SIMULATED ANNEALING FOR VEHICLE ROUTING PROBLEM WITH TIME WINDOW

BY

NATTANAN SUWANNAMANGKORN

AN INDEPENDENT STUDY SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF ENGINEERING (LOGISTICS AND SUPPLY CHAIN SYSTEMS ENGINEERING)
SIRINDHORN INTERNATIONAL INSTITUTE OF TECHNOLOGY
THAMMASAT UNIVERSITY
ACADEMIC YEAR 2025

THAMMASAT UNIVERSITY

SIRINDHORN INTERNATIONAL INSTITUTE OF TECHNOLOGY

INDEPENDENT STUDY

BY

NATTANAN SUWANNAMANGKORN

ENTITLED

SIMULATED ANNEALING FOR VEHICLE ROUTING PROBLEM WITH TIME WINDOW

was approved as partial fulfillment of the requirements for
the degree of Master of Engineering (Logistics and Supply Chain Systems Engineering)

on July 26, 2025

Member and Advisor	[illegible]
	(Assistant Professor Pham Duc Tai, Ph.D.)
Member	[illegible]
	(Associate Professor Jirachai Buddhakulsomsiri, Ph.D.)
Director	[illegible]
	(Associate Professor Kriengsak Panuwatwanich, Ph.D.)

Independent Study Title	SIMULATED ANNEALING FOR VEHICLE ROUTING PROBLEM WITH TIME WINDOW
Author	Nattanan Suwannamangkorn
Degree	Master of Engineering (Logistics and Supply Chain Systems Engineering)
Faculty/University	Sirindhorn International Institute of Technology/ Thammasat University
Advisor	Assistant Professor Pham Duc Tai, Ph.D.
Academic Years	2025

ABSTRACT

Nowadays. Vehicle routing problems (VRP) are an important component of logistics management. and is often used in transportation logistics and distribution within this article, we will look at the vehicle routing problem and present a solution using an integer linear programming model. The objective is to reduce the total load distance of transportation for each customer. For the problem, we considered the number of customers, vehicles, and the transportation distance to determine the best route for the vehicle to take from the warehouse to the customer. from the customer back to the warehouse to avoid unnecessary travel, if a shorter distance can be traveled to deliver goods, The delivery time will also be shorter, which will benefit both the customers and the transport companies. However, the use of integer linear programming is still limited. In order to resolve the issue, we developed a simulated annealing (SA) method to create delivery routes that can satisfy both requests simultaneously while reducing transportation costs and resolving a wider issue.

Keywords: Vehicle Routing Problem, Vehicle Routing Problem with Time Window, Logistics, Transportation, Simulated annealing (SA)

ACKNOWLEDGEMENTS

I want to sincerely appreciate Assistant Professor Pham Duc Tai, my advisor, for all of his or her help, support, and wise counsel during this endeavor. His support and knowledge have been essential to this work's effective conclusion.

Additionally, I would like to thank the committee members for their insightful comments and helpful recommendations, which have significantly raised the caliber of this research.

In addition, I would like to express my gratitude to all of the faculty and staff at Sirindhorn International Institute of Technology (SIIT) for offering me the resources I needed for my study as well as an engaging academic atmosphere.

Lastly, I want to express my gratitude to my friends and family for their continuous encouragement and support throughout this journey.

Nattanan Suwannamangkorn

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1
INTRODUCTION

The vehicle routing problem (VRP) is crucial to the transportation industry. because this method can calculate the route of the vehicles to deliver the goods to the customer while reducing the travel distance. When the route is shorter, the duration and cost of delivering the goods will also be reduced. Most of the problems that vehicle routing problems will be solved are, for example, routes that use more cars than necessary. But on another route, there are not enough cars to deliver products to customers. This causes unnecessary expenses, and there may be more trips than necessary. Solving problems using VRP will reduce these problems and make the delivery truck more efficient.

A vehicle routing problem must be solved by choosing the optimal route selection for the vehicles that will be used to serve a specific customer. Also, this is among the most critical and well-studied combinational optimization issues. since Dantzig and Ramser introduced the problem in 1959. They offered the first mathematical programming theory and provided real-world applications for transporting fuel to service stations. Their algorithmic approach has many practical applications in the fields of distribution, logistics, and network design problems. To achieve the objectives and solve problems such as shortest distance, lowest cost, and the shortest duration within specific constraints such as product requirements, delivery, delivery times Vehicle capacity limitations travel restrictions time constraints, etc. may be less, the number of vehicles used as little as possible (Chao & Yang,2017).

The vehicle routing problem with time windows (VRPTW) is the optimal routing of a vehicle between a warehouse and several customers that must visit within a predetermined time. which we refer to as the time window. Heuristics for vehicle routing issues fall into the following categories:

- Construction heuristics
- Improvement heuristics
- Metaheuristics

Parallel or sequential approaches known as construction heuristics are used to create early solutions to routing issues that can later be enhanced by improvement heuristics or metaheuristics. Using decision functions for the selection of the customer to be placed in the route and the insertion point inside the route, sequential algorithms construct a route for each vehicle one at a time. Using a pre-calculated estimate of the number of routes, parallel algorithms construct the routes for all vehicle in parallel (Zirour, 2008)

For the vehicle routing problem, we are provided with a set of customers with established demand, a vehicle, and a depot. The problem is reducing the overall cost of transport without limiting vehicle capacity. And there may be distance limitations. The vehicle routing problem with deliveries and pickups (VRPDP) can find a solution to the problem with depot-to-customer delivery and customer-to-depot return. The VRPDP is a significant logistical issue and has a wide range of applications (Kumar & Jayachitra,2016).

.

CHAPTER 2
REVIEW OF LITERATURE

The literature review of earlier studies that have been done on the subject of this thesis is included in this chapter. The vehicle routing problem (VRP) is first described in general terms, and then the various solution methods are discussed.

2.1 Vehicle Routing Problem with time window

The distribution modified by the vehicle routing problem reduces the total travel costs incurred by vehicles in response to customer demand. One of the topical variations is the "vehicle routing problem with a time window." that has been the subject of the most research, whose time window ensures that each customer must visit at a specific time. In the case study problems and effective solutions for VRPTW have been presented. (Taş, Jabali, & Van Woensel, 2014) They are studying a flexible time limit needed to solve the vehicle routing problem; the content is This VRPTW concept implies that time windows will be considered a strict constraint. In many real-life situations, constraints on the time window are somewhat flexible. Therefore, they evaluate the operational profit of using a fixed relaxation of the time window constraints, where the vehicle is allowed to deviate from the customer's time window as appropriate. This flexibility reduces carriers' operating costs. because customers may receive the product before the specified time. (Pan, Zhang & Lim, 2021) They study routing problems in urban transport. Considering the travel time depends on time. Concurrent loading times at the warehouse and many trips per vehicle. The objective is to reduce total distance by following a time window. vehicle capacity and travel time restrictions. This is called a "multi-trip time-dependent vehicle routing problem with time windows (MT-TDVRPTW)" and there is also an article of (Martin, Jacques and Marius, 1991) They present about developing a new algorithm for the solution. Dynamic programming was utilized to identify the shortest path while keeping time window and capacity restrictions to apply column formation to solve the LP relaxation problem of VRPTW partitioning. the addition of potential columns as necessary, and the results show that the test was successful using VRP as standard. This algorithm can solve 100 customer problems appropriately.

2.2 Vehicle Routing Problem with Delivery and Pick-up

In part, the vehicle routing problem with delivery and pick-up (VRPDP) is a very well-liked subject and is frequently utilized to solve the problem of pickup and delivery of goods to customers. There has been a lot of research done on VRPDP, such as by Gutiérrez-Jarpa et al. (2010) Talk about deliveries, selective pickups of the product, and the time window. The data consists of customer deliveries and pickup customers. Vehicles of the same capacity will be stationed at the depot and must be delivered within the time window indicated. The objective is to minimize distance costs, minus the revenue earned from pickup and delivery and another of research is (Kumar & Jayachitra,2016) Study the problem of concurrent receipt and dispatching between two warehouses with several distributed nodes and capacitated vehicle routing. Using the maximum vehicle capacity while reducing the number of vehicles used are the objectives of this study. The suggested heuristic can accomplish both goals and has been proven effective in resolving issues in various contexts.

2.3 Linear programing and Mixed integer linear programming

When multiple restrictions are applied, a linear function is maximized or minimized using the linear programming method of mathematical modeling. This method is helpful for calculating decisions in business planning. This technique is widely used in transportation businesses to achieve desired objectives such as reducing travel distances. Reduce the number of vehicles used for travel, including reducing the cost of transportation and A mixed integer linear program is an optimization of a linear function under a linear constraint when some variables have integer values. There is research that uses this technique to solve the problem (Bai, Xue, Chen & Roberts, 2015) research an integer linear programming approach, investigate a bidirectional multi-shift full truckload transportation problem. The objective is to minimize the total distance of all routes. Real-world problem with container handling at substantial container terminals is the source of this problem. (Saksuriya & Likasiri, 2022) using mixed-integer linear programming to determine a path that will allow each caregiver to start and return to the beginning location. Every patient must only pay for one visit at the lowest possible cost. Same as (Haitam, Najat & Abouchabak, 2021) Research has been done on home health care (HHC) programs as services for patients at home or for patients who are unable to go to the hospital. This project's goal is to deliver high-quality service. Minimize total costs as much as possible and limit losses. This issue is solved using mixed-integer linear programming to lower the cost of HHC projects.

CHAPTER 3
ALGORITHM

3.1 Integer linear programming for LDVRPTW

We partially present an ILP formulation for the issue and discuss revisions that could be made. To describe the LDVRPTW as a mixed integer linear programming problem, we specify the pertinent sets, parameters, and decision variables:

Sets:

$C = \{1,2,\dots,n\}$: set of customers

$N = \{0,1,2,\dots,n\}$: set of all nodes including depot which is represented by node 0

Parameters:

n: The number of customers

d_{ij}: Distance from node i to node j (km)

Z_{ij}: Cumulative time from node i to node j (hours)

U_{ij}: unloading time from node i to node j (hours)

T_{ij}: Travel time from node i to node j (hours)

q_i: The nonnegative weight (demand) of node i (tons)

Q_0: The tare of a vehicle (tons)

Q: The capacity of a vehicle (tons)

Decision Variables:

x_{ij}: binary variable, which indicates whether a vehicle travels from node i to node j or not.

y_{ij}: load weight carried by a vehicle when it travels from node i to node j (tons)

Here are the problem's constraints, listed under the headings they belong to:

$$\min \sum_{i\in N}\sum_{j\in N} d_{ij} \times y_{ij} \tag{3.1}$$

Subject to:

$$\sum_{i \in C} x_{0i} = 2 \qquad (3.2)$$

$$\sum_{i \in C} x_{i0} = 2 \qquad (3.3)$$

$$\sum_{i \in N} x_{ij} = 1 \qquad \forall j \in C \qquad (3.4)$$

$$\sum_{j \in N} x_{ij} = 1 \qquad \forall i \in C \qquad (3.5)$$

$$\sum_{j \in N, j \neq i} y_{ji} - \sum_{j \in, j \neq i} y_{ij} = q_i \qquad \forall i \in C \qquad (3.6)$$

$$y_{i0} = Q_0 \times x_{i0} \qquad \forall i \in C \qquad (3.7)$$

$$y_{ij} \leq (Q + Q_0 - q_i) \times x_{ij} \qquad \forall i, j \in N, i \ \neq j \qquad (3.8)$$

$$y_{ij} \geq \left(Q_0 + q_j\right) \times x_{ij} \qquad \forall i, j \in N, i \neq j \qquad (3.9)$$

$$x_{ij} \in \{0,1\} \qquad \forall i \in N, \forall j \in N \qquad (3.10)$$

Minimizing the total load distance is objective (3.1). Two vehicles will always be used because of restrictions (3.2) and (3.3). The degree constraints for each node are constraints (3.4) and (3.5). The classical conservation of flow equation, which balances the inflow and outflow of each node and forbids any unauthorized detours, is constraint (3.6). The cost structure of the problem requires that constraint (3.7) initialize the flow on the first arc of each route. Constraints (3.9) produce lower bounds for the flow on any arc, while constraints (3.8) handle capacity restrictions and forces y_{ij} to zero when the arc (i, j) is not on any route. Integrality constraints are given in (3.10), we do not require nonnegativity constraints (3.9). Let's refer to the restrictions (3.7), (3.8), and (3.9) as the formulation's boundary constraints.

This is constraints of the load distance vehicle routing problem, from now on we will add the constraints of the time window to constraints (3.11), (3.12), (3.13), (3.14) as follow:

Constraint (11): Subtour elimination constraint and it guarantees that the solution contains no illegal subtours.

$$\sum_{j \in N, j \neq i} Z_{ij} - \sum_{j \in N, j \neq i} Z_{ji} = \sum_{j \in N} \left(t_{ij} + u_{ij}\right) \times x_{ij} \qquad \forall i \in C \qquad (3.11)$$

Constraint (12): ensures that at least the amount of time a vehicle can take to go from point j to the depot must be subtracted from the maximum total time T_{ij}

$$Z_{ij} \leq \left(Z_{max} - t_{j0}\right) \times x_{ij} \qquad \forall i \in N, \forall i \in C \qquad (3.12)$$

Constraint (13): Ensures that the overall time spent getting to the depot does not go over the allotted amount of time.

$$Z_{ij} \leq Z_{max} \times x_{i0} \qquad \forall i \in C \qquad (3.13)$$

Constraint (14): Ensures that total time traveled up to j must be at least $T_{i0} + T_{ij}$.

$$Z_{ij} \leq \left(t_{0i} + u_i + t_{ij} + u_j\right) \times x_{ij} \qquad \forall i \in C, \forall j \in N \qquad (3.14)$$

3.2 Simulated annealing for vehicle routing problem

Simulated annealing, a local search-based heuristic, can avoid becoming stuck at a local optimum by accepting, with a tiny probability, poorer solutions as it iterates through the problem. It has been successfully used to solve numerous extremely challenging combinatorial optimization issues. The annealing procedure employed in the metallurgical industry is where the idea for this technology originated. Slow cooling is used during the annealing process. to crystallize the metal in a more aligned low energy condition. Similar to the gradual cooling step, the SA enhancement step achieves a (nearly) global bottom. The physical melting process begins with a default random solution. The algorithm selects a new solution from a predetermined area near the current answer for each iteration. This new solution's objective function value will be contrasted with that of the existing solution. evaluate the update to see if it has been updated. When a new iteration of the search is conducted, the new solution replaces the current one if the objective function value of the new one is superior, i.e., smaller in the case of miniaturization. With a very small probability offered by the Boltzmann function, $e^{-\Delta/kT}$, where k is a preset constant and T denotes temperature, fresh solutions with larger objective function values may be accepted as current solutions. Avoiding a solely solution-oriented approach currently is crucial to preserving the value of the objective function. but allow adjustments that improve the value of the objective function as well.

In this process, the temperature is set to a high initial temperature at the start of the search, which makes it simpler to accept a subpar result. A local search is carried out by the algorithm until a predetermined number of iterations has been reached. The local search then resumes after the temperature is cooled down by a certain rate. The algorithm continues to run until the temperature drops below the target temperature, at which point it stops, and the best answer is presented.

Simulated annealing is one of the techniques we use to solve routing problems. The code for Simulated Annealing through Google Colab, which consists of a total of 11 steps for computing the route, is written as follows:

Step 1: Google Colab import of routing data from Excel.

```
import pandas as pd
distance_df = pd.read_excel('/content/Data_Distance.xlsx',
sheet_name='18_cust',index_col=0)

demand_df = pd.read_excel('/content/Data_Distance.xlsx',
sheet_name='18_cust_de', usecols=['NODE','DEMAND'])

num_vehicle = int(pd.read_excel('/content/Data_Distance.xlsx',
sheet_name='demand', usecols=['number_of_vehicle']).values[0])

max_cap = int(pd.read_excel('/content/Data_Distance.xlsx',
sheet_name='demand', usecols=['max_capacity']).values[0])

vehicle_weight = int(pd.read_excel('/content/Data_Distance.xlsx',
sheet_name='demand', usecols=['vehicle_weight']).values[0])

max_time = float(pd.read_excel('/content/Data_Distance.xlsx',
sheet_name='demand', usecols=['max_time_vehicle']).values[0])

vehicle_velocity =
int(pd.read_excel('/content/Data_Distance.xlsx',
sheet_name='demand', usecols=['vehicle_velocity']).values[0])

loading_time = int(pd.read_excel('/content/Data_Distance.xlsx',
sheet_name='demand', usecols=['loading_time']).values[0])
```

Step 2: **Calculation of Travel Time**: `travel_time = distance_df / vehicle_velocity` calculates the travel time between different nodes based on the distance and vehicle velocity.

Decoding Function: The function `decoding_func(route)` takes a route as input (in the form of a list of nodes) and processes it to determine feasible routes for a vehicle, considering constraints like vehicle capacity and time. Here's what the function does:

- Initialize variables: `from_node`, `cum_load`, `cum_time`, `route`, `route_solution`, and `k`.
- While there are nodes in the chromosome (the input route)
- Get the first node as `to_node`.
- Determine the customer load (`cus_load`) of the current node.
- Calculate the travel time between the `from_node` and `to_node` using the `travel_time` matrix.
- Calculate the unloading time based on the customer load and loading time.
- If the current node is the starting node (0), append it to the route.
- Check if adding the current node to the route violates capacity and time constraints.
- If not, add the node to the route, update cumulative time and load, remove the node from the chromosome, and set the `from_node` as the current node.
- If yes, end the current route by appending the starting node (0) and reset variables.
- Finally, append the starting node (0) to close the last route, update cumulative time, and add the route to `route_solution`.

Return Value: The function returns a list of routes (`route_solution`), each of which represents a feasible route for a vehicle that adheres to capacity and time constraints.

```
travel_time =  distance_df / vehicle_velocity

def decoding_func (route):
  chromosome= route.copy()
  from_node = 0
  cum_load = 0
  cum_time = 0
  route = []
  route_solution = []
  k = 1

  while len(chromosome) > 0:
    to_node =  chromosome[0]

    cus_load = 
int(demand_df[demand_df['NODE']==to_node]['DEMAND'])

    travel_time_df = travel_time.loc[from_node,to_node]

    unloading_time = (loading_time/60*cus_load)

    if from_node == 0:
      route.append(0)

    if(cum_load + cus_load <= max_cap) and (cum_time + 
travel_time_df + unloading_time <= max_time - 
travel_time.loc[from_node,0]):
      route.append(to_node)

      cum_time = cum_time + unloading_time + travel_time_df
      cum_load = cum_load + cus_load
      chromosome.remove(to_node)
      from_node = to_node
    else:
      route.append(0)
      cum_time = cum_time + travel_time.loc[from_node,0]
      route_solution.append(route)

      from_node = 0
      cum_time = 0
      cum_load = 0
      k = k + 1
      route = []

  route.append(0)
  cum_time = cum_time + travel_time.loc[from_node,0]
  route_solution.append(route)

  return route_solution
```

Step 3: **Function Definition**: `def get_sum_load(route): ` defines a function named `get_sum_load` which takes a single argument `route`.

Variable Initialization: Inside the function, `route_n` is created as a copy of the input `route`. `from_node` is initialized as 0 (the starting node), and `sum_load` is initialized as 0 to keep track of the cumulative load.

Looping through the Route:

- A `while` loop is used to iterate through the nodes in the `route_n` list until there are no nodes left.
- The first node in the `route_n` list is extracted as `to_node`.
- The distance between `from_node` and `to_node` is fetched from the `distance_df` DataFrame using `.loc[from_node, to_node]`.
- `from_node` is updated to `to_node` for the next iteration.
- The node `from_node` is removed from the `route_n` list to move to the next node.
- The demand (load) associated with the current node (`to_node`) is fetched from the `demand_df` DataFrame using `.loc` and then calculated as an integer value.
- The demand of the current node is added to the `sum_load`.

Return Value: After looping through all nodes in the route, the function returns the calculated `sum_load`, which represents the total load associated with the given route.

```
def get_sum_load(route):
  route_n = route.copy()
  from_node = 0
  sum_load = 0
  while len(route_n) > 0:
    to_node = route_n[0]
    distance = distance_df.loc[from_node,to_node]
    from_node = to_node
    route_n.remove(from_node)

    cus_load = 
int(demand_df[demand_df['NODE']==to_node]['DEMAND'])

    sum_load = sum_load + cus_load

  return sum_load
```

Step 4: **Function Definition**: `def calculate_total_LD(route_n):` defines a function named `calculate_total_LD` which takes a single argument `route_n`.

Variable Initialization:

- `copy_route` is created as a copy of the input `route_n`.
- `sum_load` is calculated using the previously defined `get_sum_load` function, which computes the total load of the route.
- `cum_load` is initialized as 0 to keep track of cumulative load.
- `from_node` is initialized as 0, representing the starting node.
- `load_distance` is initialized as 0 to accumulate the load-distance value.

Looping through the Route:

- A `while` loop is used to iterate through the nodes in the `copy_route` list until there are no nodes left.
- The first node in the `copy_route` list is extracted as `to_node`.
- The distance between `from_node` and `to_node` is fetched from the `distance_df` DataFrame using `.loc[from_node, to_node]`.
- The load-distance value for the current segment of the route is calculated using the formula: `(sum_load - cum_load + vehicle_weight) * distance`.
- `cus_load` is calculated by fetching the demand (load) associated with the current node from the `demand_df` DataFrame.
- `cum_load` is updated by adding the demand of the current node.
- The current node (`to_node`) is removed from the `copy_route` list to move to the next node.
- `from_node` is updated to `to_node` for the next iteration.

Calculating Load-Distance for Last Segment:

- After the loop, the distance between the last node and the starting node (0) is fetched from the `distance_df` DataFrame.
- The load-distance value for the last segment of the route is calculated similarly to before.
- The load-distance values for all segments are accumulated to get the total load-distance value.

Return Value: The function returns the calculated `load_distance`, which represents the load-distance value associated with the given route.

```
def calculate_total_LD (route_n):
  copy_route = route_n.copy()
  sum_load = get_sum_load(route_n)
  cum_load = 0
  from_node = 0
  load_distance = 0

  while len(copy_route) > 0:
    to_node = copy_route [0]
    distance = distance_df.loc[from_node,to_node]
    load_distance = load_distance + (sum_load-cum_load +
vehicle_weight)*distance

    cus_load =
int(demand_df[demand_df['NODE']==to_node]['DEMAND'])

    cum_load = cum_load + cus_load
    copy_route.remove(to_node)
    from_node = to_node

  distance = distance_df.loc[from_node,0]

  load_distance = load_distance + (sum_load-cum_load +
vehicle_weight)*distance
  return load_distance
```

Step 5**: Function Definition**: `def total_LD_func(all_route):` defines a function named `total_LD_func` which takes a single argument `all_route`, expected to be a collection of routes.

Variable Initialization:

- `sum_load_distance` is initialized as 0. This variable will be used to accumulate the load-distance values of all routes.

Loop through Routes and Accumulate Load-Distance:

- A `for` loop is used to iterate through each route in the collection `all_route`.
- For each route, the `calculate_total_LD(route)` function is called to compute the load-distance value for that specific route.
- The calculated load-distance value for the current route is added to the `sum_load_distance` variable.

Return Value: After looping through all routes, the function returns the final value of `sum_load_distance`, which represents the cumulative load-distance value for all the provided routes.

```
def total_LD_func(all_route):
  sum_load_distance = 0

  for route in all_route:
    sum_load_distance = sum_load_distance +
calculate_total_LD(route)
  return sum_load_distance
```

Step 6: **Function Definition**: `def acceptance (current_LD, candidate_LD, T):` defines a function named `acceptance` which takes three arguments:

- `current_LD`: The load-distance value of the current solution.
- `candidate_LD`: The load-distance value of the candidate solution.
- `T`: The current temperature in the simulated annealing process.

Comparison and Decision:

- The function begins with an `if` statement to compare the load-distance value of the candidate solution (`candidate_LD`) with the load-distance value of the current solution (`current_LD`).
- If the `candidate_LD` is less than the `current_LD`, this means the candidate solution is better, so the function returns `"accept"`.

- If the `candidate_LD` is not better, it means the candidate solution is worse or equivalent. In this case, the function calculates the energy difference `E` between the candidate solution and the current solution: `E = -candidate_LD + current_LD`.
- The probability `P` of accepting the worse solution is calculated using the exponential function `math.exp(E/T)`. The higher the temperature `T`, the more likely it is to accept worse solutions.
- A random value `R` between 0 and 1 is generated using `random.random()`.
- If `P` is greater than or equal to `R`, the function returns `'accept'`. This means there's a chance to accept a worse solution based on the current temperature and the energy difference.
- If `P` is less than `R`, the function returns `'reject'`, indicating that the worse solution should be rejected.
- The code includes print statements to display the calculated probability `P` and the random value `R` for diagnostic purposes.

```
import random
import math

def acceptance(current_LD,candidate_LD, T):

  if candidate_LD < current_LD:
    return 'accept'
  else:
    E = -candidate_LD+current_LD
    P = math.exp(E/T)
    R = random.random()
    print("P =", P)
    print("R =",R)
    if P>R:
      return 'accept'
    else:
      return 'reject'
```

Step 7: **Function Definition**: `def swap(route):` defines a function named `swap` which takes a single argument `route`.

Copying the Route:

- The function starts by creating a copy of the input route called `temp_route`. This copy will be modified to generate a new candidate solution.

Randomly Selecting Indices:

- The function generates two random indices: `idx1` and `idx2`. These indices will correspond to positions in the `temp_route` list where the elements will be swapped.
- `idx1` is generated using `random.randrange(0, len(route))`. It's the index of the first element to be swapped.
- `idx2` is generated similarly, but it's important to ensure that it is different from `idx1` to avoid swapping an element with itself. A `while` loop is used to repeatedly generate `idx2` until it is different from `idx1`.

Swapping Elements:

- The elements at positions `idx1` and `idx2` in `temp_route` are swapped. This simulates the process of swapping two nodes in the route, which can lead to a new candidate solution.

Return Value:

- The function returns the modified `temp_route`, which now represents the candidate solution after the swap operation.

```
def swap(route):
    temp_route = route.copy()
    idx1=random.randrange(0,  len(route))
    idx2=random.randrange(0,  len(route))

    while idx1 == idx2:
      idx1=random.randrange(0,  len(route))
      idx2=random.randrange(0,  len(route))

    temp = temp_route [idx1]
    temp_route [idx1] = temp_route [idx2]
    temp_route [idx2] = temp
    return temp_route
```

Step 8: **Function Definition**: `def candidate_func(current_sol, T):` defines a function named `candidate_func` which takes two arguments:

- `current_sol`: The current solution represented as a list of nodes.
- `T`: The current temperature in the simulated annealing process.

Calculating `search_dist`:

- The code calculates `search_dist` as the square root of `T`, rounded to the nearest integer. This value determines the number of swap operations to be performed in this iteration.

Loop and Generating Candidate Solutions:

- A `while` loop is used to perform the swapping operation a certain number of times, specified by `search_dist`.
- A copy of the `current_sol` is created using `temp_current_sol = current_sol.copy()`. This copy will be modified to generate a candidate solution.
- The `swap(temp_current_sol)` function is called to generate a candidate solution by swapping nodes in the current solution.
- The `decoding_func(candidate_sol)` function is then called to decode the candidate solution and obtain the corresponding routes.
- If the number of routes in the decoded candidate solution is equal to the specified number of vehicles (`num_vehicle`), the candidate solution is accepted and assigned to `current_sol`. Otherwise, the `current_sol` remains unchanged.

Iteration Counter and Printing:

- The iteration counter `i` is incremented in each iteration of the loop.
- The function prints the final candidate solution after all iterations.

Return Value:

- The function returns the modified `current_sol`, which now represents the candidate solution after multiple swap operations.

```
def candidate_func (current_sol, T):

  search_dist = round(math.sqrt(T),0)

  i=1
  while i <= search_dist:
    temp_current_sol = current_sol. copy()
    candidate_sol = swap (temp_current_sol)
    decode_candidate_sol = decoding_func (candidate_sol)

    if len (decode_candidate_sol)== num_vehicle:
      current_sol = candidate_sol
    else:
      current_sol = current_sol
    i+=1
  print ('The final candidate_solution',current_sol)
  return current_sol
```

Step 9: **Function Definition**: `def SA_LDVRPTW (init_T, final_T, num_iter, max_cap, current_sol, current_LD):` defines a function named `SA_LDVRPTW` that takes several parameters:

- `init_T`: The initial temperature for the simulated annealing process.
- `final_T`: The final temperature, at which the process will stop.
- `num_iter`: The number of iterations to be performed at each temperature level.
- `max_cap`: The maximum capacity of the vehicles.
- `current_sol`: The initial solution (list of nodes) to start the algorithm.
- `current_LD`: The initial load distance associated with the `current_sol`.

Initialization and Main Loop:

- The function initializes variables `temp_cycle` and `T` (current temperature) to manage the annealing process.
- The main loop continues as long as `T` is greater than `final_T`.
- Inside the loop, the algorithm performs the specified number of iterations (`num_iter`) to explore solutions at the current temperature.

Iteration Loop:

- For each iteration.
- The current solution is decoded using `decoding_func` to obtain routes, and the current load distance is calculated using `total_LD_func`.
- A copy of the current solution (`newsol`) is created to generate a candidate solution.

- `candidate_func` is used to generate a candidate solution by performing swaps on the nodes in `newsol`.
- The decoded candidate solution's load distance is calculated.
- The `acceptance` function determines whether to accept the candidate solution based on the difference between the current load distance and the candidate load distance, as well as the current temperature.
- If the candidate solution is accepted, the current solution and load distance are updated accordingly. If the candidate's load distance is better than the best load distance seen so far (`cur_best_LD`), the best solution and load distance are also updated.
- The algorithm prints relevant information, like the acceptance status and current solution, during each iteration.

Temperature Update:

- After completing `num_iter` iterations at the current temperature, the temperature `T` is updated based on a cooling schedule. Here, it's set as `init_T - (0.95 * temp_cycle)`, where `temp_cycle` is the current temperature cycle.

Finalization:

- Once the temperature falls below `final_T`, the algorithm exits the main loop.
- The algorithm prints the final best solution and its associated load distance.
- The decoded version of the best solution is printed using `decoding_func`.
- The function returns the final best solution and its load distance.

```
def
SA_LDVRPTW(init_T,final_T,num_iter,max_cap,current_sol,current_LD
):

  temp_cycle = 0
  T = init_T
  cur_best_sol = current_sol.copy()
  cur_best_LD = current_LD.copy()
  while T > final_T:
    print('----------------Temp cycle ',temp_cycle, ",",'T =
',T,'-------------')
    for i in range(0,num_iter):
      print('........................iteration
',i+1,'........................')
      print ('The current solution is', current_sol)
      decode_current_sol = decoding_func (current_sol)
      current_LD = total_LD_func(decode_current_sol)
      print ('The current load distance is', current_LD)
      newsol = current_sol.copy()
      candidate_sol = candidate_func(newsol,T)
      print ('The current solution is', candidate_sol)
      decode_candidate_sol = decoding_func (candidate_sol)
      candidate_LD = total_LD_func(decode_candidate_sol)
      print ('The candidate load distance is', candidate_LD)
      accept = acceptance(current_LD,candidate_LD, T)
      if accept == 'accept':
        print('accept')
        current_sol = candidate_sol
        current_LD = candidate_LD
        print("Current LD after accept", current_LD)
        if current_LD < cur_best_LD:
            cur_best_sol = current_sol.copy()
            cur_best_LD = current_LD
      else:
        print('reject')
      print("current solution after compare", current_sol)
    temp_cycle = temp_cycle+1
    T = init_T-(0.95*temp_cycle)

  print("Final solution = ", cur_best_sol)
  print("Final LD =", cur_best_LD)
  print ('The final solution after decoding is', decoding_func
(cur_best_sol))
  return cur_best_sol, cur_best_LD
```

Step 10: **Function Definition**: def initial_pop(num_chromosome): This line defines a function named initial_pop that takes one argument, num_chromosome, which represents the number of chromosomes (individuals) in the population.

Initialization:

- i = 1: This initializes a variable i to 1. It's used as a counter to control the number of chromosomes generated.
- chromosome_lst = []: This initializes an empty list chromosome_lst to store the generated chromosomes.

While Loop:

- while i <= num_chromosome:: This initiates a while loop that continues until i is greater than num_chromosome. The purpose of this loop is to generate the specified number of chromosomes.

Chromosome Generation:

- chromosome = (list(np.random.permutation(len(distance_df)))): This line generates a random permutation of indices from 0 to len(distance_df) - 1. It uses NumPy's permutation function and converts the result to a list. This is a common way to represent a permutation.
- chromosome.remove(0): This removes the element 0 from the generated chromosome. This suggests that the permutation is intended to represent indices, and 0 is being excluded (possibly indicating a starting point).

List Update:

- i += 1: This increments the counter i by 1.
- chromosome_lst.append(chromosome): The generated chromosome is added to the list chromosome_lst.

Return Statement:

- return chromosome_lst: The function returns the list of generated chromosomes.

```
def initial_pop(num_chromosome):
  i = 1
  chromosome_lst =[]
  while i <= num_chromosome:
    chromosome = (list(np.random.permutation(len(distance_df))))
    chromosome.remove (0)
    i+=1
    chromosome_lst.append (chromosome)
  return chromosome_lst
```

Step 11: **Importing Libraries**: The code imports the NumPy library as `np`.

Initializing Population:

- `num_chromosome` specifies the number of initial solutions (chromosomes) in the population.
- `chromosome_lst` is a list that will hold the initial solutions generated for each chromosome using the `initial_pop` function.

Iterating Through Chromosomes:

- A loop iterates through each `initial_sol` (initial solution) in the `chromosome_lst`. This represents different initial solutions for each chromosome.

Simulated Annealing for Each Chromosome:

- For each `initial_sol`.
- The initial temperature (`init_T`), final temperature (`final_T`), number of iterations (`num_iter`), and other parameters are set.
- `current_sol` is initialized with the `initial_sol`.
- The initial solution is decoded and its load distance is computed using the `decoding_func` and `total_LD_func`.
- The `SA_LDVRPTW` function is called to perform simulated annealing using the provided parameters. This function returns the best solution and its associated load distance.
- The best solution and load distance are added to respective lists (`cur_best_sol_lst` and `cur_best_LD_lst`).

Finding the Best Solution from Population:

- After iterating through all chromosomes, the minimum load distance (`best_LD`) from the population is found using the `min` function.
- The index of `best_LD` in `cur_best_LD_lst` is used to find the corresponding best solution (`best_sol`) in `cur_best_sol_lst`.

Printing Results:

- The code then prints out various results.
- The list of best solutions found for each chromosome: `cur_best_sol_lst`.
- The list of best load distances associated with the best solutions: `cur_best_LD_lst`.

- The minimum load distance: `best_LD`.
- The best solution found: `best_sol`.
- The decoded version of the best solution using `decoding_func`: `best_decode_sol`.

```
import numpy as np

num_chromosome = 20
chromosome_lst = initial_pop(num_chromosome)
cur_best_sol_lst=[]
cur_best_LD_lst =[]
for initial_sol in chromosome_lst:
  print ('-------------start----------------------')
  print('The initial solution is',initial_sol)
  init_T = 10
  temp_cycle = 0
  final_T = 1
  num_iter = 10
  T = init_T
  current_sol = initial_sol
  decode_initial_sol = decoding_func(initial_sol)
  initial_LD = total_LD_func(decode_initial_sol)
  current_LD = initial_LD
  print('The initial load distance',initial_LD)
  cur_best_sol, cur_best_LD =
SA_LDVRPTW(init_T,final_T,num_iter,max_cap,current_sol,current_LD
)
  cur_best_sol_lst .append (cur_best_sol)
  cur_best_LD_lst.append (cur_best_LD)
  print ('-------------end----------------------')
print ('The current best solution list is', cur_best_sol_lst)
print ('The current best load distance list is', cur_best_LD_lst)
best_LD = min(cur_best_LD_lst)
print ('The minimum LD is', best_LD)
best_sol = cur_best_sol_lst[cur_best_LD_lst.index(best_LD)]
print ('The best solution is', best_sol)
best_decode_sol = decoding_func (best_sol)
print ('The best decoded solution is', best_decode_sol)
```

and here is an example of the response you will receive after running this code.

- The current best solution list is [[16, 18, 13, 6, 17, 1, 5, 2, 9, 11, 7, 10, 3, 12, 4, 15, 14, 8], [14, 6, 3, 9, 11, 15, 8, 7, 2, 16, 10, 13, 12, 4, 5, 18, 17, 1], [4, 16, 15, 9, 11, 18, 2, 7, 12, 14, 10, 5, 13, 1, 8, 6, 17, 3], [1, 16, 18, 17, 15, 4, 6, 14, 8, 11, 12, 3, 2, 13, 9, 7, 10, 5], [3, 15, 18, 13, 5, 12, 9, 17, 6, 1, 7, 2, 4, 10, 14, 16, 8, 11]]
- The current best load distance list is [3833.44, 3551.44, 3846.0800000000004, 3774.3599999999997, 4176.849999999999]
- The minimum LD is 3551.44
- The best solution is [14, 6, 3, 9, 11, 15, 8, 7, 2, 16, 10, 13, 12, 4, 5, 18, 17, 1]
- The best decoded solution is [[0, 14, 6, 3, 9, 11, 15, 8, 0], [0, 7, 2, 16, 10, 0], [0, 13, 12, 4, 5, 18, 0], [0, 17, 1, 0]]

where each execution will produce a different result.

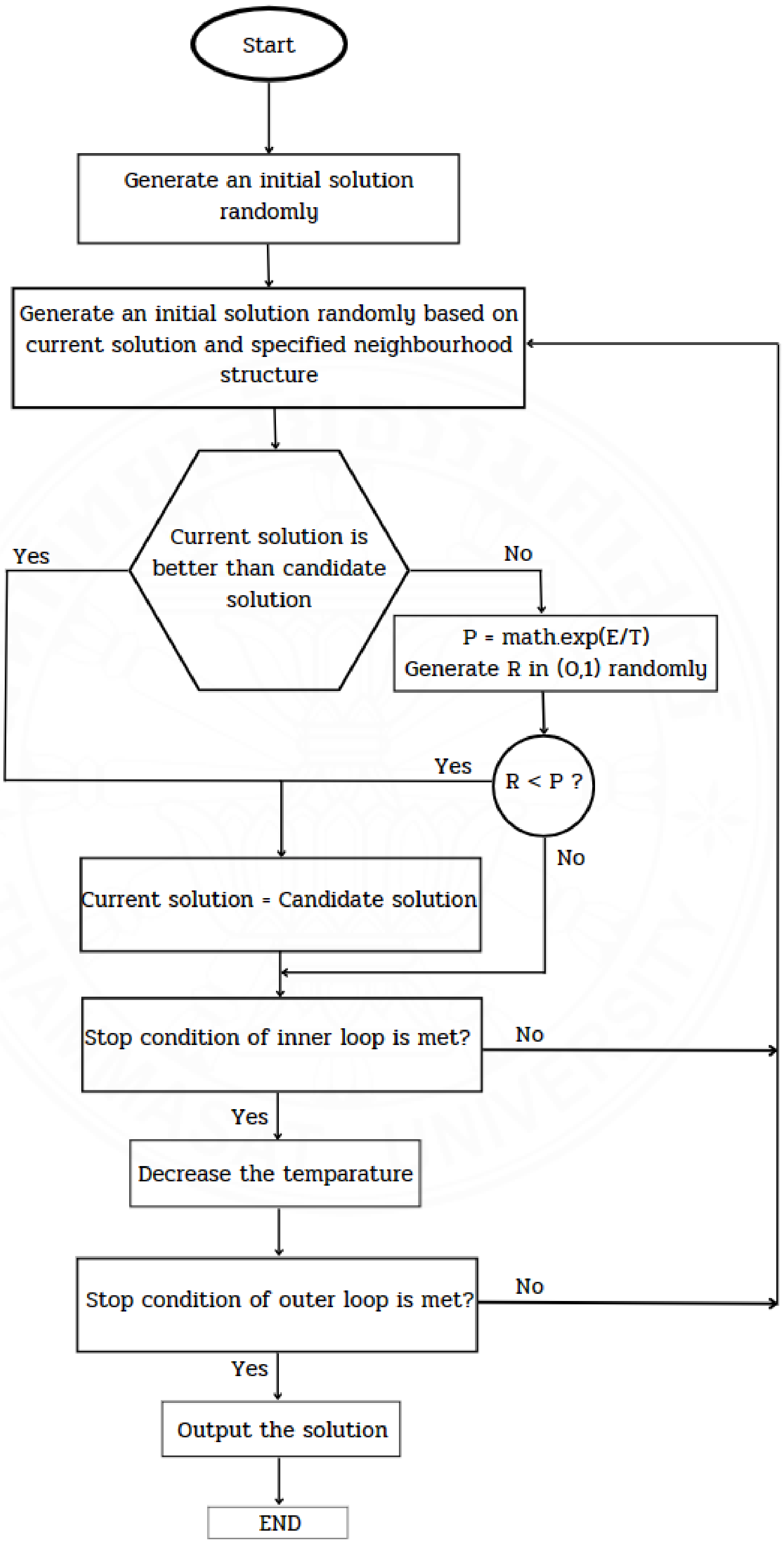
Start
Generate an initial solution randomly
Generate an initial solution randomly based on current solution and specified neighbourhood structure
Current solution is better than candidate solution
Yes
No
P = math.exp(E/T)
Generate R in (0,1) randomly
R < P ?
Yes
No
Current solution = Candidate solution
Stop condition of inner loop is met?
No
Yes
Decrease the temparature
Stop condition of outer loop is met?
No
Yes
Output the solution
END

Figure 3.1 The flow chart of simulated annealing

Figure 3.1 provides a detailed flowchart representing all steps of SA.

CHAPTER 4 RESULT AND DISCUSSION

Simulated annealing We've set a hyperparameter this time. based on the following metrics: Number of chromosomes is 20, Initial Time is 10, Final Time is 1, and Number of iterations is 10. We calculated one round of this issue in roughly 3 minutes and 40 seconds. We performed 40 computations and obtained various results. As a result, we selected the best solution to this issue is:

- **The current best solution list is** [[12, 13, 1, 7, 4, 11, 3, 16, 2, 17, 15, 18, 5, 8, 6, 14, 10, 9], [11, 8, 2, 14, 7, 1, 5, 10, 12, 3, 17, 16, 9, 18, 13, 15, 6, 4], [1, 5, 8, 9, 10, 7, 15, 6, 11, 14, 4, 17, 16, 2, 13, 3, 12, 18], [11, 12, 18, 16, 4, 13, 9, 15, 3, 14, 7, 6, 1, 10, 17, 5, 2, 8], [8, 17, 15, 11, 4, 2, 10, 18, 16, 7, 6, 14, 13, 3, 9, 5, 1, 12], [15, 12, 5, 11, 9, 10, 17, 8, 16, 2, 3, 6, 14, 1, 4, 7, 18, 13], [1, 17, 11, 6, 16, 3, 8, 7, 18, 14, 9, 4, 15, 12, 5, 10, 13, 2], [11, 15, 6, 3, 8, 16, 14, 12, 18, 4, 1, 5, 17, 2, 10, 9, 13, 7], [4, 11, 13, 14, 6, 15, 17, 2, 3, 12, 7, 8, 16, 9, 18, 10, 5, 1], [9, 12, 11, 3, 6, 1, 4, 15, 7, 2, 5, 8, 13, 10, 17, 18, 14, 16], [5, 13, 6, 9, 11, 17, 4, 14, 18, 2, 8, 10, 7, 3, 16, 12, 15, 1], [1, 14, 15, 9, 16, 17, 13, 6, 4, 12, 8, 11, 7, 10, 3, 2, 18, 5], [14, 16, 2, 13, 10, 18, 7, 15, 9, 11, 8, 17, 4, 5, 1, 3, 6, 12], [5, 6, 13, 14, 17, 1, 12, 16, 7, 8, 2, 3, 11, 15, 10, 4, 18, 9], [14, 5, 8, 10, 12, 9, 2, 16, 13, 17, 4, 3, 11, 7, 6, 15, 18, 1], [1, 3, 5, 10, 9, 4, 12, 13, 16, 2, 14, 18, 17, 11, 15, 8, 7, 6], [13, 11, 5, 3, 2, 16, 6, 12, 17, 4, 7, 18, 1, 10, 9, 14, 15, 8], [16, 13, 10, 8, 3, 12, 2, 17, 9, 14, 11, 18, 15, 7, 5, 1, 6, 4], [15, 14, 13, 17, 5, 7, 12, 1, 4, 16, 8, 3, 10, 9, 6, 11, 2, 18], [2, 11, 6, 18, 8, 10, 14, 3, 5, 16, 12, 1, 7, 15, 13, 17, 4, 9]]
- **The current best load distance list is** [3965.21, 3788.0699999999997, 4041.01, 3998.44, 3610.6499999999996, 3651.32, 4074.7499999999995, 3407.98, 3043.89, 3363.2999999999997, 3766.46, 3902.54, 3525.38, 3661.8500000000004, 3914.71, 4042.2799999999997, 3465.3500000000004, 4025.46, 3687.76, 4011.06]
- **The minimum LD is** 3043.89
- **The best solution is** [4, 11, 13, 14, 6, 15, 17, 2, 3, 12, 7, 8, 16, 9, 18, 10, 5, 1]

- **The best decoded solution is** [[0, 4, 11, 13, 14, 6, 15, 17, 0], [0, 2, 3, 12, 7, 8, 16, 0], [0, 9, 18, 10, 5, 0], [0, 1, 0]]
- **Considering the outcomes**, we chose the routes [4, 11, 13, 14, 6, 15, 17, 2, 3, 12, 7, 8, 16, 9, 18, 10, 5, 1] that have values out of a total of 20 routes. 3043.89 is the shortest load distance among the 20 paths. The best decoded answer is [[0, 4, 11, 13, 14, 6, 15, 17, 0], [0, 2, 3, 12, 7, 8, 16], [0, 9, 18, 10, 5], [0, 1, 0]], for which we used the excel solver method to get the minimum load distance for all 4 routes. It's 872.36, 1155.16, 955.09, and 388.37.

CHAPTER 5
CONCLUSIONS

Studies show that a simulated annealing technique developed in Python can solve VRP. The fact that there are just 18 customers in total places limitations on this strategy. When solving the problem, consideration was given to the number of cars, maximum capacity, vehicle weight, maximum vehicle time, vehicle velocity, and loading time. The tool we developed to solve this problem is currently unable to determine the right optimal for customers that have more than 18 customers.

In accordance with subsequent study recommendations, we should make the code more user-friendly and efficient. To evaluate how much of this result may be used in those circumstances, we should also conduct experiments in real-world scenarios.

REFERENCES

Aurachman, R., Baskara, D. B., & Habibie, J. (2021). Vehicle routing problem with simulated annealing using python programming. In *IOP Conference Series: Materials Science and Engineering* (Vol. 1010, No. 1, p. 012010). IOP Publishing.

Bai, R., Xue, N., Chen, J., & Roberts, G. W. (2015). A set-covering model for a bidirectional multi-shift full truckload vehicle routing problem. *Transportation Research Part B: Methodological*, *79*, 134-148.

Cao, W., & Yang, W. (2017). A survey of vehicle routing problems. In *MATEC Web of Conferences* (Vol. 100, p. 01006). EDP Sciences.

Desrochers, M., Desrosiers, J., & Solomon, M. (1992). A new optimization algorithm for the vehicle routing problem with time windows. *Operations research*, *40*(2), 342-354.

Gutiérrez-Jarpa, G., Desaulniers, G., Laporte, G., & Marianov, V. (2010). A branch-and-price algorithm for the vehicle routing problem with deliveries, selective pickups and time windows. *European Journal of Operational Research*, *206*(2), 341-349.

Haitam, E., Najat, R., & Abouchabaka, J. (2021). GRASP combined with ILS for the vehicle routing problem with time windows, precedence, synchronization and lunch break constraints. *International Journal of Advanced Computer Science and Applications*, *12*(5).

Kumar, V. S., & Jayachitra, R. (2016). Linear Sweep Algorithm for Vehicle Routing Problem with Simultaneous Pickup and Delivery between Two Depots With Several Nodes. *Global Journal of Pure and Applied Mathematics*, *12*(1), 897-908.

Pan, B., Zhang, Z., & Lim, A. (2021). Multi-trip time-dependent vehicle routing problem with time windows. *European Journal of Operational Research*, *291*(1), 218-231.

Redi, A. A. N. P., Maula, F. R., Kumari, F., Syaveyenda, N. U., Ruswandi, N., Khasanah, A. U., & Kurniawan, A. C. (2020). Simulated annealing algorithm for solving the capacitated vehicle routing problem: a case study of pharmaceutical distribution. *Jurnal Sistem dan Manajemen Industri*, *4*(1), 41-49.

Saksuriya, P., & Likasiri, C. (2022). Hybrid Heuristic for Vehicle Routing Problem with Time Windows and Compatibility Constraints in Home Healthcare System. *Applied Sciences*, *12*(13), 6486.

Taş, D., Jabali, O., & Van Woensel, T. (2014). A vehicle routing problem with flexible time windows. *Computers & Operations Research*, *52*, 39-54.

Zirour, M. (2008). Vehicle routing problem: models and solutions. *Journal of Quality Measurement and Analysis JQMA*, *4*(1), 205-218.