MATH 603 Final Project: Optimization in Vehicle Routing Problem

Yichen Zhao, Yuetong Wu, Zhuoping Zhou

Mar 29, 2022

Abstract

We introduce the vehicle routing problems and focus on two specializations: vehicle routing problem with deliveries and pickups (VRPDP) and vehicle routing problem with time window constraints (VRPTW). We then discuss various algorithms that are popular in the routing problems: heuristic algorithm and savings algorithm. We further implement Google Developer's OR-Tools to implement these algorithms into VRPDP and VRPTW for parallel evaluation and finally give an optimal (or near-optimal) solution to each problem.

1 Introduction

Vehicle Routing Problem (VRP) has been one of the most classical problems in optimization since proposed by George Dantzig and John Ramser in 1959, in which the goal is to find the "best" routes for one or multiple vehicles visiting a set of locations. Here, "best" route is usually defined as route with the least total distance or cost. VRP can be applied into many aspects of our community, including the following two most common scenarios:

- Food delivery drivers need a route to pick up food from multiple restaurants and deliver to multiple households.
- Technicians from the maintenance department need a route to go through every maintenance order at different locations in which each location has its available time window.

The most fundamental routing problem is the Travelling Salesperson Problem (TSP), which is to simply find the shortest route for a salesperson who needs to visit customers at different locations and return to the starting point. TSP is a NP-complete problem, meaning that the complexity of finding an optimized solution increases tremendously as more destinations are added. In fact, the number of all possible routes is 362880 when there are 10 destinations while the number jumps to 2432902008176640000 for 20 destinations. Even with the

implementation of optimization techniques, the time required by running the algorithm grows exponentially with regards to the size of the problem.

In most real-life cases, VRPs have constraints. The following are some examples that reflect some common real-world circumstances:

- VRP with capacity constraints, in which vehicles have maximum capacities for the items they can carry.
- VRP with pickups and deliveries, in which vehicles pick up items at some locations and drop off each item at their corresponding drop-off location. In such case, ordering of visit matters.
- VRP with time window constraints, in which the vehicles must visit the locations in specified time intervals.
- VRP with resource constraints, such as space or personnel to load and unload vehicles at the starting point of the route.
- VRP with dropped visits, in which vehicles aren't required to visit all locations, but must pay a quantified penalty for each missed location.

This project mainly focuses on the following two scenarios: VRP with pickups and deliveries and VRP with time window constraints. In order to evaluate several popular optimization algorithms in these two scenarios, we simulate two specific problems and implement algorithms in each of them for an optimized solution. The two simulated problems, the optimization algorithms we will use, and implementation of algorithms will be discussed in the next few sections.

2 Problem Formulation

2.1 Problem 1

Problem 1 specifies a vehicle routing problem in the case of pickups and deliveries. The Philadelphia office of a food delivery company, Chowbus, receives a series of order at multiple restaurants all over the city. The coordinates of these restaurants are $x_1, x_2, ..., x_n$. For each order, the driver needs to deliver the food to its orderer's place. Denote coordinates of these orderer's places as $y_1, y_2, ..., y_n$, where each y_i orders food at restaurant x_i. Drivers all start at x_0 (depot) and will return to x_0 after their task finishes. Denote D_{ij} as the distance between locations i and j (x_i's and y_i's). Now we need to determine how many drivers are needed and assign an optimized route for each driver in terms of the total distances they travel.

2.2 Problem 2

Problem 2 specifies a vehicle routing problem with time window constraints. Technicians from the tech support department of a grand company would like to complete multiple maintenance orders at different branches of the company, where each branch has its working hours. We suppose that the time window constraints for these branches are $(x_1, y_1), (x_2, y_2), ..., (x_n, y_n)$, where x_i represents opening time and y_i represents closing time. The time window for depot (in this case, the headquarter) is (x_0, y_0) which is the time slot for technicians to start their route. For this problem, we will collect the time required to travel T_{ij} between every two nodes i and j instead of their distance. Then we need to compute an optimized route that minimizes the total travel time of this family.

3 Algorithms

In this section, we will introduce three algorithms that are commonly used in vehicle routing problems: classic heuristic algorithm, insertion heuristic algorithm and savings algorithm. Please note that for the sake of conciseness, the following discussions are under the assumption of a standard vehicle routing problem (with no constraints). The algorithms will behave slightly differently while the fundamental ideas remain the same when they are implemented in each of the two simulations mentioned in the last section.

3.1 Heuristic Algorithm

There are two versions of heuristic algorithms: classic heuristic algorithm and insertion heuristic algorithm. Before describing the algorithms, we define the relevant terminology and notations that will be used. There are n locations the vehicle needs to travel to. The travel cost between two locations i and i is $T_{i,j}$. A route is a trip from the depot to a sequence of customers and back to the depot. Here for convenience, we abuse notation and represent a route as $(0, 1, 2, ..., i, ..., n + 1)$, where 0 and $n + 1$ both refer to the depot and $1, 2, ..., n$ represents each visit.

Classic heuristic algorithm is essentially the nearest neighbor algorithm. In each iteration, it locates the vertex that is closest (or more precisely, has the lowest cost) to the current position but is not yet part of the route, and then add it into the route. Then the current position will be moved to the newly added vertex before next iteration starts. The algorithm is as follows, where $Append(j)$ is adding j to the end of current route, and $Cost(j)$ is the cost after $Append(j)$:

Algorithm 1 Nearest Neighbor

```
N = set of unassigned customers
r = empty route
while N ≠ ∅ do
  c* = ∞
  for j ∈ N do
    if Cost(j) < c* then
      j* = j
      c* = Cost(j)
    end if
  end for
  Append(j*)
  N = N − j*
end while
```

The basic insertion heuristic algorithm is a parallel insertion heuristic, where multiple routes are being built at the same time. No seed selection is required, however, for the insertion heuristic algorithm to work. When it is cheapest to insert an uninserted customer on an empty route rather than an existing route, the customer will be inserted on the new route. In each major iteration, a location is selected and the insertion cost is evaluated at every possible insertion point. Then this location is inserted into a partial route where the minimal cost is reached and the affected route is updated. The detailed algorithm is as follows, where $Insert(i, j)$ means inserting j between $i-1$ and i, and $Cost(i, j)$ is the cost after $Insert(i, j)$:

Algorithm 2 Insertion Heuristic

```
N = set of unassigned customers
R = set of partial routes; initially only contains empty route
while N ≠ ∅ do
  c* = ∞
  for j ∈ N do
    for r ∈ R do
      for (i − 1, i) ∈ r do
        if Cost(i, j) < c* then
          r* = r; i* = i; j* = j
          c* = Cost(i, j)
        end if
      end for
    end for
  end for
  Insert(i*, j*); Update(r*);
  N = N − j*
end while
```

3.2 Savings Algorithm

The savings algorithm is one of the most known heuristic for VRP. Developed by Clarke and Wright in 1964, it introduces the concept of "savings" in VRP and solves for the optimal route by maximizing the savings. The basic setup for the savings algorithm is this: suppose we have the depot located at x_0, and there are n delivery locations that we need to visit, denoted by $x_1, x_2, \cdots, x_n$. The maximum number of vehicles is n, so that the situation we starts with is that exactly one vehicle is assigned to each delivery location. The route of a vehicle is represented by an ordered list of indices of locations, where the first and last element must be 0 and other indices should appear no more than once. The cost of a route(or part of a route) is denoted by $c(route)$. If two routes $\{0, i, 0\}$ and $\{0, j, 0\}$ $(i \neq j)$ can be merged into a single route $\{0, i, j, 0\}$, then the savings of such merge is defined as $s_{ij} = c(\{i, 0\}) + c(\{0, j\}) - c(\{i, j\})$.

Given the setup, the algorithm goes like this:

Algorithm 3 Savings Algorithm

```
n = maximum number of vehicles
Make n routes: {0, i, 0} for each i ≥ 1
for i, j ∈ {1, ..., n} and i ≠ j do
    The savings s_ij = c({i, 0}) + c({0, j}) − c({i, j})
end for
while additional savings can be achieved do
    Sort all s_ij in non-increasing order and form a list L
    for s_ij ∈ L do
        Find the two routes that is associated with s_ij, denote them by r_1, r_2
        if i, j ∈ r_1 or i, j ∈ r_2 then
            continue
        else if {0, j} or {i, 0} is not found in neither r_1 nor r_2 then
            continue
        else
            merge r_1 and r_2
        end if
    end for
end while
```

4 Implementation and Programming

4.1 Pickup and Delivery Problem

In this section we will simulate a pickup and delivery problem for our food delivery company. First we set up the coordinates of restaurants (pickup locations) and

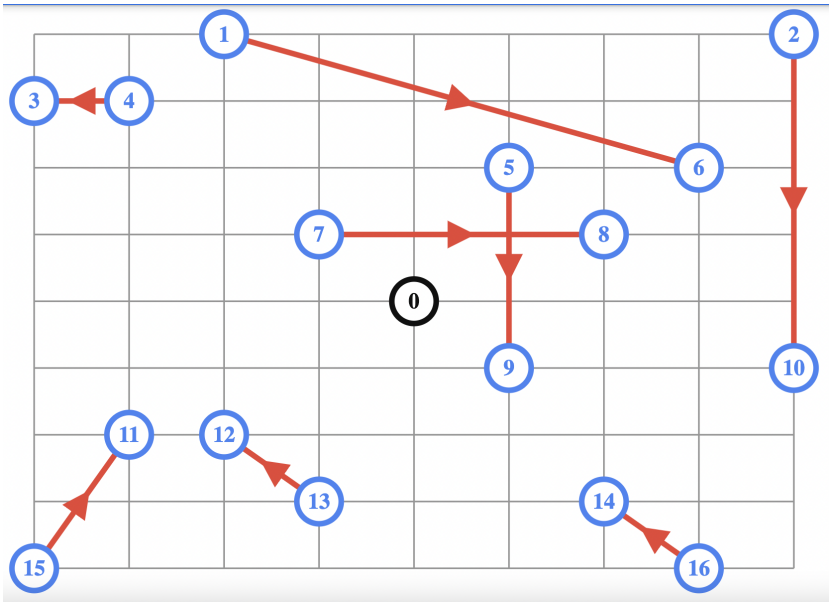


Figure 1: Setup

households (delivery locations) as shown below, where every spot is an intersection on a grid where dimension of each unit is 114 * 80.

The above diagram presents the pickup and delivery locations. Each delivery order is represented by a directed edge from its pickup location to its delivery location. For example, there is an order which is picked up at location 1 and delivered at location 6.

For the sake of a better simulation in setting up the problem, we will be using Manhattan distance to calculate distances between locations, since road structure of a city resembles a grid-like system. Manhattan distance is defined as the distance between two points (x_1, y_1) and (x_2, y_2) being $|x_1 - x_2| + |y_1 - y_2|$. With those, we now solve the problem.

4.1.1 Create the data

```
def create_data_model():
    """
    Stores the data for the problem.
    """
    data = {}
    data['distance_matrix'] = [
        [
            0, 548, 776, 696, 582, 274, 502, 194, 308, 194,
            536, 502, 388, 354, 468, 776, 662
```

```
        ],
        [
            548, 0, 684, 308, 194, 502, 730, 354, 696, 742,
            1084, 594, 480, 674, 1016, 868, 1210
        ],
        ...
        [
            662, 1210, 754, 1358, 1244, 708, 480, 856, 514, 468,
            354, 844, 730, 536, 194, 798, 0
        ],
    ]
    data['pickups_deliveries'] = [
        [1, 6],
        [2, 10],
        [4, 3],
        [5, 9],
        [7, 8],
        [15, 11],
        [13, 12],
        [16, 14]
    ]
    data['num_vehicles'] = 4
    data['depot'] = 0
    return data
```

The data consists of:

- distance_matrix: a 2-dimensional array of Manhattan distances between locations.
- pickups_deliveries: a list of pairs where first entry is index of pickup location and second entry is index of delivery location of an order.
- num_vehicles: number of vehicles (drivers) involved in the problem, will be tuned for optimization.
- depot: index of depot where all vehicles start and end their routes.

4.1.2 The routing model

We now initializes the model with the data in last section with an index manager that converts the solver's internal indices to actual indices that are assigned for locations.

```
data = create_data_model()
manager = pywrapcp.RoutingIndexManager(len(data['distance_matrix'])
    , data['num_vehicles'], data['depot'])
routing = pywrapcp.RoutingModel(manager)
```

4.1.3 Sets cost of arc

The cost of arc evaluator indicates the cost of each travel between locations and it is passed to the solver by the transit callback index. In this problem, cost of arc is a distance callback: a function that returns distance between any pair of

locations. So we create the distance callback function and then use it to define the transit callback index.

```
#creates a transit callback
def distance_callback(from_index, to_index):
    """
    Returns the manhattan distance between two nodes.
    """
    from_node = manager.IndexToNode(from_index)
    to_node = manager.IndexToNode(to_index)
    return data['distance_matrix'][from_node][to_node]

#defines cost of each arc
transit_callback_index = routing.RegisterTransitCallback(
    distance_callback)
routing.SetArcCostEvaluatorOfAllVehicles(transit_callback_index)
```

4.1.4 Add distance dimension

We adds a distance dimension to the solver in order to evaluate the routing solution provided by the solver. A dimension in the routing model is to keep track of costs (quantities) that accumulate over each route. In this problem, since distance callback is implemented for transit, the distance dimension is introduced to compute the cumulative distance traveled by each vehicle (driver) along their route.

```
dimension_name = 'Distance'
routing.AddDimension(
    transit_callback_index,
    0,
    3000, # vehicle maximum travel distance
    True, # start cumulative to 0
    dimension_name)
    distance_dimension = routing.GetDimensionOrDie(dimension_name)
    distance_dimension.SetGlobalSpanCostCoefficient(100)
```

In the method AddDimension, 3000 is passed in to set a maximum total distance of travel of each vehicle. The method SetGlobalSpanCostCoefficient sets a large coefficient 100 for the global span of routes, which makes the global span the predominant factor in generating routes. Both hyperparameters mentioned above are to minimize the length of the longest route so that drastic situations could be largely averted.

4.1.5 Add pickup and delivery constraint

The following two constraints that are specific to a pickup and delivery problem are added to our model to guarantee that the model represents a valid pickup and delivery system in real life.

- First, each delivery order must be picked up and delivered to orderer by the same vehicle.

- Second, within each delivery order, driver must reach its pickup location before delivery location.

To implement these two constraints to the routing model,

```
#defines pickup and delivery request
for request in data['pickups_deliveries']:
    #first entry being pickup, second entry being delivery
    pickup_index = manager.NodeToIndex(request[0])
    delivery_index = manager.NodeToIndex(request[1])
    routing.AddPickupAndDelivery(pickup_index, delivery_index)

    #makes sure that one vehicle serves each pair of pickup and
    delivery node
    routing.solver().Add(
        routing.VehicleVar(pickup_index) == routing.VehicleVar(
    delivery_index))

    #makes sure that the vehicle reached pickup node before
    delivery node for each pair
    routing.solver().Add(distance_dimension.CumulVar(pickup_index)
    <= distance_dimension.CumulVar(delivery_index))
```

4.1.6 Sets parameter and algorithm

As we built a standard pickup and delivery routing problem, we use default search parameters for the model and will try three common routing algorithms for parallel evaluation.

```
search_parameters = pywrapcp.DefaultRoutingSearchParameters()
search_parameters.first_solution_strategy = (
    routing_enums_pb2.FirstSolutionStrategy.
    PARALLEL_CHEAPEST_INSERTION)
```

PARALLEL_CHEAPEST_INSERTION passed to the search parameters indicates that insertion heuristic will be implemented. Other possible keywords are:

- PATH_CHEAPEST_ARC, for classic heuristic.
- SAVINGS, for savings algorithm.

4.2 Time Window Constraint Problem

In this simulation example with time window constraint, we will still be using the locations in previous example. However, since this problem involves time windows, the data should include a time matrix, which contains travel times between locations, instead of a distance matrix as in the pickup and delivery example, for our convenience.

We assume that the vehicles all travel in a constant speed of 90 unit distance per unit time, and we implements this speed to convert distance matrix to a time matrix. In addition, we defines a list of time windows for the locations, which are valid times for a visit at each location.

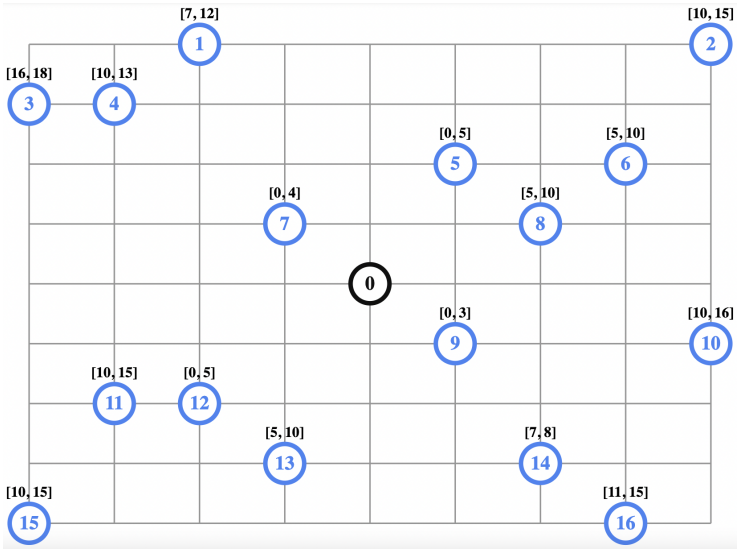


Figure 2: Setup

The diagram above shows the locations for maintenance tasks and depot at location 0. Each location bonds to a time window. Each vehicle must visit a location within its corresponding time window.

The model construction of the time window constraint problem share a similar structure to what we have done for pickup and delivery problem, along with a couple of outstanding differences due to background of the problem, which will be illustrated as follows.

4.2.1 The data

```
def create_data_model():
    """
    Stores the data for the problem.
    """
    data = {}
    data['time_matrix'] = [
        [0, 6, 9, 8, 7, 3, 6, 2, 3, 2, 6, 6, 4, 4, 5, 9, 7],
        [6, 0, 8, 3, 2, 6, 8, 4, 8, 8, 13, 7, 5, 8, 12, 10, 14],
        ...
        [7, 14, 9, 16, 14, 8, 5, 10, 6, 5, 4, 10, 8, 6, 2, 9, 0],
    ]
    data['time_windows'] = [
        (0, 5),  # depot
        (7, 12),  # 1
        (10, 15),  # 2
        (16, 18),  # 3
        (10, 13),  # 4
        (0, 5),  # 5
```

```
        (5, 10),  # 6
        (0, 4),  # 7
        (5, 10),  # 8
        (0, 3),  # 9
        (10, 16),  # 10
        (10, 15),  # 11
        (0, 5),  # 12
        (5, 10),  # 13
        (7, 8),  # 14
        (10, 15),  # 15
        (11, 15),  # 16
    ]
    data['num_vehicles'] = 4
    data['depot'] = 0
    return data
```

The data consists of:

- time_matrix: an array of travel times between locations which is converted from distance matrix in previous example given that speed of every vehicle is constant.
- time_window: a list of time slots for the locations. A vehicle can only reach a location within the time window of that location.
- num_vehicles: number of vehicles (drivers) involved in the problem, will be tuned for optimization.
- depot: index of depot where all vehicles start and end their route.

4.2.2 Time callback

In accordance with the dominance of idea of time over distance in this problem, a time callback is implemented in order for setting the cost of arc to be travel time between locations.

```
#creates a transit callback
def time_callback(from_index, to_index):
    """
    Returns travel time between two nodes.
    """
    #converts from routing variable index to time matrix NodeIndex
    from_node = manager.IndexToNode(from_index)
    to_node = manager.IndexToNode(to_index)
    return data['time_matrix'][from_node][to_node]

#defines cost of each arc
transit_callback_index = routing.RegisterTransitCallback(
    time_callback)
routing.SetArcCostEvaluatorOfAllVehicles(transit_callback_index)
```

4.2.3 Add time dimension and time window constraint

We add a time dimension to track travel time for each vehicle, similar to the dimension for travel distance in the previous example. The variable time_dimension.CumulVar(index)

in the code below stores the cumulative travel time as a vehicle reaches location of given index.

```
#adds time window constraints
time = 'Time'
routing.AddDimension(
    transit_callback_index,
    30,  # allow waiting time
    30,  # maximum time per vehicle
    False, # does not force start cumulative to 0
    time)
time_dimension = routing.GetDimensionOrDie(time)
```

The method AddDimension takes the following arguments (as in the example):

- transit_callback_index: the index for travel time callback function
- 30: an upper bound for slack, the wait time at the locations
- 30: an upper bound for total travel time over each vehicle's route
- False: the cumulative variable will not be set to 0 when vehicle returns depot (since we track time)

Next we set time window for each location and add the time constraint:

```
#adds time window constraints for each location except depot
for location_index, time_window in enumerate(data['time_windows']):
    #checks if location is depot
    if location_index == data['depot']:
        continue
    index = manager.NodeToIndex(location_index)
    time_dimension.CumulVar(index).SetRange(time_window[0],
    time_window[1])
#adds time window constraints for depot
depot_index = data['depot']
for vehicle_id in range(data['num_vehicles']):
    index = routing.Start(vehicle_id)
    time_dimension.CumulVar(index).SetRange(data['time_windows'][
    depot_index][0],
        data['time_windows'][depot_index][1])
#instantiates route start and end times to produce feasible times
for i in range(data['num_vehicles']):
    routing.AddVariableMinimizedByFinalizer(time_dimension.CumulVar
    (routing.Start(i)))
    routing.AddVariableMinimizedByFinalizer(time_dimension.CumulVar
    (routing.End(i)))
```

where the lines:

```
time_dimension.CumulVar(index).SetRange(time_window[0],
                                        time_window[1])
```

do most of the work.

5 Evaluation

In this section, we will, for each of the two problems, perform a cross-grid search with different algorithms (classic heuristic, insertion heuristic, savings) and hyperparameter tuning (number of vehicles) and evaluate all solutions with visualizations.

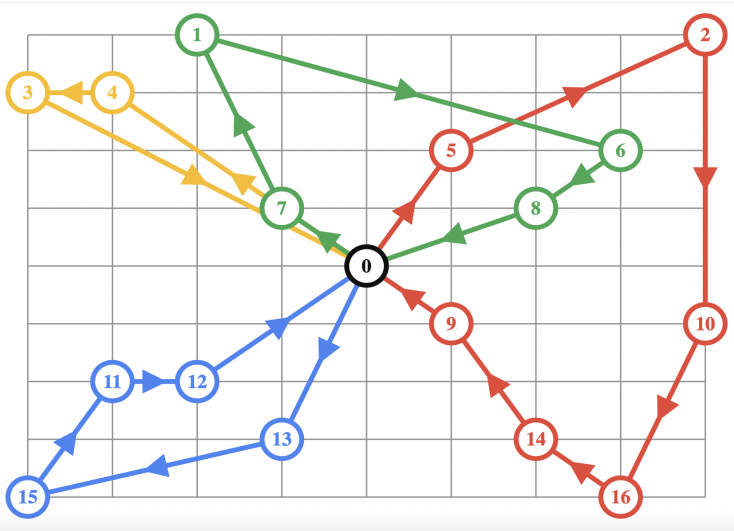
5.1 Pickup and Delivery Problem

5.1.1 An example of optimized solution

The following presents the optimized routing assignment with num_vehicle = 4 and insertion heuristic.

```
Route for vehicle 0:
 0 ->  13 ->  15 ->  11 ->  12 -> 0
Distance of the route: 1552

Route for vehicle 1:
 0 ->  5 ->  2 ->  10 ->  16 ->  14 ->  9 -> 0
Distance of the route: 2192

Route for vehicle 2:
 0 ->  4 ->  3 -> 0
Distance of the route: 1392

Route for vehicle 3:
 0 ->  7 ->  1 ->  6 ->  8 -> 0
Distance of the route: 1780

Total distance of all routes: 6916
```

The diagram below visualizes the solution:



5.1.2 Grid search and hyperparameter tuning

For searching optimization, we tune the hyperparameter num_vehicle for 3, 4, 5, 6, and 7, and apply classic heuristic, insertion heuristic and savings as algorithm. The following table records the total travel distance for each situation, where DNE means that savings algorithm does not fit in given situations:

	3	4	5	6	7
classic heuristic	6780	7304	8148	8148	8148
insertion heuristic	5980	6916	7760	7760	7760
savings	DNE	DNE	DNE	DNE	DNE

We first notice that savings algorithm is not an appropriate algorithm to use in this pickup and delivery problem.

Comparing the total travel distance between classic heuristic and insertion heuristic algorithm, we see that with no matter what value of number of vehicles, insertion heuristic has a better performance than classic heuristic (has lower travel distance). So we will focus on insertion heuristic for further observations.

In terms of number of vehicles, we realize an interesting phenomenon:

- As there are 3, 4 or 5 vehicles, total travel distance increases when there are more vehicles while average travel distance of each vehicle decreases. That is, each driver takes over a shorter route in average as number of driver increases, as shown below.

	3	4	5
insertion heuristic	1993	1729	1552

- The following shows comparison between solution of 5 vehicles and 6 vehicles, when number of vehicles reaches 6:

  Solution for 5 vehicles:

```
Route for vehicle 0:
0 ->  13 ->  15 ->  11 ->  12 -> 0
Distance of the route: 1552

Route for vehicle 1:
0 ->  16 ->  14 -> 0
Distance of the route: 1324

Route for vehicle 2:
0 ->  4 ->  3 -> 0
Distance of the route: 1392

Route for vehicle 3:
0 ->  7 ->  1 ->  6 ->  8 -> 0
Distance of the route: 1780

Route for vehicle 4:
```

```
    0 ->  5 ->  2 ->  10 ->  9 -> 0
    Distance of the route: 1712

    Total distance of all routes: 7760

```

Solution for 6 vehicles:

```
    Route for vehicle 0:
    0 ->  13 ->  15 ->  11 ->  12 -> 0
    Distance of the route: 1552

    Route for vehicle 1:
    0 ->  16 ->  14 -> 0
    Distance of the route: 1324

    Route for vehicle 2:
    0 ->  4 ->  3 -> 0
    Distance of the route: 1392

    Route for vehicle 3:
    0 ->  7 ->  1 ->  6 ->  8 -> 0
    Distance of the route: 1780

    Route for vehicle 4:
    0 ->  5 ->  2 ->  10 ->  9 -> 0
    Distance of the route: 1712

    Route for vehicle 5:
    0 -> 0
    Distance of the route: 0

    Total distance of all routes: 7760

```

The new-added vehicle 5 when there are 6 vehicles does not complete any delivery orders while the other 5 vehicles are assigned the same route as of there are 5 vehicles. This means that after putting 5 vehicles into the system, any additional vehicle in use is redundant, i.e. costing money with no work done.

Therefore, we may conclude that under the implementation of insertion heuristic algorithm, an optimized routing assignment will be produced with lowest average travel distance per vehicle when 5 vehicles are active. The solution is presented above.

5.2 Time Window Constraint Problem

5.2.1 An example of optimized solution

The following presents the optimized routing assignment with num_vehicle = 4 and classic heuristic.

```
Route for vehicle 0:
0 Time(0,0) -> 9 Time(2,3) -> 14 Time(7,8) -> 16 Time(11,11) -> 0
    Time(18,18)
Time of the route: 18

Route for vehicle 1:
0 Time(0,0) -> 7 Time(2,4) -> 1 Time(7,11) -> 4 Time(10,13) -> 3
    Time(16,16) -> 0 Time(24,24)
Time of the route: 24

Route for vehicle 2:
0 Time(0,0) -> 12 Time(4,4) -> 13 Time(6,6) -> 15 Time(11,11) -> 11
     Time(14,14) -> 0 Time(20,20)
Time of the route: 20

Route for vehicle 3:
0 Time(0,0) -> 5 Time(3,3) -> 8 Time(5,5) -> 6 Time(7,7) -> 2 Time
    (10,10) -> 10 Time(14,14) -> 0 Time(20,20)
Time of the route: 20

Total time of all routes: 82
```

For each location on a route, Time(a, b) is the solution window: the technician that visits the location should arrive and leave between a and b to stay on schedule. For example, we take a detailed look at the following portion of solution route for vehicle 1:

```
0 Time(0,0) -> 7 Time(2,4) -> 1 Time(7,11)
```

The solution time window at location 7 is (2,4), which means that the vehicle must arrive and leave between 2 and 4 at location 7. From time matrix we know that it takes technician 2 unit time to travel from depot to location 7, so to follow the solution route, the technician can stay at location 7 for at most 2 unit time before they leave location 9 and move to next location on their route.

A routing solution for time window constraint problem is given in a way that the total time of each vehicle spent on the whole route remains the same no matter how much time they stay at any location as long as they follow the schedule.

5.2.2 Grid search and hyperparameter tuning

For searching optimization, we tune the hyperparameter num_vehicle for 3, 4, 5, 6, and 7, and apply classic heuristic, insertion heuristic and savings as algorithm. The following table records the total travel time for each situation, where DNE means that savings algorithm does not fit in given situations:

	3	4	5	6	7
classic heuristic	DNE	82	82	82	82
insertion heuristic	DNE	DNE	DNE	DNE	DNE
savings	DNE	82	82	82	82

We first notice that insertion heuristic algorithm is not an appropriate to use in this time window constraint problem. Additional to that, the headquarter

should assign more than 3 technicians to complete all maintenance tasks. We also notice that classic heuristic and savings perform equally well in this problem, so they both serve as the best algorithm.

Similar to the previous pickup and delivery problem, the phenomenon mentioned in above section appear again when number of vehicle reaches 5. The following are solution with 4 vehicles and with 5 vehicles (here only shows the situation when classic heuristic is applied, same phenomenon when savings is applied):

Solution for 4 vehicles:

```
Route for vehicle 0:
0 Time(0,0) -> 9 Time(2,3) -> 14 Time(7,8) -> 16 Time(11,11) -> 0
    Time(18,18)
Time of the route: 18

Route for vehicle 1:
0 Time(0,0) -> 7 Time(2,4) -> 1 Time(7,11) -> 4 Time(10,13) -> 3
    Time(16,16) -> 0 Time(24,24)
Time of the route: 24

Route for vehicle 2:
0 Time(0,0) -> 12 Time(4,4) -> 13 Time(6,6) -> 15 Time(11,11) -> 11
     Time(14,14) -> 0 Time(20,20)
Time of the route: 20

Route for vehicle 3:
0 Time(0,0) -> 5 Time(3,3) -> 8 Time(5,5) -> 6 Time(7,7) -> 2 Time
    (10,10) -> 10 Time(14,14) -> 0 Time(20,20)
Time of the route: 20

Total time of all routes: 82
```

Solution for 5 vehicles:

```
Route for vehicle 0:
0 Time(0,0) -> 0 Time(0,0)
Time of the route: 0

Route for vehicle 1:
0 Time(0,0) -> 9 Time(2,3) -> 14 Time(7,8) -> 16 Time(11,11) -> 0
    Time(18,18)
Time of the route: 18

Route for vehicle 2:
0 Time(0,0) -> 7 Time(2,4) -> 1 Time(7,11) -> 4 Time(10,13) -> 3
    Time(16,16) -> 0 Time(24,24)
Time of the route: 24

Route for vehicle 3:
0 Time(0,0) -> 12 Time(4,4) -> 13 Time(6,6) -> 15 Time(11,11) -> 11
     Time(14,14) -> 0 Time(20,20)
Time of the route: 20

Route for vehicle 4:
0 Time(0,0) -> 5 Time(3,3) -> 8 Time(5,5) -> 6 Time(7,7) -> 2 Time
    (10,10) -> 10 Time(14,14) -> 0 Time(20,20)
```

```
Time of the route: 20

Total time of all routes: 82
```

This indicates the fact that after putting 4 technicians into the system, any additional working technician is redundant, i.e. costing money with no work done.

Therefore, we may conclude that an optimized routing assignment will be produced when 4 technicians are active under implementation of either classic heuristic or savings algorithm.

The following is the optimized solution with savings algorithm and 4 technicians active, just for readers' reference:

```
Route for vehicle 0:
0 Time(0,0) -> 7 Time(2,4) -> 1 Time(7,11) -> 4 Time(10,13) -> 3
    Time(16,16) -> 0 Time(24,24)
Time of the route: 24

Route for vehicle 1:
0 Time(0,0) -> 9 Time(2,3) -> 5 Time(4,5) -> 6 Time(6,7) -> 2 Time
    (10,10) -> 10 Time(14,14) -> 0 Time(20,20)
Time of the route: 20

Route for vehicle 2:
0 Time(0,0) -> 12 Time(4,4) -> 13 Time(6,6) -> 15 Time(11,11) -> 11
     Time(14,14) -> 0 Time(20,20)
Time of the route: 20

Route for vehicle 3:
0 Time(0,0) -> 8 Time(5,5) -> 14 Time(8,8) -> 16 Time(11,11) -> 0
    Time(18,18)
Time of the route: 18

Total time of all routes: 82
```

6 Reference

1. https://medium.com/opex-analytics/heuristic-algorithms-for-the-traveling-salesman-problem-6a53d8143584#:~:text=Heuristic%20for%20STSP%20%E2%80%94%20Nearest%20Neighbor&text=It%20works%20as%20follows%3A,the%20route%20includes%20each%20vertex.

2. https://www.academia.edu/27827242/Clarke_and_Wrights_Savings_Algorithm

3. https://www2.isye.gatech.edu/~ms79/publications/insertion-final.pdf

4. https://developers.google.com/optimization/routing/pickup_delivery

5. https://developers.google.com/optimization/routing/vrptw