**Requirement B: Core Algorithm Overview**

**Stated Problem:**

The main objective of this project is implement an algorithm and use it to solve the travelling salesman problem which is a popular problem that is categorised as a combinatorial optimization. The project is supposed to apply this concept to try to optimise a route delivery that will allow meeting all delivery deadlines while travelling the least number of miles.

For this project we will use a simple heuristic algorithm called 2-opt to develop an effective and efficient solution. The whole idea behind 2-opt is to take an initial route that crosses over itself and gradually improve it using local search until it reaches a local optimum and no more improvements can be made (improvement is done through reordering).

Therefore, our project will have an initial route which is fed to the algorithm. The algorithm analyzes the route to determine an approximate optimal route. The output is the best optimized route.

**Algorithm Overview:**

***Procedure***

1. Create an initial tour that satisfies our problem constraints
2. Perform a local search on the initial tour
3. Improve the tour by swapping the crossing points

example;

- A B - - A - B -

X ==>

- C D - - C - D -

Remove (A,D) and (B, C), then add (A,B) and (C, D). This is swapping.

To make the tour shorter, only swap the points when;

distance(A, D) + distance(B, C) > distance(A,B) + distance(C, D).

To find the best possible swap, we need to do (1 + n - 3) \* 2 checks, n being the number of points in a tour.

*Pseudocode*

***Procedure ITERATED LOCAL SEARCH***

***S0 ← GeratedInitialSolurion() # Random***

***S\* ← LocalSearch(S0)***

***L ← number of nodes eligible to be swapped***

***While termination condition not met***

***for i in range(L - 3):***

***for k in range(n+2, L - 1):***

***new\_route = 2optSwap(existing\_route, i, k)***

***new\_distance = calculateDistance(new\_route)***

***if new\_distance < best\_distance:***

***existing\_route = new\_route***

***best\_distance = new\_distance***

***goto start\_again***

***End while***

***End procedure***

***Analysis***

All our tour construction heuristic based on the procedure we have outlined above would seem to require at least O(n^2) time for an arbitrary instance, since in general every edge is a potential candidate for membership in the tour and there are n(n-1)/2 edges to be considered. The worst case of the algorithm can be O(n!) since removing edges that cross each can potentially create new crossing edges.

***Data structure***

Hash Table

Ideally, hashing is done by distributing the entries which are key/value pairs across an array of buckets. When a key is passed, the algorithm computes an index to use for finding an entry. E.g, given a table of data containing N distinct keys (k1, k2, …, kn), the search problem consists of taking a given argument k and determining whether k = ki for some value (i).

Finding the index of key;

***Index = hash(key) % array\_size***

Inserting into the data structure; **insert(key, value)**

***M ← length of array***

***Set key. key ← the search argument***

***Step A1. index ← hash\_function(key)***

***Step A2. if array[index] == key # the algorithm terminates successfully***

***Step A3. if array[index] == 0 # (0 no item) the algorithm terminates unsuccessfully***

***Step A4. index ← index - hash\_function\_2(key) # hash\_function\_2 is an incrementing hash if key***

***if index < 0***

***index = index + M***

***Return to step A2***

Getting a value from the data structure using key; **get(key)**

***Step A1. index ← hash\_function(key)***

***Step A2. for item in array:***

***if item == key***

***return item[key]***

***Running Times***

Loop up method **get(key)** guarantees O(1) worst case look up time

Insertion method **insert(key, value)** guarantees O(1) worst case time as long as the number of keys is kept below half of the capacity of the hash table, i.e., the load factor is below 50%.

Load factor = n/k

Where:

* n is the number of entries occupied in the hash table
* K is the number buckets

With this information, it’s clear that a hash table is:

* Pretty fast