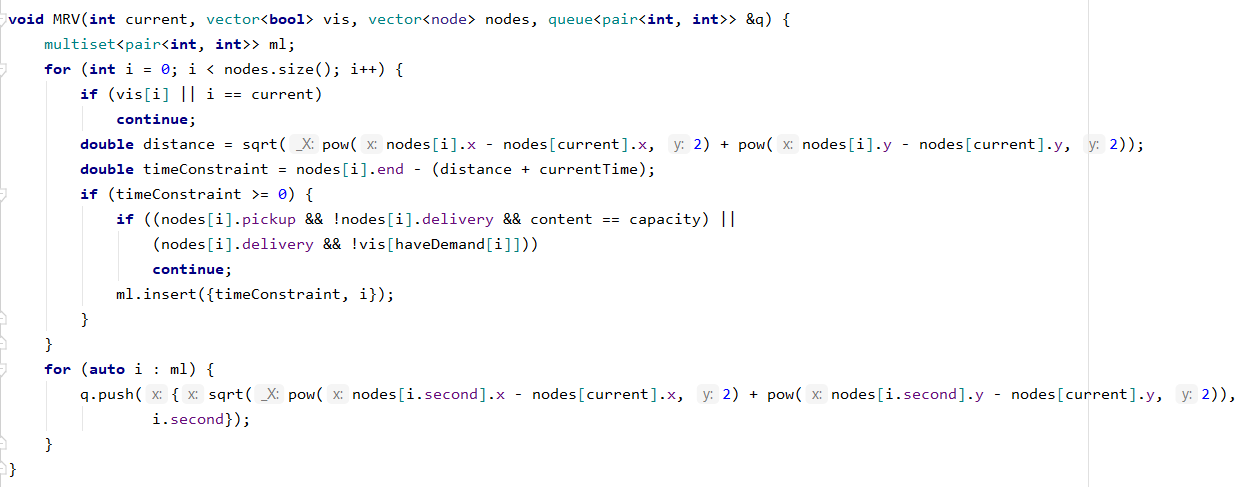


Figure : MRV Function Implementation

As shown in the previous code, first of all a multiset is created because it keeps it self sorted after each insertion.

Secondly, we will loop through all the children of the current node with skipping the locations which are already visited or if the location is a delivery location and its demand has not taken yet. Now the remaining locations are the ones we can move to, then all the reaming locations will be inserted into the multi set as a pair with the difference between their and time and the theoretical time when reaching it, after looping through them all the multi set now has all the locations with their time difference sorted ascending, but the problem now that we do not need this difference but we need the actual time needed to travel. To bypass this problem, we will iterate over those pairs of locations again from the multiset and insert them again to a queue as a pair containing the id of the location and the required time to reach it from the current node.

Notes:

* We cannot depend on the required time to reach a node to determine which node we should visit next because the travel time might be very large but also the time window is very large hence this is not a good estimate.
* A multiset containing pairs is always sorted ascending depending on the first value of the pair, for that we put the time difference in the beginning of the pair.
* We used a queue due to its property FIFO, since we are inserting the locations having the highest priority first and then popping them one by one from the front, so this is the best data structure that fits with this application.

## Backtracking function:

The concept behind the back-tracking function is very simple, the only thing we do is to try to move to every available location from the one we are now in the order found by the MRV as was explained before.

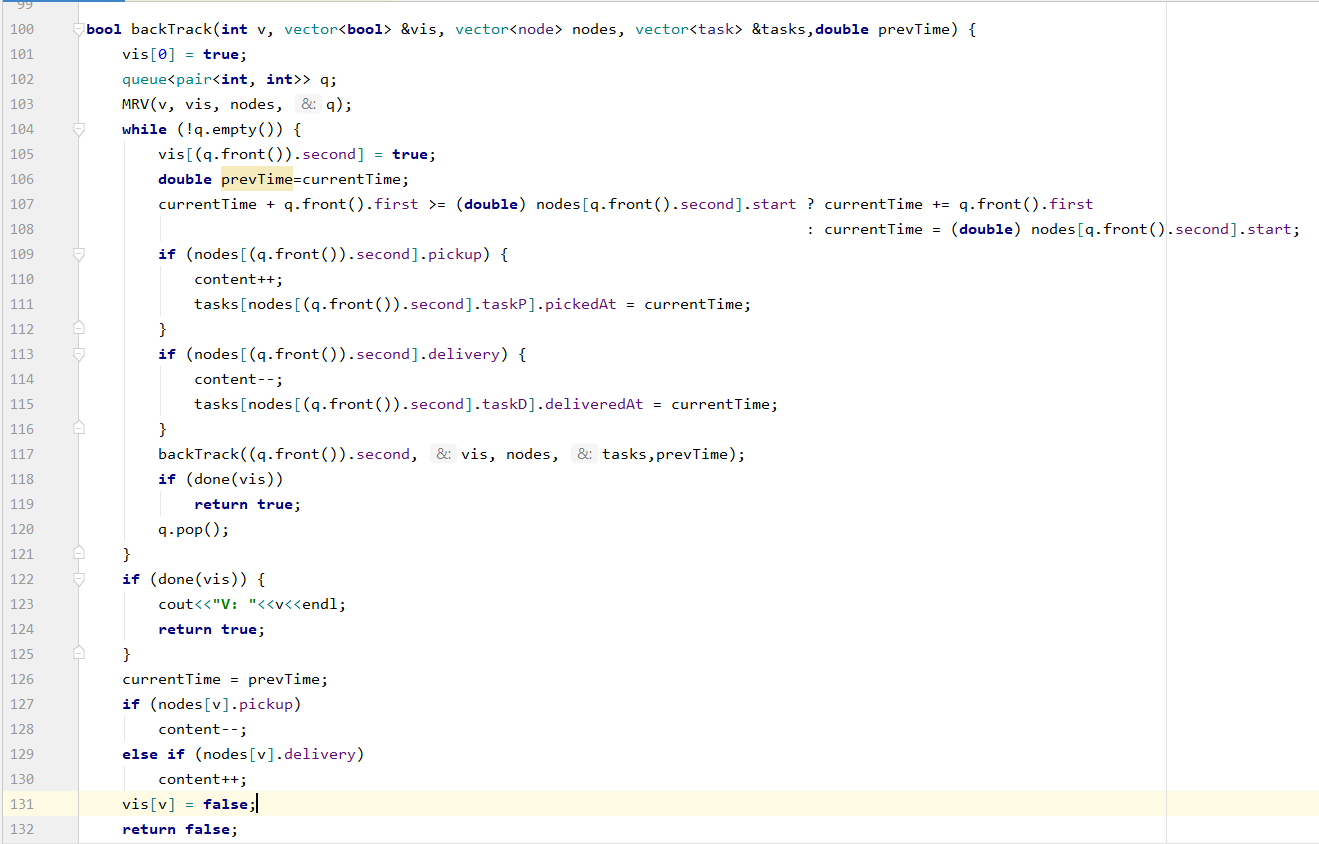


Figure : Backtracking Function Implementation

As shown in the previous code, first of all the MRV function is invoked and the results got from it is saved into a queue called q, after this we will start looping through the locations in the queue ( which are sorted already by the MRV function), for every location we will call the back track function again by marking the node as visited and taking its delivery if it is a pick up location or delivering its delivery if it is a delivery location, like this the function will keep calling itself for every location in the queue and whenever we reach a dead end we will return the time to the previous time before calling the function at this level , also the location will be remarked as not visited more than that if the location is a pick up one the demand is returned to it including if it is a delivery location the demand is returned from it and then we will try the other locations on the previous level.

Every call for the function we call a function called “done” that will check if every location in the tasks is visited, if so then all the tasks are completed successfully, otherwise the function will keep trying by moving to the next location in every level. At the end if it returns to the initial location after expanding all its children it will return false indicating that the problem has no solution under this constrains.

# Discussion ad simulation results:

## Sample test 1:

We will begin with a simple test that does not require any back tracking.

Coordinates file content:

4  
 0 0   
 0 10  
 0 20  
 0 30

The tasks file is as follows:

1  
 1 2 0 20  
 2 3 0 30

### Simulation results:

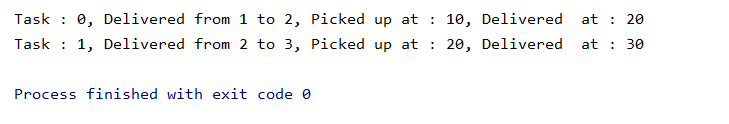


Figure 5:TEST 1 Simulation result

As we can see the two tasks were done successfully during their available time window and without any conflict between them.

# Conclusion:

At the end we can say that the project was a very interesting one because it takes us from the theoretical part of the algorithms to a real-life application requiring algorithms like this to arrange the tasks required from it.

Also, we recognized how the algorithms used like the MRV reduces a lot of backtrackings that could happen without it; however, those algorithms are not sufficient to solve like this problem in an optimal way when the state space is very large.