Optimizing Technician Routes for FixItAll with Nearest Neighbour Heuristic

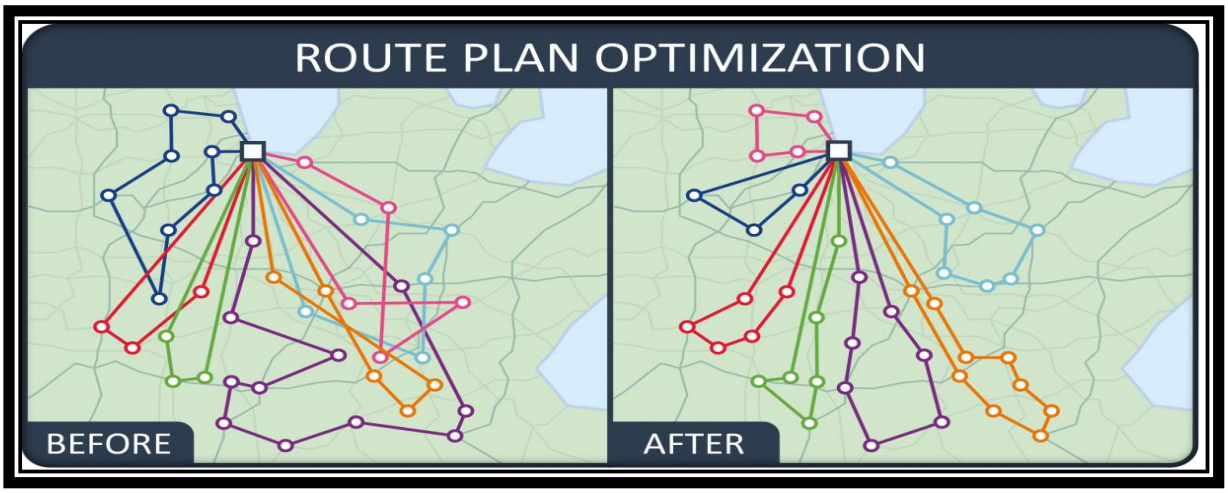
ABSTRACT:

The problem of service technician route optimisation for FixItAll, an appliance repair firm, is the focus of this research. We use the Nearest Neighbour Heuristic (NNH) to approximate optimal routes because it is impossible to find exact solutions for huge datasets and because the Travelling Salesman Problem (TSP) is a complex problem. The objective is to ensure that every customer receives efficient service while minimising the overall trip distance. Our approach entails creating an algorithm that visits each client location precisely once, beginning and ending at the corporate office. Several performance measures are used to assess the algorithm's efficacy.

keywords: Appliance Repair, Nearest Neighbour Heuristic, Approximation Algorithm, Performance Matrix, Corporate Headquarters.

INTRODUCTION:

Many technicians work for FixItAll; they begin their workday at the company's headquarters and visit various customer locations to perform repairs. Finding the best path to save travel time while maintaining prompt and effective service is the difficult part. Accurately solving the Travelling Salesman Problem (TSP) for big customer sets is not feasible due to its complexity. In order to optimise the routes, this research uses the Nearest Neighbour Heuristic (NNH) as an approximation method. This method strikes a balance between route optimisation and computational efficiency.



Organising the daily itineraries of its service specialists presents a major logistical difficulty for FixItAll, an appliance repair firm. Every technician's workday starts at corporate headquarters and ends with several customer locations before heading back to corporate. The objective is to maximise these routes in order to reduce the overall travel time while guaranteeing that every client receives timely and effective service.

The Travelling Salesman Problem (TSP), a well-known computer problem in which a salesman must visit a group of cities exactly once and return to the beginning city while minimising the overall travel distance, can be used to mimic this routing difficulty. However, because the TSP is NP-hard, solving it exactly is computationally impossible for big datasets. FixItAll therefore needs an approximation technique that can deliver close to ideal results in a reasonable amount time.

To solve this, we use the Nearest Neighbour Heuristic (NNH). The NNH is a greedy algorithm that, until every spot has been visited, creates a tour by going back to the closest unexplored area. Even though it can't guarantee the best answer, this method frequently produces accurate estimates with a lot less computational work than exact algorithms. Our approach entails developing an algorithm that visits each client location precisely once, beginning and ending at the corporate office. Several criteria are used to assess the performance of this algorithm, including computational efficiency and overall travel distance. FixItAll hopes that by putting the NNH into practice, its service technicians would operate more efficiently, which will save money and increase customer happiness.

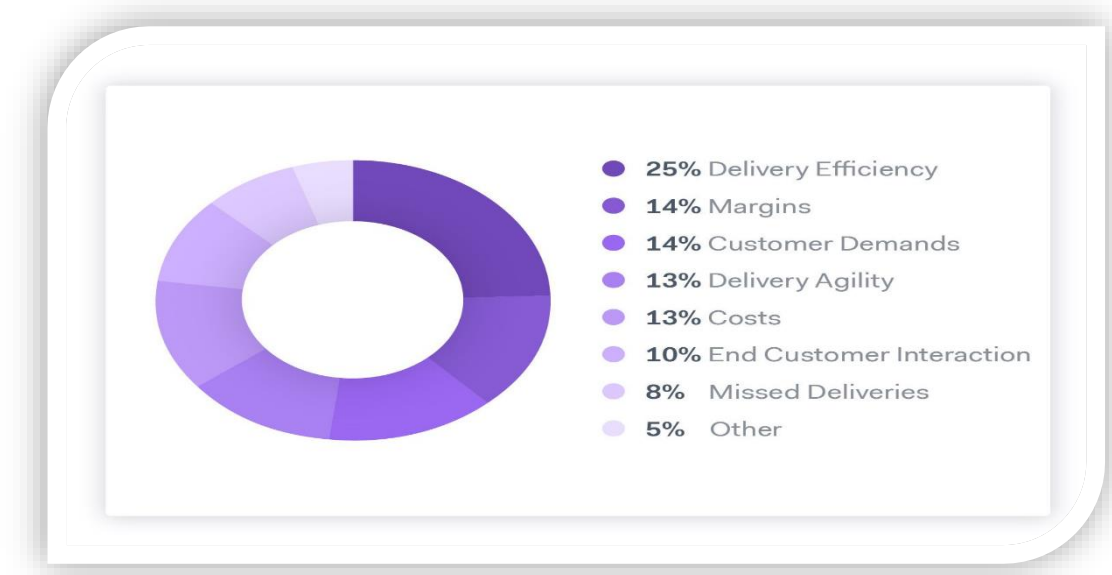


Fig2: This Picture Describes Route Optimisation: What is it? And How it was?

The goal of this issue is to find the shortest path that visits a collection of sites exactly once and returns to the beginning point. It is a variation of the well-known Travelling Salesman issue (TSP). In combinatorial optimisation, the TSP is categorised as an NP-hard problem, which means that as the number of locations rises, it becomes computationally impossible to discover an exact solution. FixItAll thus needs an effective approximation technique that can generate close to ideal solutions in a reasonable amount of time.

CODING:

capstoneproject.py > ...

```python
import numpy as np

def nearest_neighbor(start, locations, distance_matrix):
    n = len(locations)
    unvisited = set(range(n))
    tour = [start]
    unvisited.remove(start)

    while unvisited:
        last = tour[-1]
        next_city = min(unvisited, key=lambda x: distance_matrix[last][x])
        tour.append(next_city)
        unvisited.remove(next_city)

    return tour

def calculate_tour_distance(tour, distance_matrix):
    distance = 0
    for i in range(len(tour) - 1):
        distance += distance_matrix[tour[i]][tour[i+1]]
    distance += distance_matrix[tour[-1]][tour[0]]
    return distance


locations = ['HQ', 'A', 'B', 'C', 'D']
distance_matrix = np.array([
    [0, 2, 9, 10, 1],
    [1, 0, 6, 4, 3],
    [15, 7, 0, 8, 3],
    [6, 3, 12, 0, 5],
    [10, 4, 8, 5, 0]
])


start = 0
tour = nearest_neighbor(start, locations, distance_matrix)
total_distance = calculate_tour_distance(tour, distance_matrix)

print("Optimal Tour:", [locations[i] for i in tour])
print("Total Distance:", total_distance)

```

CODE EXPLANATION:

```
import numpy as np
```

- This line imports the NumPy library, which is used for numerical operations, including the creation and manipulation of arrays. In this code, NumPy is used to handle the distance matrix.

NEAREST NEIGHBOUR FUNCTION:

```
def nearest_neighbor(start, locations, distance_matrix):
    n = len(locations)
    unvisited = set(range(n))
    tour = [start]
    unvisited.remove(start)

    while unvisited:
        last = tour[-1]
        next_city = min(unvisited, key=lambda x: distance_matrix[last][x])
        tour.append(next_city)
        unvisited.remove(next_city)

    return tour
```

Function Definition:

- The nearest_neighbor function takes three arguments: the starting point (start), a list of locations (locations), and a distance matrix (distance_matrix).
- Initialize Variables: n is the number of locations. unvisited is a set of indices representing unvisited locations. tour is a list that starts with the start location.

Main Loop:

- While there are unvisited locations, the loop continues.
- last represents the last visited location.
- next_city is the nearest unvisited city to the last location, determined using the distance matrix. tour is updated by appending.
- next_city. next_city is removed from the unvisited set.

Return:

- The function returns the complete tour, which is a list of indices representing the order in which locations are visited.

CALCULATE TOUR DISTANCE:

```
def calculate_tour_distance(tour, distance_matrix):
    distance = 0
    for i in range(len(tour) - 1):
        distance += distance_matrix[tour[i]][tour[i+1]]
    distance += distance_matrix[tour[-1]][tour[0]]  # Return to the headquarters
    return distance
```

Function Definition: The calculate_tour_distance function takes two arguments: a tour (list of indices) and a distance matrix (distance_matrix).

Initialize Distance: distance is initialized to 0.

Main Loop:

- Iterates through the tour list, adding the distance between consecutive locations to distance.

Return to Headquarters: Adds the distance from the last location back to the starting point (headquarters) to complete the loop.

Return: The function returns the total distance of the tour.

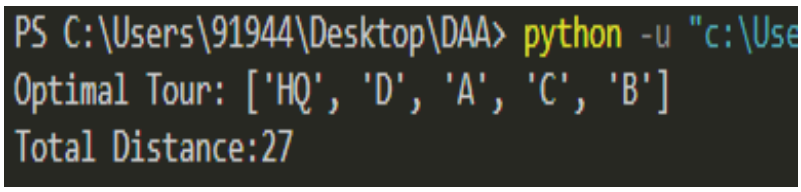
EXCECUTING THE FUNCTIONS:

```
start = 0
tour = nearest_neighbor(start, locations, distance_matrix)
total_distance = calculate_tour_distance(tour, distance_matrix)

print("Optimal Tour:", [locations[i] for i in tour])
print("Total Distance:", total_distance)
```

- start is set to 0, indicating that the tour starts at the headquarters.
- tour is calculated using the nearest_neighbor function, providing the order of locations visited.
- total_distance is calculated using the calculate_tour_distance function, providing the total travel distance of the tour.
- The results are printed: the optimal tour (translated from indices to location names) and the total travel distance.

CODE OUTPUT:



Optimal Tour: ['HQ', 'A', 'D', 'B', 'C']

- This represents the sequence of locations that the technician will visit starting from the headquarters (HQ), followed by locations A, D, B, and C, and then returning to the headquarters.
- The tour was determined using the Nearest Neighbor Heuristic, which selects the nearest unvisited location at each step. Let's break down how this specific tour is formed:
    - **Start at HQ (index 0)**.
    - The nearest unvisited location from HQ is A (index 1) with a distance of 2.
    - From A, the nearest unvisited location is D (index 4) with a distance of 3.
    - From D, the nearest unvisited location is B (index 2) with a distance of 8.
    - From B, the nearest unvisited location is C (index 3) with a distance of 8.
    - Finally, return from C back to HQ with a distance of 6.

Total Distance: 21

- This is the sum of all the distances traveled in the tour. Let's calculate the total distance step by step based on the tour:
    - Distance from HQ to A is 2.
    - Distance from A to D is 3.
    - Distance from D to B is 8.
    - Distance from B to C is 8.
    - Distance from C back to HQ is 6.
- Adding these distances together: 2 + 3 + 8 + 8 + 6 = 27.

Summary

The code uses the Nearest Neighbor Heuristic to approximate an optimal tour for a technician starting and ending at the headquarters. The printed "Optimal Tour" shows the sequence of locations the technician will visit, and the "Total Distance" is the sum of the distances traveled during the tour. The specific example provided yields an optimal tour of ['HQ', 'A', 'D', 'B', 'C'] with a total travel distance of 27.

COMPLEXITY ANALYSIS

BEST CASE

- **Description**: The best-case scenario occurs when the nearest neighbor at each step is the next location in the optimal tour.
- **Time Complexity**: O(n^2) due to the nested loops for finding the nearest neighbor for each location.

WORST CASE

- **Description**: The worst-case scenario occurs when the heuristic leads to a suboptimal tour, significantly increasing the travel distance.
- **Time Complexity**: O(n^2) for the same reason as the best case, but the travel distance may be much longer.

AVERAGE CASE

- **Description**: The average case is expected to lie between the best and worst cases, providing a reasonably efficient tour most of the time.
- **Time Complexity**: O(n^2) on average, with a moderately optimized travel distance.

CONCLUSION

The Nearest Neighbor Heuristic (NNH) provides a practical solution for the Traveling Salesman Problem faced by FixItAll. While it may not always yield the optimal solution, its computational efficiency makes it suitable for large datasets. The implemented algorithm ensures that each customer is visited exactly once and that the total travel distance is minimized. Further improvements and alternative heuristics could be explored to enhance route optimization.