**DATABASES AND DATA STRUCTURE 1**

**MODULE CODE: COMP50004**

**DYNAMIC ROUTE ALLOCATION SYSTEM FOR “HANOI ROADWAYS”**

**Individual Coursework:** 23080330

***#Word Count:*** *548*

**Table of Contents**

[**Truck Loading Problems** 3](#_Toc185569283)

[**1.** **Problem Overview** 3](#_Toc185569284)

[**1.1** **Scenario** 3](#_Toc185569285)

[**1.2** **Major Assumption** 3](#_Toc185569286)

[**2.** **Data Structures** 3](#_Toc185569287)

[**3.** **Sample Data** 4](#_Toc185569288)

[**3.1** **Sample Input Data** 4](#_Toc185569289)

[**3.2** **Sample Output Data** 4](#_Toc185569290)

[**4.** **Algorithms** 5](#_Toc185569291)

[**5.** **Psuedo-Code** 5](#_Toc185569292)

[**5.1 Psuedo-Code for Dynamic Route Allocation** 5](#_Toc185569293)

[**5.2 Psuedo-code for backup route allocation** 7](#_Toc185569294)

[**6.** **Coding Implementation** 8](#_Toc185569295)

[**References** 9](#_Toc185569296)

[**Appendix: Original work without AI usage** 9](#_Toc185569297)

# **Truck Loading Problems**

## **Problem Overview**

### **Scenario**

Another issue has popped up after solving the packing problems with “Hanoi Roadway”. At times, drivers were unaware that some particular cities had no parcels to deliver which led to frustration feeling and directed complaints at the manager of the collector center in Hanoi because of the inability to remember specific cities that required deliveries.

### **Major Assumption**

In this scenario, the major assumption might include:

* **Parcels and Trucks:** The characteristics of this component stay maintained compared to the first scenario.
* **Stops**: Each trip of the delivery will have a maximum of 5 stops with the starting point always in Hanoi central center.
* **City List**: Only cities with parcels to deliver are included in the route plan.
* **Geographical Coordinates**: All cities have predefined geographical coordinates (latitude, longitude), which can act as a backup plan if the IT system fails.

## **Data Structures**

|  |  |
| --- | --- |
| Data Structure | Purpose |
| List | To store the optimized route of cities that need deliveries. |
| Array | To manage and process large numerical datasets such as distances between cities in a fixed-size structure for optimized computation. |
| Graph | To store distances between cities, where the cities are nodes, and distances between them are edges. |
| Dictionary | To store city data including city names and geographical coordinates which use as backup plan if the IT system fails |
| String | Store text-based information of parcels, routes, and trucks’ details |

## **Sample Data**

### **Sample Input Data**

|  |  |  |  |
| --- | --- | --- | --- |
| Parcel ID | Weight | Destination | Customer Name |
| P968 | 23 | HCMC | Hikaru |
| P554 | 45 | HCMC | Azuma |
| P726 | 62 | HCMC | Hwee |
| P269 | 34 | Dalat | Ngoc |
| P605 | 123 | Da Nang | Diep |
| P322 | 78 | Nha Trang | Vinh |
| P085 | 95 | Nha Trang | Joey |
| P529 | 92 | HCMC | Gordon |
| P749 | 5 | Hai Phong | Huy |

Table 1: Sample Customers’ Input Data

### **Sample Output Data**

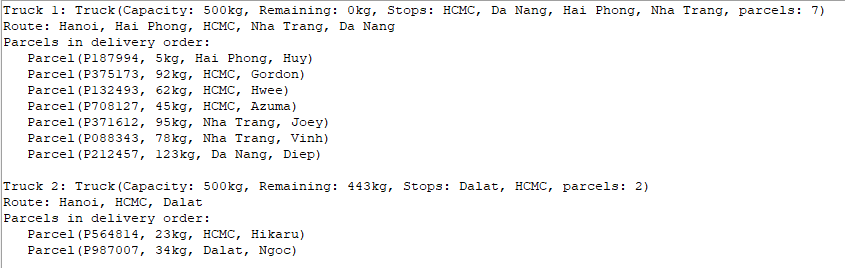


Fig.1: Dynamic route allocated for each trucks

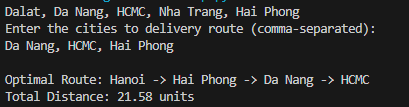
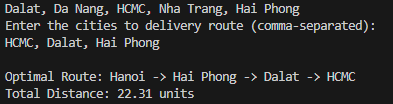
 

Fig. 2,3: Optimal route allocate of backup route allocation

## **Algorithms**

This scenario addresses the Traveling Salesman Problem (TSP) using the Held-Karp algorithm for dynamic route allocation and the Brute Force algorithm as a backup in case of IT system failures.

The **Traveling Salesman Problem** (TSP) is a classic optimization problem where the goal is to determine the shortest route that visits a set of cities exactly once and returns to the starting point. It is classified as an NP-hard problem, meaning the computational effort required grows exponentially with the number of cities, making exact solutions challenging for large datasets.

The **Held-Karp algorithm** is a dynamic programming approach for solving the TSP, where the input is the distances between a set of cities, and the objective is to find the shortest route that visits all cities exactly once and returns to the starting point. The algorithm represents subsets of visited cities using a bitmask, with each bit indicating whether a specific city is included. For each subset, it calculates and stores the minimum distance required to visit all cities in that subset and end at a specific city. Starting from a designated initial city, where the distance is initialized to zero, the algorithm iteratively builds solutions for larger subsets by evaluating the distances from visited cities to unvisited ones and updating the total costs. Once all subsets have been processed, the algorithm determines the optimal route by identifying the endpoint that minimizes the total distance, including the return to the starting city. To reconstruct the route, a parent table is used to trace back the sequence of cities based on the decisions made during computation. By using sub-problem optimization and memorization, it solves TSP in O time, which is more efficient but still exponential (Yulianto Bayu Prasetyo, 2023).

The **Brute Force algorithm**, by contrast, systematically evaluates all possible routes, calculating the total distance for each permutation and identifying the shortest route. While it guarantees optimal solutions, its factorial time complexity O(n!) makes it impractical for large datasets. However, it serves as a robust backup strategy, ensuring accurate results when the primary IT system fails to process routes in time.

## **Psuedo-Code**

### **5.1 Psuedo-Code for Dynamic Route Allocation**

Class Parcel:

Method \_\_init\_\_(parcel\_id, parcel\_weight, parcel\_destination, customer\_name):

Initialize parcel with:

- parcel\_id

- parcel\_weight

- parcel\_destination

- customer\_name

Method \_\_str\_\_():

Return string representing the parcel details

Class Trucks:

Method \_\_init\_\_(truck\_max\_capacity):

Initialize truck with:

- truck\_max\_capacity (maximum capacity)

- remaining\_capacity (same as truck\_max\_capacity)

- parcels (empty list)

- destinations (empty set)

Method truck\_accommodation(parcel):

If remaining capacity >= parcel weight:

Return True

Else:

Return False

Method add\_parcel(parcel):

Parcels.add(parcel)

Remaining\_capacity = truck\_max\_capacity – parcel\_weight

Destinations.add(parcel\_destination)

Method \_\_str\_\_():

Return string representing truck's details

Class AllocateRouteSystem:

Method \_\_init\_\_(truck\_capacity):

Initialize with:

- truck\_capacity (maximum capacity for all trucks)

- trucks (empty list)

Method packing\_parcels(parcels):

Using Best Decreasing Fit Algorithm to efficincently pack parcels for each trucks in scenario 1

Method held\_kalp\_algorithm(cities):

n = len(cities)

Initialize a DP table to store the minimum route for visiting cities

Initialize a parent table to store the path taken for reconstruction

dp[1][0] = 0 = Hanoi

for mask from 1 to (1<<n)

for u from 0 to (n):

if u not in mask:

continue

for v from 0 to (n):

if v already exist in mask:

continue

new\_mask = mask | 1<< v

distance = dp[mask][u] + distance\_between(cities[u], cities[v])

if distance < dp[new\_mask][v]

dp[new\_mask][v] = distance

parent[new\_mask][v] = u

mask = (1 << n) – 1

last\_city = city with minimum (dp[mask][i] + distance\_to\_start\_city(i, cities[0])) for i from 1 to n

min\_route = dp[mask][last\_city] + distance\_to\_start\_city(last\_city, cities[0])

best\_route = [empty list]

current\_city = last\_city

while current\_city not -1:

best\_route.append(cities[current\_city])

next\_city = parent[mask][current\_city]

mask = mask XOR (1 << current\_city)

current\_city = next\_city

reverse.best\_route()

return best\_route, min\_route

Method generate\_dynamic\_route():

for each truck in list of trucks:

if truck.destinations not None:

destinations\_for\_route = truck.destinations

destinations\_for\_route.insert(0, Hanoi)

route\_allocation = held\_karp\_algorithm(destinations\_for\_route)

else:

route\_allocation = [empty list]

Class Application:

Creating GUI class

Destination\_distance\_dict = [Distances from one city to others]

Function get\_default\_packages\_from\_excel(file):

Read file

Initialize empty list for default\_packages

For row\_info in file:

Set parcel(parcel\_id, parcel\_weight, parcel\_destination, customer\_name)

Default\_parcels.add(parcel)

Main:

Initialize Tkinter root window

Create an instance of AllocateRouteApp and start the main loop

### **5.2 Psuedo-code for backup route allocation**

city\_coordinates = [Dictionary of geographical coordinates of all cities]

Function distance\_between\_two\_cities(city1, city2)

x1,y1 = city\_coordinates[city1]

x2,y2 = city\_coordinates[city2]

return

Function route\_distances(route)

distance = 0

for in len(route)-1

distance += distances\_between\_two\_cities(route[i], route[i + 1])

distance += distances\_between\_two\_cities(route[-1], "Hanoi")

return distance

Function permutations\_generated(ele)

If len(ele) = 0 Return None

permutations = [empty]

for i in len(ele)

rest = elements[:i] + elements[i + 1:]

for each in permutations\_generated(rest):

permutations.add([elements[i]] + each)

Return permutations

Function generate\_brute\_force(starting=Hanoi)

user\_input\_route = [User inpute]

If starting in user\_input\_route

user\_input\_route.remove(starting)

All\_routes = permutations\_generated (user\_input\_route)

Set optimal\_route = None

min\_distance = float(‘inf’)

for each in all\_route:

curr\_route = [starting] + route

curr\_distance = route\_distances(curr\_route)

if curr\_distance < min\_distance

min\_distance = curr\_distance

optimal\_route = curr\_route

return optimal\_route, min\_distance

User input functions

## **Coding Implementation**

The program builds upon the source code from the first scenario with additional updates. Once parcels are packed onto the trucks, an information terminal displays all relevant details about each truck, including its optimal delivery route.

In case an IT system failure, a separate Python file allows users to manually input the list of cities for parcel delivery. This file then generates the optimal route based on the provided data.

Github link: <https://github.com/Hikaru2035/Data-Structures-Dynamic-Route-Allocation-System>

# **References**

Yulianto Bayu Prasetyo, M. A. R., 2023. Implementation of Held-Karp Algorithm to CalculateShortest Route Path. p. 2.

# **Appendix: Original work without AI usage**

**4. Algorithms**

This scenario will be solved based on the Traveling Salesman Problem with the application of the Held-Karp algorithm for the dynamic route allocation system and the Brute Force algorithm for the backup route in case the IT systems fail.

The Traveling Salesman Problem is a classic problem in computer science. It involves finding the shortest path for a salesman to visit all given cities exactly one time and return to the starting point. TSP is an NP-hard problem because the computation effort required to solve grows rapidly with the number of cities. The complexity of the large datasets that apply to the problems makes it challenging to find exact solutions for the problems.

The Held-Karp algorithm is a dynamic programming algorithm for solving the TSP which input is the distance between a set of cities and the goal is to find the minimum-length go through all cities exactly one. First, it represents subsets of visited cities as a bitmask in which each bit indicates whether a city is included or not. Next, the minimum distances to visit all cities will be tracked in each subset and ending at a specific city. By having a default starting point where the initialized distance is marked as 0 and then iteratively builds solutions for larger subsets by evaluating the distances from visited to unvisited city and updating the total distance. After processing all the subsets, the algorithm will identify the optimal route to visit all required cities and return to the first city point by considering all possible endpoints. Finally, it uses the parent table to reconstruct the sequence of the cities in the optimal route by tracing backward through the decisions made. By using sub-problem optimization and memorization, it solves TSP in O time, which is more efficient but still exponential.

A Brute Force algorithm approach in solving TSP with the beginning of listing all possible routes that include the starting city and all other destinations. Then, it calculates the total distance for each permutation, including the distance needed to return to the starting point, and finally identifies the smallest total distance. While it also generates optimal routes, its computational inefficiency makes it impractical with large datasets with a time complexity is O(n!). So this algorithm will be applied in backup strategies when the IT system fails to evaluate the optimal route in time.

**6.Code Implementation**

The program uses the source code from the first scenario with some updates. After packing parcels onto trucks, there will be an information terminal that prints out all the necessary information about the truck including the optimal route.

In case the IT system fails, there is a separate Python file that requests the user to type in all cities needed to deliver parcels to generate the optimal route.