PROJECT REPORT ON

# “VRPPC (Vehicle-routing Problem with Private fleet and Common carrier)”


**Submitted by :**

**Sayan Sarkar**

[Roll No. :- 90/MCA/200017; Registration No.:- 100318 of 2020-2021]

**Sayandip Adhikary**

[Roll No. :- 90/MCA/200018; Registration No.:- 100319 of 2020-2021]

Under the guidance of

**Dr. Priya Ranjan Sinha Mahapatra**

Professor, Department of Computer Science and Engineering,
University of Kalyani

**Dr. Soumen Atta**

Postdoctoral Researcher,
University of Nantes, France

**University of Kalyani**
Kalyani , Nadia
West Bengal

# University of Kalyani

Faculty Of Engineering Technology and Management

Department of Computer Science & Engineering

**Dr. Anirban Mukhopadhyay**
**Professor and Head**
Dept. of Comp. Sc. & Engg

Kalyani, Nadia – 741235
West Bengal, India
E-mail: hodcomputer_sci@klyuniv.ac.in

Date : 25. 07. 2022

## Certificate

This is to certify that the project report entitled "**VRPPC(Vehicle Routing Problem with Private fleet and Common Carrier)"**, submitted to Department of C.S.E, University of Kalyani in partial fulfilment of the requirement for the award of the degree of "Master of Computer Applications" is an original work carried out by **Sayan Sarkar**(Roll. **90/MCA** No. **200017**; Reg. **100318 of 2020-2021**) and **Sayandip Adhikary** (Roll. **90/MCA** No. **200018**; Reg. **100319 of 2020-2021**) under my guidance and supervision.

The matter in this project is a genuine work done by the student and has not been submitted elsewhere of any course of study. I permit them to submit it.

**Signature of the Co-supervisor**
**Dr. Soumen Atta**
Postdoctoral Researcher,
University of Nantes, France

**Signature of the Supervisor**
**Dr. Priya Ranjan Sinha Mahapatra**
Professor,
Dept. of Computer Science and Engg.,
University of Kalyani,
Kalyani, Nadia

**Signature of the External Examiner**

**Signature of the Head of Department**
**Dr. Anirban Mukhopadhyay**
Professor and Head
Dept. of Computer Science and Engg.,
University of Kalyani,
Kalyani, Nadia

# Declaration

I declare that this written submission represents my ideas in my own words and where others' ideas or words have been included, I have adequately cited and other referenced the original sources.

I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea / data / fact / source in my submission.


**Sayandip Adhikary**
Roll 90/MCA No. 200018
Reg. No. 100319 of 2020-2021
Dept. of Computer Science and Engineering
University of Kalyani

**Sayan Sarkar**
Roll 90/MCA No. 200017
Reg. No. 100318 of 2020-2021
Dept. of Computer Science and Engineering
University of Kalyani

# Acknowledgement

A project work is not an obligation done by one person towards a predetermined and very specific goal. Rather, it is the coming together of many elements, some direct & some indirect, towards the planning & implementation of a long-drawn effort. A project work requires hard works, perseverance and dedication and above all a lot of support, as only teamwork can make good, better and best. So right at the onset I would like to express my apologies to those whose names I have missed out to mention.

I am highly indebted to UNIVERSITY OF KALYANI and all my beloved teachers without whose moral help and extensive cooperation I would not be able to submit this report. My gratitude goes to our project supervisor Dr. Priya Ranjan Sinha Mahapatra, Professor of Computer Science and Engineering, Faculty of Engineering, Technology and Management, University of Kalyani and Dr. Soumen Atta, Postdoctoral Researcher, University of Nantes, France, for their valuable advice, guidance and unending support right from the stage and suggestions during the difficult phases of the project. Finally, yet important my sincere thanks goes to all who have directly and indirectly extended their valuable guidance and advice during the preparation of this report, which will give us the continuous flow of inspiration to complete this.

**Sayandip Adhikary**
Dept. of Computer Science
and Engineering
University of Kalyani

**Sayan Sarkar**
Dept. of Computer Science
and Engineering
University of Kalyani

# Content

# Introduction

The vehicle routing problem (VRP) is a combinatorial optimization and integer programming problem which asks "What is the optimal set of routes for a fleet of vehicles to traverse in order to deliver to a given set of customers?" It generalises the travelling salesman problem (TSP). It first appeared in a paper by George Dantzig and John Ramser in 1959, in which the first algorithmic approach was written and was applied to petrol deliveries. Often, the context is that of delivering goods located at a central depot to customers who have placed orders for such goods. The objective of the VRP is to minimize the total route cost. In 1964, Clarke and Wright improved on Dantzig and Ramser's approach using an effective greedy algorithm called the savings algorithm.

The vehicle-routing problem with private fleet and common carrier (VRPPC) is a variant of the VRP in which customers can be subcontracted at a customer-dependent cost if the privately-owned capacity is insufficient to serve all customers, or if doing so is beneficial from a cost point of view. Consequently, the subcontracted customers do not need to be served on vehicle routes of the privately-owned fleet, but a cost is paid for outsourcing customers to the so-called common carrier.

The VRPPC also has applications in the planning of same-day parcel deliveries. By choosing adequate customer-dependent outsourcing costs, important customers—e.g., subscribers of Amazon Prime or customers that have already been postponed on previous days—can be favoured over regular new requests.

The vehicle routing problem with private fleet and common carrier represent one of the most and important variant to be included in the larger family known as Vehicle Routing Problem . In literature this problem holds the attention of several researchers for many years. There is a large literature on the optimization of the vehicle routing which provide an overview of the VRP and its variants. To make a relation with our studied problem, the VRP have been tackled in two types: Firstly, several researchers have studied the VRP with limited fleet within the own vehicles. All these items using a mixed limited fleet but sufficient capacity to serve all customers. Secondly, the VRP with Private fleet and common Carrier are studied. At the level of routing problem with external carrier which demonstrated that the problem involving a fleet of one single vehicle and external carriers can be rewritten as Traveling Salesman Problem – TSP. The VRP with limited fleet whose objective was to decide that customers visited with external carrier and optimize the tour of remaining customers. It consider a variant of vehicle routing problem with multiple trip and time windows, the objective of the problem is about the scheduling of trucks to a number of customers in the presence of fixed fleet and time windows constraints.

To solve the Vehicle routing problem, we have first tried to get an initial solution based on Greedy approach(section – 3.iii). To make the routes more optimal, we have applied few algorithms which were heuristic based. Namely, a modified version of Two Opt algorithm has been implemented and another algorithm (section – 3.iv), Max Link Insertion(section – 3.x), to remove a customer who is located at far off place than the other customers visited by that particular fleet. Similarly we have applied, Adjacent swapping

(section – 3.v), one-one swapping (section – 3.vi) and single insertion (section – 3.vii) algorithms for modifying the customers within a particular fleet. We also applied an algorithm, External Node Insertion and Swapping (section – 3.viii & 3.ix), which inserts and swaps a node in the Private fleet from External fleet. In Section 5 we perform the experiments results. And finally the conclusion is in Section 6.

# Problem Definition

To define the VRPPC as a graph-theoretical problem, let G = (V, E) be a complete undirected graph with vertices $V = \{v_0\} \cup N$ and edges $E = V \times V$. Vertex $v_0$ denotes the depot, the other vertices represent customers $i \in N$ . Each customer $i \in N$ is assigned a demand qi and a cost $h_i$ for subcontracting the customer. Each edge $\{i, j\} \in E$ is assigned a travel cost $c_{ij}$ . At the depot, a set of vehicles K, which represent the private fleet, is based. The vehicles $k \in K$ can differ with regard to capacity $C_k$ and fixed cost $F_k$. The fixed cost $F_k$ is only incurred if a route is assigned to vehicle k. Typically, not all vehicles are different, and we can group the vehicles according to their attributes such that vehicles with identical attributes are in the same group $l \in L$. We denote the number of vehicles in group l as $z_l$ . The VRPPC now calls for: i) satisfying the demand of every customer with exactly one visit, either using the common carrier or a vehicle of the private fleet, and ii) planning at most one route for each vehicle of the private fleet so that every route starts and ends at the depot, and the vehicle capacity is respected. The goal is to minimise the total cost consisting of the sum of fixed cost, the cost of routing the vehicles of the private fleet, and the cost of subcontracting customers.

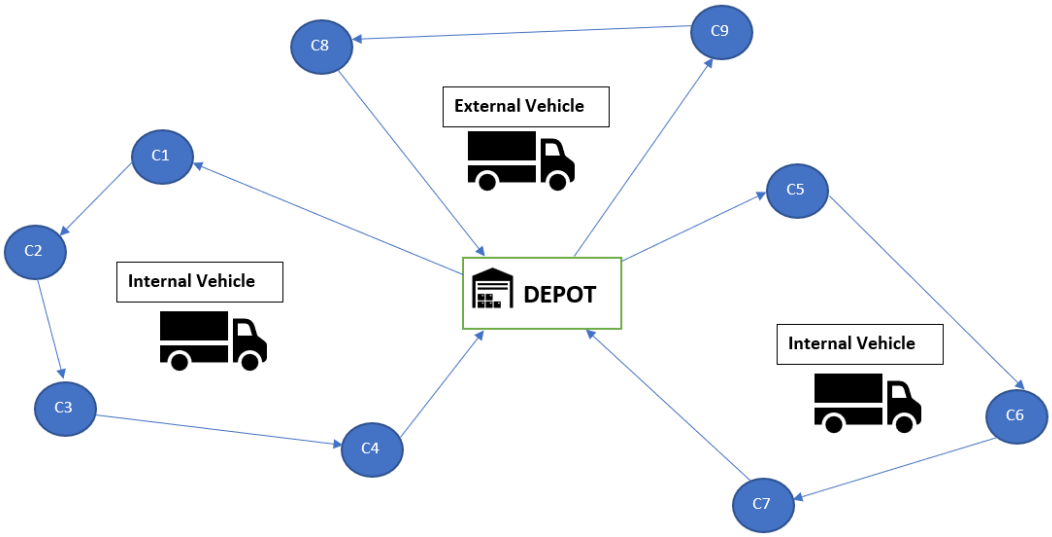


Fig 1 : Example of VRPPC problem

In this paper, we are studying the VRPPC in the following environment. One or more products are distributed when the all routes associated with the private fleet start and end at the depot. Each customer must be served once by the private fleet either by a common carrier; each private fleet vehicle performs only one route, the internal fleet is composed of a limited number of vehicles. The total demand of any route does not exceed the capacity of the vehicle assigned to it; finally, the total cost is minimized. In practice, several common carriers may be used to serve any of the customers unvisited by the private fleet. The capacity of vehicles is determined in terms of units produced. The vehicles have both a fixed cost of use and a variable cost depending on the distance traveled. The costs of external transport are set for each customer depending on its geographical location and quantities required. Fig. 1 provides an example of the VRPPC with definite characteristics such as the number of vehicle that equal to 2 vehicles for the common carrier and one vehicle as an external fleet. Also we suppose that there are 9 customers to be served.

## Data Reading and preprocessing

To solve this problem we were given text files, which is consists of Name of the Text Dataset, some comments, Number of customers, Number of vehicles, Details of every customer (i.e. Customer Number, Coordinates, Demands, External Cost) and Details of every vehicle (i.e. Truck ID, Capacity, Fixed Cost, Variable Cost).

First thing we had to do is reading the dataset using a function named *takeInputFile()*. Then we're reading the first five line of every dataset. Line number 3 and 4 was important for us because it holds the value of total customers and total vehicles respectively.

Declared two integer variables named *customer* and *vehicle*, storing the number of customers in customer and number of customers in vehicle. Based on customer and vehicle we're going to read next lines of the dataset and storing the data in respective arrays.

*customerArr* is declared to store the details of customers, the size of this array is *[(customer + 1) * 5]*. Then all the customer details are read and stored in the *customerArr* using *getCustomerData()* function.

Similarly, *vehicleArr* is declared to store the details of vehicles, the size of this array is *[vehicle * 4]*. Then all the vehicle details are read and stored in the *vehicleArr* using *getVehicleData()* function.

When reading of data is completed and stored in respective arrays, next part is to create a distance matrix of customer locations (using coordinates) using getDistanceMatrix() function which is a 2D vector.

# Objective Function

This is a function for computing the objective value after obtaining the routes.

The calculations are as follows :

i. Variable cost of a vehicle (A) = total distance traveled * VAR. COST

ii. Total cost for the internal fleet (B) = FIXED COST for the used vehicles + variable cost for the used vehicles (A)

iii. Objective value = B + Cost of external transporter for the external vehicles

## Initial solution ( Greedy Approach)

i. Find the sum of demand for all customers.

ii. Find the sum of capacity for all Internal Trucks.

iii. Calculate ratio of External Cost to Demand for each customer.

iv. Sort the above ratio in increasing order, keeping track of the customers too.

v. Store the Node(customer nos) which have lower ratio(External cost : Demand) in a data structure.

vi. Method for storing :

```
while (sumOfDemand > sumOfCapacity)
{
    externalFleet.push_back(node_having_lowest ratio);
    sumOfDemand -= demand_of_node_having_lowest_ratio;
}
```

This method will make sure of the customers that should be served by External Fleet.

vii. Store the remaining nodes in a data structure to keep track of nodes that should be served by Internal Fleet.

viii. Keep a Copy of distance matrix and increase all the rows and columns of the nodes visited by external fleet by a high positive number.

ix. Now, Find the nodes whose distance are shortest from each other and store that node in Route and keep storing until remaining capacity of that truck gets exhausted.

x. Continue this process until all the Trucks are served.

Method of finding the initial routes:

Then check if there are any customers left unserved. Serve such unserved customers by External Fleet.

```
while (v < vehicle) // Outer loop will run for no of Truck Present
{
    float totaldistance = 0;
    IF(all truck Capacity are equal)
    THEN
    {
        IF(all variable cost of Truck are same)
        THEN
        {
            RemainingCapacity = Truck[v].capacity;
            routeIndex = v;
        }
        ELSE
        {
                                                RemainingCapacity       =
Truck[route_having_lowest_VariableCost].capacity;
            routeIndex = route_having_lowest_VariableCost;
        }
    }
    ELSE
    {
        remcapacity = Truck[route_having_lowest_TruckCapacity].capacity;
        routeIndex = route_having_lowest_TruckCapacity;
    }
   // this loop will continue till remaining capacity remains > 0

    while (remcapacity > 0)
    {
        for (int j = 0; j < customer + 1; j++)
        {   // storing all distances from a particular node
```

```
            checkingarr.push_back(distanceCopy[i][j]);
        }


        // storing the index(i.e node) having shortest(or min) distance
        among all other except those already got served

        min = min_func(checkingarr, visited);
        checkingarr.clear();
         int minDistance = *min_element(demandOfUnservedCustomer.begin(),
demandOfUnservedCustomer.end());
           // Breaking condition 1 : // storing the minimum distance found for
           visiting a node is greater than the remaining capacity of Truck
        if (remcapacity < minDistance)
        {
            break;
        }
        // Breaking condition 2 : If demand of the node having shortest
        distance (if visited) is greater than the remaining capacity of
        Truck
        if (customerdata[min][3] > remcapacity)
        {
            break;
        }
        route.push_back(min);   // storing the nodes if break conditions are
                                not satisfied
         // deleting the details of the node that's been inserted already
                    unservedCustomer.erase(remove(unservedCustomer.begin(),
unservedCustomer.end(), float(min)), unservedCustomer.end());
        demandOfUnservedCustomer.erase(remove(demandOfUnservedCustomer.beg
in(),        demandOfUnservedCustomer.end(),        customerdata[min][3]),
demandOfUnservedCustomer.end());
         //updating the remaining capacity after node's been inserted.
        remcapacity -= customerdata[min][3];
        i = min; // i now pointing to the node that has been inserted.
        //updating the visited array after a node is inserted
        visited.insert(visited.end(), route.begin(), route.end());
    }
    // end of inner while, which will give one route
    route.push_back(0);// adding 0 to the end as it's the depot
    s1.vehicleroute[routeIndex] = (route);    // storing the route for the
    route.clear();                            particular truck
    route.push_back(0);
    //storing remaining capacity of each Truck for future in the solution
```

```
    s1.remcapacityofvehicle.push_back(remcapacity);
    v++;
    i = 0; // i is now pointing to 0 since for next iteration the route
}          will again search for node which is nearest to depot(i.e, 0)
```

## Modified Two-Opt Algorithm

In optimization, 2-opt is a simple local search algorithm. The main idea behind it is to take a route that crosses over itself and reorder it so that it does not.

A complete 2-opt local search will compare every possible valid combination of the swapping mechanism. This technique can be applied to the **vehicle routing problem (VRP)** as well as the capacitated VRP(VRPPC), which require minor modification of the algorithm.

The mechanism by which we have applied modified 2-opt swapping is described below;

**PSEUDOCODE :**

```
for (int k = 0; k < vehicleNo; k++)
{
    if (vehicleNo == 2)
    {
        break;
    }
    // creating a 2D distance matrix with mean coordinates of each
    route

    meandistMatrix = find_meandistmatrix(customerdata, route, vehicle);
    // storing the column number having minimum distance from above
    distance matrix
    min_meandist = smallestInRow(meandistMatrix);
    int l = min_meandist[k];
    // One loop will run till no of nodes in route no 'k'and another will
    run till  no of nodes in route no 'l'
    for (int i = 1; i < vehicleRoute[k].size() - 1; i++)
    {
        for (int j = 1; j < vehicleRoute[l].size() - 1; j++)
        {   //  Taking copy of Route[k] and Route[l]
            routecopy1 = vehicleRoute[k];
            routecopy2 = vehicleRoute[l];
           //  Pairwise swapping is done with a 3rd variable
```

```
            int temp = routecopy1[i];
            routecopy1[i] = routecopy2[j];
            routecopy2[j] = temp;
            //  Checking capacity of both the routes
            int capacity1 = capacityCheck(routecopy1, customerdata);
            int capacity2 = capacityCheck(routecopy2, customerdata);

            //  Checking distance of both the routes

            float distance1 = checkDistance(routecopy1, distMatrix);
            float distance2 = checkDistance(routecopy2, distMatrix);
            //  Storing the new distances found
            routeDistance[k] = distance1;
            routeDistance[l] = distance2;
            //  checking the feasibility of capacity constraint
                if (capacity1 <= truckarr[k].capacity && capacity2 <=
truckarr[l].capacity)
            {   //  checking if the sum of distances of new routes less
                than sum of old distance of the two routes and
                multiplying random value[1.01 – 1.05] so that condition
                checking is a bit relaxed
                      if (distance1 + distance2 < (routeDistance[k] +
routeDistance[l]) * randomMultiplier)

                {
                    route[k] = routecopy1;      //  If two conditions are
                    route[l] = routecopy2;      satisfied, two routes
                }                               will be modified
            }
        }
    }
}
```

## Adjacent Swapping Algorithm

In this algorithm, each pair of adjacent nodes in each route will be swapped sequentially, then it will be checked for its feasibility conditions and if better routes are obtained ,then original routes will be updated.

**PSEUDOCODE :**

```
//  Outer loop will run till no of Truck present
for (int x = 0; x < vehicle; x++)
{
    routeCopy = route[x]; //  Taking copy of each route
    //  Inner loop will run till no of Nodes/customers present in each
    route
    for (int i = 1; i < route[x].size() - 2; i++)
    {
        var = routeCopy[i];                    //  adjacent node swapping is
        routeCopy[i] = routeCopy[i + 1];       done with a 3rd variable
        routeCopy[i + 1] = var;
        //  calculating distance after swapping
        distance = distanceCheck(routeCopy, distMatrix);
        //  if new distance of the route < previous route distance , then
        update the route and also update the previous route distance
        if (distance < prevRouteDistances[x])
        {
            route[x] = routeCopy;
            prevRouteDistances[x] = distance;
        }
        //  else undo the changes in the route
        else
        {
            routeCopy = route[x];
        }
    }
    // Store the updated route distance
    newRouteDistances.push_back(prevRouteDistances[x]);
    // calculating the objective value after each route, check if new
    objective value is better than the previous objective value
      float objectiveVal = objfunc(customerdata, distMatrix, truckarr,
vehicle, prevRouteDistances, s1.ext_transportcost);
```

```
    // Allowing 1% – 5% relaxation in checking, in the hope that other
    functions might bring better results
    if (objectiveVal < objValueCopy * randomMultipier)
    {
        s1.vehicleroute = route;
    }
}
```

## One-One Swapping Algorithm:

In this algorithm, each node(customer) of a route is swapped with all other nodes(customers) present in that route only and checked for any improvement in the cost. The swapping which brings best improvement among all possible swapping is granted and hence final route is modified.

**PSEUDOCODE :**

```
//  Outer loop will run till no of Truck present
for (int x = 0; x < vehicle; x++)
{
    routeCopy = route[x]; //  Taking copy of each route
    //  Inner two loops both for single route and both will run till no
    of Nodes/customers present in the route, one loop for keeping track
    of a fixed node that will be constantly swapped and another loop for
    traversing the whole route for constant swapping except the fixed
    node.
    for (int i = 1; i < route[x].size() - 1; i++)
    {
        for (int j = 1; j < route[x].size() - 1; j++)
        {
            if (i == j)
            {
                continue;
            }
            var = routeCopy[i];          //  pair of node swapping is
            routeCopy[i] = routeCopy[j]; done with a 3rd variable
            routeCopy[j] = var;
            distance = distanceCheck(routeCopy, distMatrix);
            //  each swapping will give new distances, which needs to be
            stored
            storeDist.push_back(distance);
```

```
            routeCopy = route[x];
        }
        //  minimum distance of all the distances from swapping is found
        float min = *min_element(storeDist.begin(), storeDist.end());
        //  index where minimum distance is found, this (index+2) is the
        position of the node that is swapped
         int min_index = min_element(storeDist.begin(), storeDist.end()) -
storeDist.begin();
        //  if new distance of the route(after swapping) < previous route
        distance , then update the route and also update the previous
        route distance
        if (min < prevRouteDistances[x])
        {
            var = route[x][i];
            route[x][i] = route[x][min_index + 2];
            route[x][min_index + 2] = var;
            prevRouteDistances[x] = min;
        }
        storeDist.clear();
    }
    // Store the updated route distance
    newRouteDistances.push_back(prevRouteDistances[x]);
    // calculating the objective value after each route, check if new
    objective value is better than the previous objective value
      float  objectiveVal  =  objfunc(customerdata,  distMatrix,  truckarr,
vehicle, prevRouteDistances, s1.ext_transportcost);
    // Allowing 1% – 5% relaxation in checking, in the hope that other
    functions might bring better results
    if (objectiveVal < objValueCopy * randomMultipier)
    {
        s1.vehicleroute = route;
    }
}
```

## Single Insertion Algorithm :

In this algorithm, each node(customer) of a route will be inserted between each edge except the left and right edge of the node(customer) and checked for any improvement in the cost. The insertion which brings best improvement among all possible insertion is granted and hence final route is modified.

**PSEUDOCODE :**

```
for (int x = 0; x < vehicle; x++)  // Outer loop will run till no of
{                                  Truck present
    // Inner two loops both for single route and both will run till no
    of Nodes/customers present in the route, one loop for keeping track
    of a fixed node that will be constantly inserted and another loop
    for traversing the whole route for constant insertion in edges
    except.
    for (int i = 1; i < route[x].size() - 1; i++)
    {
        for (int j = 0; j < route[x].size() - 1; j++)
        {
            routeCopy = route[x]; // Taking copy of each route
            If (j <= i - 2)
            {
                routeCopy.insert(routeCopy.begin() + j + 1, routeCopy[i]);
                routeCopy.erase(routeCopy.begin() + (i + 1));
            }
            If (j == i - 1)
            {
                continue;          // if the current iteration is situated
            }                      in left or right edge of the selected
            if (j == i)            node, current iteration will be skipped
            {
                continue;
            }
            if (j >= i + 1)
            {
                routeCopy.insert(routeCopy.begin() + j + 1, routeCopy[i]);
                routeCopy.erase(routeCopy.begin() + (i));
            }
            routesAfterInsert.push_back(routeCopy);
            distance = distanceCheck(routeCopy, distMatrix);

            // each insertion will give new distances, which needs to be
            stored

            storeDist.push_back(distance);
        }
         // minimum distance of all the distances from insertion is
         found
```

```
        float min = *min_element(storeDist.begin(), storeDist.end());

        // index where minimum distance found is stored, for keeping
        track of the position

        int min_index = min_element(storeDist.begin(), storeDist.end()) -
storeDist.begin();

        //  if new distance of the route(after swapping) < previous
        route distance , then update the route and also update the
        previous route distance

        if (min < prevRouteDistances[x])
        {
            prevRouteDistances[x] = min;
            route[x] = routesAfterInsert[min_index];
        }
        storeDist.clear();
        routesAfterInsert.clear();
    }
    newRouteDistances.push_back(prevRouteDistances[x]);

    // calculating the objective value after each route, check if new
    objective value is better than the previous objective value

     float objectiveVal = objfunc(customerdata, distMatrix, truckarr,
vehicle, prevRouteDistances, s1.ext_transportcost);

    // Allowing 1% – 5% relaxation in checking, in the hope that other
    functions might bring better results

    if (objectiveVal < objValueCopy * randomMultipier)
    {
        s1.vehicleroute = route;
    }

}
```

## External Node Insertion Algorithm:

In this algorithm, one node(customer) which are served by external vehicle, is repeatedly inserted in all possible positions of every route and checked if there is any improvement in objective value. If any improvement is found then the routes are updated and the track of nodes(customers) served by external fleet is also updated.

**PSEUDOCODE:**

```
// outer loop will run till number of nodes that are visited by external
fleet.
for (int k = 0; k < s1.notvisited.size(); k++)
    {
        // Inner two loops will run for traversing all routes available,
        one loop for the number of routes and another for nodes present
        in the routes
        for (int i = 0; i < route.size(); i++)
        {   // Taking a copy of the original route and distance of that
            route
            routeCopy = route[i];
            pseudodistance = s1.routeDistance[i];

            for (int j = 0; j < route[i].size() - 1; j++)
            {   // inserting the external node between every sequential
                pair of nodes in each route
                routeCopy.insert(routeCopy.begin()    +    j    +    1,
s1.notvisited[k]);
                // calculating the new distance and the capacity of the
                route after inserting
                totalCapacity = checkCapacity(routeCopy, customerdata);
                distance = distanceCheck(routeCopy, distMatrix);
                // feasibility checking of the route wrt capacity of
                vehicle
                if (totalCapacity <= truckarr[i].capacity)
                {   // checking if new distance travelled is less than
                    that of previous distance of the route
                    if (distance <= pseudodistance)
                    {
```

```
                        flag = 1;
                        //keeping track of route no
                        trackOfConsideration[0] = i;
                        //keeping track of position of node
                        trackOfConsideration[1] = j + 1;
                        pseudodistance = distance;
                    }
                }
                routeCopy.erase(routeCopy.begin() + j + 1);
            }
        }
        if (flag == 1)
        {
            //since route number for which the new distance is minimum is
            recorded, external node is inserted in that route in perfect
            position
            route[trackOfConsideration[0]].insert(route[trackOfConsiderati
on[0]].begin() + trackOfConsideration[1], s1.notvisited[k]);
            // distance of updated routes are also modified
                        s1.routeDistance[trackOfConsideration[0]]    =
distanceCheck(route[trackOfConsideration[0]], distMatrix);
            // Nodes from External Vehicle which are successfully
            inserted  in Pvt. Routes, are recorded, for later functions.
            recordOfInsertedNodes.push_back(s1.notvisited[k]);
            extCost = extCost-customerdata[s1.notvisited[k]][4];

        }
        // calculating the objective value after each route, check if new
        objective value is better than the previous objective value
        float objectiveVal = objfunc(customerdata, distMatrix, truckarr,
vehicle, s1.routeDistance, extCost);
        // Allowing 1% - 5% relaxation in checking, in the hope that
        other functions might bring better results
        if(objectiveVal < objValueCopy * randomMultipier)
        {
            s1.vehicleroute = route;
        }

        flag = 0;
    }
```

## External Node Swapping Algorithm:

In this algorithm, one node(customer) which are served by external vehicle, is repeatedly swapped with all possible nodes of every route and checked if there is any improvement in objective value. If any improvement is found then the routes are updated and the track of nodes(customers) served by external fleet is also updated.

**PSEUDOCODE:**

```
// outer loop will run till number of nodes that are visited by external
fleet.
for (int k = 0; k < s1.notvisited.size(); k++)
{
    // Next loop will continue till no of vehicle/route present
    for (int i = 0; i < route.size(); i++)
    {
        // Taking a copy of the original route
        routeCopy = route[i];
        // In a loop, find the distances from each node/customer to a
        particular node visited by External fleet and store these
        distances sequentially
        for (int j = 1; j < route[i].size() - 1; j++)
        {
            storeDist.push_back(distMatrix[routeCopy[j]][s1.notvisited[k]]
);
            trackof_j.push_back(j);// keep track of the node position,
        }                           i.e value of j
        // minimum distance of all the distances stored above
        float min = *min_element(storeDist.begin(), storeDist.end());
        // index where minimum distance found is stored, for keeping
        track of the node position
         int min_index = min_element(storeDist.begin(), storeDist.end()) -
storeDist.begin();
        trackOfrecords[i][0] = (min); // keeping track of min distance
        // storing node position of the node having min distance
        trackOfrecords[i][1] = (trackof_j[min_index]);
        // keeping track of route no of the node
        trackOfrecords[i][2] = (i);
        storeDist.clear();
        trackof_j.clear();
    }
```

```
    // finding route number for which distance is least
    int final_minIndex = findMinFromMatrix(trackOfrecords);
    // Taking a copy of the original route calculated above
    routeCopy1 = route[int(trackOfrecords[final_minIndex][2])];
    routeCopy = route[int(trackOfrecords[final_minIndex][2])];
    // External node is swapped with the suitable node from a route
    routeCopy1[int(trackOfrecords[final_minIndex][1])] = s1.notvisited[k];
    // new distance and capacity is calculated, after swapping
    float dist = distanceCheck(routeCopy1, distMatrix);
    int capacity = checkCapacity(routeCopy1, customerdata);
    // feasibility checking of the route wrt capacity of vehicle
                                if              (capacity              <=
truckarr[int(trackOfrecords[final_minIndex][2])].capacity)
    {   // checking if new distance travelled is less than that of
        previous distance of the route, also allowing 1% – 5% relaxation
                                            if          (dist          <
prevRouteDistances[int(trackOfrecords[final_minIndex][2])]              *
randomMultipier)
        {
            // Finally, Route, record of nodes visited by external fleet
           and External Transportation cost, all are modified
            route[int(trackOfrecords[final_minIndex][2])] = routeCopy1;
            newRouteDistances2d[int(trackOfrecords[final_minIndex][2])][k]
= dist;
                                                    s1.notvisited[k]       =
routeCopy[int(trackOfrecords[final_minIndex][1])];
              externalTransportationCost  =  externalTransportationCost  +
customerdata[routeCopy1[int(trackOfrecords[final_minIndex][1])]][4]      -
customerdata[s1.notvisited[k]][4];
             prevRouteDistances[int(trackOfrecords[final_minIndex][2])] =
dist;
        }
    }
}
```

## Max Link Insertion Algorithm:

In this algorithm, basically the node which takes most distance to reach from one node and to go from that node to another node. So firstly, for all nodes the sum of distance of reaching a particular node from previous node and distance of reaching next node from that node is calculated for each node for every routes, separately.

Next this algorithm will try to find max distance sum from the previously calculated distances' sum, The node with maximum sum , for a particular route will signify that this node has maximum deviation.

So now this algorithm, in a loop, will take that max-deviated node from a route and repeatedly insert in all other possible positions of remaining routes, trying to find improvement in objective value. If any improvement is found then the routes are updated.

**PSEUDOCODE:**

```
// Outer loop will run till no of Truck present
for (int i = 0; i < route.size(); i++)
    {
        // sum of distance of reaching a particular node from previous
        node and  distance of reaching next node from that node, is stored
        for each node of route
        for (int j = 0; j < route[i].size() - 2; j++)
        {
            deviation.push_back(distMatrix[route[i][j]][route[i][j + 1]] +
distMatrix[route[i][j + 1]][route[i][j + 2]]);
        }
        float      maxDeviation      =      *max_element(deviation.begin(),
deviation.end());
        int     index_maxDeviation     =     max_element(deviation.begin(),
deviation.end())  -  deviation.begin();  //  index  having  max  deviation  is
stored
        int  appNode  =  route[i][index_maxDeviation  +  1];  //  Node  having
maximum deviation is noted
        // Two loops will run to traverse all the routes except the route
which contains the node having max deviation
        for (int k = 0; k < route.size(); k++)
        {
            if (k == i)
                continue;
```

```
            routeCopy = route[k]; // keeping copy of route(where node will
be inserted and checked)
            routeCopy1 = route[i]; // keeping copy of route that contains
the node with max deviation
            float oldDistance = distanceCheck(routeCopy, distMatrix) +
distanceCheck(routeCopy1, distMatrix); // keeping sum of distance of two
routes
            for (int x = 0; x < routeCopy.size() - 1; x++)
            {
                routeCopy.insert(routeCopy.begin() + x + 1, appNode);
                routeCopy1.erase(remove(routeCopy1.begin(),
routeCopy1.end(), appNode), routeCopy1.end()); // Node is inserted in new
route
                float newDistance = distanceCheck(routeCopy, distMatrix) +
distanceCheck(routeCopy1, distMatrix); // that node which is inserted is
deleted from its original route
                int capacity = checkCapacity(routeCopy, customerdata);
                // feasibility checking of the route wrt capacity of
                vehicle
                if (truckarr[k].capacity >= capacity)
                {
                    if (newDistance <= oldDistance) // after insertion,
change in new distance is compared with old distance
                    {
                        // all routes are updated
                        route[i] = routeCopy1;
                        route[k] = routeCopy;
                        for (int m = 0; m < route.size(); m++)
                        {
                            if (m == i || m == k)
                                continue;
                            route[m] = s1.vehicleroute[m];
                        }
                    }
                }
                // changes that were made to the routes are reverted
                routeCopy1.insert(routeCopy1.begin() + index_maxDeviation
+ 1, appNode);
                routeCopy.erase(remove(routeCopy.begin(), routeCopy.end(),
appNode), routeCopy.end());
            }
        }
        // after one pass of updation of routes, distance of all routes are
calculated for finding objective value
        for (int i = 0; i < route.size(); i++)
        {
            routeDistance[i] = distanceCheck(route[i], distMatrix);
        }
```

```
        // calculating the objective value after each modification of
        route, check if new objective value is better than the previous
        objective value
        float objectiveVal = objfunc(customerdata, distMatrix, truckarr,
vehicle, routeDistance, s1.ext_transportcost);
        // Allowing 1% – 5% relaxation in checking, in the hope that
        other functions might bring better results
        if (objectiveVal < objValueCopy * randomMultipier)
        {
            s1.vehicleroute = route;
        }
        else
        {
            route = s1.vehicleroute;
        }
        deviation.clear();
        flag = 0;
    }
```

## Function Calling Procedure

```
solution s = initialSolution();
bestFoundSolution = s;
while (count <= 50) // if 50 times objective value is not improved, loop
will terminate
{
    itr++; // this iterator will count total no of times loop ran
    declare randomMultiplier(ranging in[1.01 - 1.05]);
    // each time time objective value is updated, count1 is incremented
    // if count1 gets more than 20 times, won't allow any more
randomMultiplier
    if (count1 > 20)
    {
        randMultiplier = 1;
    }
    flag = false;
    s = twoOpt(s); // calling Modified Two Opt function
    if (s.objectiveValue < bestFoundSolution.objectiveValue) // if new
solution gets better than that of Best found solution, later gets updated
and flag is made true
    {
        flag = true;
        bestFoundSolution = s;
        count1++
    }
    s = adjacentSwapping(s); // calling Adjacent Swapping function
    if (s.objectiveValue < bestFoundSolution.objectiveValue) // if new
solution gets better than that of Best found solution, later gets updated
and flag is made true
    {
```

```
        flag = true;
        bestFoundSolution = s;
        count1++
    }
    s = one_oneSwapping(s); // calling One-One Swapping function
    if (s.objectiveValue < bestFoundSolution.objectiveValue) // if new
solution gets better than that of Best found solution, later gets updated
and flag is made true
    {
        flag = true;
        bestFoundSolution = s;
        count1++
    }
    s = singleInsertion(s); // calling Single Insertion function
    if (s.objectiveValue < bestFoundSolution.objectiveValue) // if new
solution gets better than that of Best found solution, later gets updated
and flag is made true
    {
        flag = true;
        bestFoundSolution = s;
        count1++
    }
    s = maxLinkInsertion(s); // calling Max Link Insertion function
    if (s.objectiveValue < bestFoundSolution.objectiveValue) // if new
solution gets better than that of Best found solution, later gets updated
and flag is made true
    {
        flag = true;
        bestFoundSolution = s;
        count1++
    }
```

```
    s = externalNodeInsertion(s); // calling External Node Insertion
function
    if (s.objectiveValue < bestFoundSolution.objectiveValue) // if new
solution gets better than that of Best found solution, later gets updated
and flag is made true
    {
        flag = true;
        bestFoundSolution = s;
        count1++
    }
    s = externalNodeSwapping(s); // calling External Node Swapping
function
    if (s.objectiveValue < bestFoundSolution.objectiveValue) // if new
solution gets better than that of Best found solution, later gets updated
and flag is made true
    {
        flag = true;
        bestFoundSolution = s;
        count1++
    }
    if (flag == false) // after all function is called, if any of the
functions doesn't improve the solution, then count is incremented once
    {
        count++;
    }
}
```

## Experiment Results:

We perform experiments on the vehicle routing problem with private fleet and common based on datasets described below. The datasets are classified into different categories. There are 14 medium sized homogeneous instances (CE instance) and also 14 medium sized heterogeneous instances (CE-H instances) and 5 small sized heterogeneous instances (B-H instances).

| Instance | No of vehicle | No of Customer |
|---|---|---|
| CE-01 | 4 | 50 |
| CE-02 | 9 | 75 |
| CE-03 | 6 | 100 |
| CE-04 | 9 | 150 |
| CE-05 | 13 | 199 |
| CE-06 | 4 | 50 |
| CE-07 | 9 | 75 |
| CE-08 | 6 | 100 |
| CE-09 | 10 | 150 |
| CE-10 | 13 | 199 |
| CE-11 | 6 | 120 |
| CE-12 | 8 | 100 |
| CE-13 | 6 | 120 |
| CE-14 | 7 | 100 |
| B–H-01 | 2 | 5 |
| B–H-02 | 2 | 10 |
| B–H-03 | 3 | 15 |

| | | |
|---|---|---|
| B–H-04 | 2 | 22 |
| B–H-05 | 3 | 29 |
| CE–H-01 | 4 | 50 |
| CE–H-02 | 9 | 75 |
| CE–H-03 | 6 | 100 |
| CE–H-04 | 9 | 150 |
| CE–H-05 | 14 | 199 |
| CE–H-06 | 4 | 50 |
| CE–H-07 | 9 | 75 |
| CE–H-08 | 6 | 100 |
| CE–H-09 | 10 | 150 |
| CE–H-10 | 13 | 199 |
| CE–H-11 | 6 | 120 |
| CE–H-12 | 8 | 100 |
| CE–H-13 | 6 | 120 |
| CE–H-14 | 7 | 100 |

**Results Obtained :**

| **Instance** | **Best Cost** | **Cost Obtained** | **Gap%** |
|---|---|---|---|
| CE-01 | 1118.47 | 1330.52 | 18.958935 |
| CE-02 | 1810.52 | 2044.16 | 12.9045799 |
| CE-03 | 1940.2 | 2272.16 | 17.1095763 |
| CE-04 | 2538.991 | 2909.11 | 14.577405 |
| CE-05 | 3124.7 | 3486.78 | 11.5876724 |

| | | | |
|---|---|---|---|
| CE-06 | 1207.4 | 1434.69 | 18.8247474 |
| CE-07 | 2006.52 | 2213.69 | 10.324841 |
| CE-08 | 2072.05 | 2408.16 | 16.2211337 |
| CE-09 | 2438.5 | 3197.71 | 31.1343039 |
| CE-10 | 3429.71 | 4398.98 | 28.26099 |
| CE-11 | 2332.8 | 2748.13 | 17.8039266 |
| CE-12 | 1953.55 | 2715.17 | 38.9864605 |
| CE-13 | 2859.4 | 3192.56 | 11.6513954 |
| CE-14 | 2213.08 | 2544.41 | 14.9714425 |
| B–H-01 | 423.5 | 426.43 | 0.6918536 |
| B–H-02 | 476.5 | 427.57 | -10.2686254 |
| B–H-03 | 777 | 1070.89 | 37.8236808 |
| B–H-04 | 1515 | 2351.42 | 55.2092409 |
| B–H-05 | 1609.5 | 1883.87 | 17.046909 |
| CE–H-01 | 1191.7 | 1366.78 | 14.691617 |
| CE–H-02 | 1753.35 | 1945.88 | 10.9806941 |
| CE–H-03 | 1861.93 | 2267.13 | 21.7623649 |
| CE–H-04 | 2492.32 | 3430.77 | 37.6536721 |
| CE–H-05 | 3126.99 | 3416.43 | 9.25618566 |
| CE–H-06 | 1169.84 | 1325.92 | 13.3419955 |
| CE–H-07 | 2034.5 | 2513.82 | 23.559597 |
| CE–H-08 | 2005.19 | 2404.28 | 19.9028521 |
| CE–H-09 | 2433.28 | 3219.6 | 32.3152288 |
| CE–H-10 | 2995.64 | 3517.87 | 17.4330026 |
| CE–H-11 | 2303.06 | 2299.99 | -0.13330091 |

| | | | |
|---|---|---|---|
| CE–H-12 | 1902.05 | 2101.27 | 10.4739623 |
| CE–H-13 | 2739.35 | 3775.32 | 37.8180955 |
| CE–H-14 | 1707.24 | 2279.56 | 33.5231133 |

## Conclusion :

Transportation assignments can have various impacts on a community's economic development objectives, such as productivity, employment, business activity, property values, and investment and tax revenues. Therefore, the economic benefit of the good assignment of the fleet and the implication of solving the VRPPC is shown by the result given in the maximum traveled costs (distance) regarding the basic sets of data of using an internal fleet or we make a choice to use an external fleet or both.

We have discussed the Vehicle Routing Problem with Private fleet and Common carrier and we have proposed a set of heuristic-based algorithm to solve this type of problem. Though the proposed algorithm does not provide the best results. The results that we obtained had an average gap of 15 - 20 %.

The results showed that the solution produced by our proposed approach was highly dependent on the choice of the initial solution.

## Reference :

[1] G. Clarke, J.W. Wright, Scheduling of vehicles from a central depot to a number of delivery points, Oper. Res. 12 (1964)
[2] The vehicle routing problem with private fleet and multiple common carriers: Solution with hybrid metaheuristic algorithm - Jalel Euchi
[3] https://link.springer.com/article/10.1007/s10100-021-00759-0
[4] https://en.wikipedia.org/wiki/Vehicle_routing_problem