**WGUPS- Delivery Routing Application**

**WGU C950 – Data Structures and Algorithms 2**

**Submitted by Anthony Munyan**

**ID 001566287**

WGUPS, a local delivery service that serves the Salt Lake City area in Utah, has procured the development of software that will perform routing duties for their daily local deliveries. The company has been struggling to meet promised delivery deadlines, and desires software that will help find more efficient and effective delivery routes. WGUPS has plans to expand into other cities and increase the size of their operations per city.

**Core Algorithm:**

The ‘nearest neighbor’ algorithm is the core algorithm for this application.

**Programming Environment:**

The software is written in Python using PyCharm Community Edition 2022.3.2.

**Application Overview:**

The application finds the shortest delivery route for cargo vehicles with multiple stops, along with the actual or expected delivery time for each stop. The software communicates to the user via command-line interface, where the user is prompted to make selections from a menu. The application returns the answer to the user’s query as text in the command-line interface.

To operate, the application parses comma-separated values (.csv files) that contain parcel, address, and distance data. Files in .csv format can be created from Microsoft Excel, which has an option to save spreadsheets as .csv files. The application uses the .csv files to create:

* A hash table that contains the parcel ID, weights, special requirements, and comments for each parcel,
* An array that contains the addresses associated with parcels,
* An array with all of the distances from each address to every other address.

The parcels are assigned to a delivery vehicle, and the route is initialized at the hub’s address. For each delivery vehicle, the routing algorithm grabs the address associated with each parcel from the address array to check the distance array for the next closest stop. The parcels are routed in the shortest route. The route distance is totalized using the distance between each stop on the route. The vehicle is assigned a time to start its route, and the expected or actual time for deliveries is based on the starting time and an average vehicle speed of 18 m.p.h. After completing the route, the vehicle returns to the hub. The distance back to the hub is added to the total route distance.

**Algorithm Overview:**

The function of the core algorithm is to arrange a delivery vehicle’s stops so that it travels the shortest distance possible. This is accomplished by utilizing the nearest-neighbor algorithm delivery\_route(). The algorithm self-adjusts based on the distances between scheduled stops and finds the shortest route. The vehicle’s address is initialized as the hub address, the starting point of the route. Then, delivery\_route() finds the next closest address and the vehicle’s address is moved there. The distance traveled to the next stop is added to the vehicle’s mileage count, the time is updated, the status of the parcel associated with the address is updated to delivered, and the algorithm loops through the parcel list to find the next closest stop. The process repeats until all parcels on the vehicle are delivered.

**Core Algorithm Pseudocode:**

delivery\_route():  
 sorting list = empty list  
 for parcels on delivery vehicle:  
 parcel list = look up parcels in hash table   
 sorting list = parcel list  
 clear parcel list  
 while sorting list is not empty:   
 next address distance = 1000  
 ‘next parcel’ = None  
 for parcel in sorting list:  
 if parcel address distance <= next address distance:  
 next stop = parcel distance  
 ‘next parcel’ = parcel associated with closest address  
 append ‘next parcel’ to parcel list  
 remove ‘next parcel’ from sorting list  
 miles = miles + nest stop  
 vehicle address = ‘next package’ address  
 time = next stop / 18 mph  
 ‘next package’ delivery time = time  
   
 back to hub = distance from last stop to hub  
 miles = miles + back to hub  
 time = time + back to hub / 18 mph

**Big O Analysis:**

The application’s space/time complexity is O(n^2). The core algorithm uses a nested loop that loops through a vehicle’s parcels to find the shortest delivery route. The individual algorithms are listed below:

|  |  |  |
| --- | --- | --- |
| Method | Space Complexity | Time Complexity |
| hashtable | O(n) | O(n) |
| build\_parcel\_file | O(n) | O(n) |
| get\_distance | O(1) | O(1) |
| delivery\_route | O(n^2) | O(n^2) |
|  |  |  |
| Overall application | O(n^2) | O(n^2) |

**Scalability:**

The application can be scaled to handle a growing number of parcels. The hash function has O(1) space complexity and can easily scaled to handle more parcels. The nearest neighbor algorithm delivery\_route does have quadratic O(n^2) space complexity, but each delivery vehicle is limited in the number of parcels it can hold. That saves delivery\_route from becoming too time or space expensive. Adding more cargo vehicles will just call the core algorithm an additional time for each vehicle added. This software is designed to be stored locally on the user’s system, so additional hub locations will not affect the operation the application, though each location will have to install and maintain the application individually.

**Efficiency:**

The hash function is very time efficient, with O(1) time efficiency. The delivery\_route algorithm (nearest neighbor) is only somewhat efficient due to having O(n^2) time complexity. The software is easy to maintain because it follows general python naming guidelines for functions, along with ample comments for each section of code.

**Algorithm Justification:**

The nearest neighbor algorithm is easy to understand and returns accurate results. It is self-adjusting according to the number of data points and the value of the data.

The delivery\_route() nearest neighbor algorithm meets the requirements ordered by WGUPS. For the test data, the total distance covered by all routes (including returning to the hub) is 127.9 miles, which is less than the maximum allowable distance of 140 miles. The algorithm includes methods to keep time, making expected delivery times, actual delivery times, and the status of parcels and delivery vehicles accessible to the user through the main menu. The algorithm is accurate, self-adjusting, and scalable (to a point). The nearest neighbor algorithm strikes a balance between performance, maintainability, and resource efficiency that serves WGUPS well.

**Alternative Algorithms:**

There are several algorithms that could be tailored to optimize delivery routes. Two alternatives to the nearest neighbor algorithm are Dijkstra’s shortest path and brute force.

Dijkstra’s shortest path differs from the nearest neighbor algorithm in that it uses vertices and weighted edges to find the shortest path. As such, Dijkstra’s shortest path requires the implementation of a graph, which can be more difficult to implement and maintain. The time and space demands for Dijkstra are similar to nearest neighbor.

Brute force, in this instance, would examine the distances between every stop to every other stop to find the overall distance of every possible route. While it would indeed find the shortest possible route, doing so would demand an incredible amount of computational resources (O(n^n)). Such a method is definitely not scalable, and for large datasets, a solution may never be determined because of the sheer number of possible routes.

**Data Structure justification:**

The hash function has several strengths which make it a good choice for this application. It can store and retrieve values based on a unique identifier, in this case the parcel ID. Particular values can also be searched and returned if the iteration through the table is implemented properly. If there are limited collisions, the hash function is efficient. The capacity of this application’s hash function is self-adjusting, which helps limits collisions. The lookup function to find key-value pairs has a linear time complexity, so the computational cost of larger data sets is not too expensive.

There are weaknesses associated with hash tables. If collisions become too numerous, the method can become costly in terms of resources. They can be difficult to implement. Additionally, it can be difficult to retrieve key-value pairs in a specific order because hash tables don’t necessarily maintain an order of its elements.

The hash function meets the requirements for this application because it accurately parses and stores parcel information and allows the parcel elements to be searched by their key. Other functions in the software that depend on the hash table can easily access and retrieve information.

This application is meant to be stored locally on the user’s system, so additional hub locations will not affect other users, though each location will have to install and maintain the application individually.

**Alternative Data Structures:**

Sets are an unordered collection of elements with no duplicate items. Each parcel could be stored as an element of the set of parcels. There are methods for adding, removing, and updating elements to a set. Sets are mutable, unlike hash tables.

Linked lists are another feasible data structure for this application. Each parcel would become one of the lists. Unlike hash tables, they are unsorted. To obtain data, the entire list must be iterated. Hash tables allow you to search for elements by key value.

**Future Releases:**

This project requires that the parcels be manually loaded onto the delivery vehicles. Future releases will include a method for automatically loading parcels. The deadline time and special considerations will be factored into the automatic loading method.