TRUCK ROUTING PROGRAM FOR THE

WESTERN GOVERNORS UNIVERSITY PARCEL SERVICE (WGUPS)

1. **ALGORITHM CHOICE**

The program implements the 2-opt algorithm for the packages delivery requirement. The 2-opt is a local search heuristic algorithm that reorders a route crossing over itself to eliminate the crossing over. It gauges all the probable combinations of the swapping action, and has a time complexity of O(n2). Even though the algorithm has polynomial time complexity like competing algorithms such as the Nearest Neighbor algorithm, it is able to generate optimal solutions (Nalepa & Xhafa, 2019, p. 103).

1. **PROGRAM OVERVIEW**
2. **Algorithm Logic**

The aim of the algorithm and the program is to discover the most efficient routes and delivery distribution that would make it possible for WGUPS to quickly and efficiently do their daily deliveries of 40 packages within 140 miles using 2 trucks.

**Assumptions**

Since it is assumed that the delivery and loading times are instantaneous, the process for reading the data files is not an important aspect in this overview. It is also assumed that the amount of gas in the trucks is infinite, package IDs are unique, there are no collisions, and the distances are equal regardless of the direction travelled.

Some assumptions and parameters affect the operation of the algorithm. It is assumed a truck can carry a maximum of 16 packages, and travels at an average speed of 18 miles per hour, a day ends when all 40 packages reach their destinations, only two drivers are available for deliveries, and the earliest a truck can leave the hub is 8:00 a.m. A note can be associated with a package. The delivery address for package #9 is wrong and would be corrected at 10:20 a.m.

**2-Opt Algorithm and Pseudocode**

The mechanism of the 2-opt algorithm involves computing the best distance/ route, and performing swaps to manipulate the specific routes. The Pseudocode for the algorithm is as follows:

**2opt swap (route, vertex1, vertex2):**

* 1. new\_route = []
  2. Get routes from route[0] to route[vertext1]; insert the routes in order into new\_route
  3. Take route[vertex1 + 1] to route[vertex2] ; insert the routes in reverse order into new\_route
  4. Take route[vertex2 + 1] to route[0] ; insert the routes in order into new\_route
  5. Return new\_route

**find\_best\_route(nodes\_list[]):**

best\_distance = computeTotalDistance(current\_route)

while best\_route not changing:

for i = 1 to len(nodes\_list) - 1 :

for j = i + 1 to len(nodes\_list):

new\_route = 2opt\_swap(current\_route, i, j)

new\_distance = computeTotalDistance(new\_route)

if best\_distance >= new\_distance:

current\_route = new\_route

best\_distance = new\_distance

1. **Programming Environment**

The program was developed using the PyCharm Community Edition 2022.2 and Python 3.11. It was developed under Windows 11, although the code can run without any changes under other operating systems including Mac OS, Linux and Unix systems. The only requirement is the environments must have Python 3 installed.

1. **Space and Time Complexity**

The program is made up of separate segments (functions and structures) that contribute to the overall time and space complexity of the program. The “run” method includes code that is the heart of the program. Its time and space complexity dictate the overall complexity of the program. Other main portions of the program include methods for printing all trucks and printing all packages.

**The “run” method**

The “run” function is loads data, determines the most efficient route to deliver packages, and calculates the delivery distributions of WGUPS. The run time and space of this function is dependent on the number of delivery locations (n) and the number of packages (m). It runs in O(n2) + O(m) time, and its space complexity is also O(n2) + O(m). The function calls other smaller functions to perform specific actions.

**The “load\_packages” method**

The time and space complexities of the load\_packages method is O(n).

**The “load\_distances” method**

The load\_distances method uses a nested for loop to load distances of the routes into a graph structure, which uses an underlying hash table. The time and space complexities of the function is O(n2). The time complexity of putting and inserting a vertex or an edge into the graph is O(1) on average and O(n) in the worst case. These run times are determined by the insertion and retrieval operations of the underlying hash table structure. The space complexities of the structures are O(n). As a result,

**The “deliver\_urgent\_packages” and “get\_remaining” methods.**

The “deliver\_urgent\_packages” and “get\_remaining” methods each run in O(n) time. The space complexity of each is also O(n).

**Graph and Hash Table**

The time and space complexities of the hash table and the graph structure are important because they are used in the program for storing and providing related information where needed. The graph structure uses a hash table as the underlying data structure. On average, the time complexity of inserting data into and retrieving data from a hash table is O(1). In the worst case, the time complexity is O(n). Consequently, the graph structure has the same time complexities for insertion and retrieval. The space complexity of the hash table is O(n). Accordingly, it costs O(n) space to store vertices and edges in the graph.

**Printing truck and packages methods**

Either the time or space complexity of the method for printing all trucks is O(n). The run time of the method is dependent on the number of trucks (n). The method for printing all packages is O(n), where n in this case refers to the number of packages.

**Overall complexity**

The running time of the program grows quadratically with the number of delivery locations. The overall run time of the program is therefore O(n2). Space complexity is also O(n2).

1. **Scaling Capability**

The algorithm is capable of scaling and adapting to growing number of packages. More packages can be used to runs the programs as long as their related distances are provided. However, the performance of the algorithm is likely to be affected by a very large dataset because its runtime is polynomial in nature.

1. **Efficiency and Maintainability**

For an algorithm that solves a non-deterministic polynomial-time (NP) hard problem to run in O(n2) time is efficient.

The program is also highly maintainable because it is broken into smaller units (classes and functions) that perform specific functions. It adheres to the principles of object oriented programming. The code is well documented with sufficient comments within the code.

1. **Strengths and Weaknesses of Hash Table**

A hash table is used for storage and as the underlying structure of the graph structure because it is fast and efficient to retrieve and store data. As Tapia, García and Hernandez (2022) have highlighted, insertion, search and retrieval operations of a hash table are amortized constant time. On average, the operations take O(1) time (Gittleman, 2003). For the hash table used in the program, it is particularly advantageous with large data because it uses separate chaining method for collision resolution.

With separate chaining, data can be added to the hash table without limitation. The technique is less sensitive to load factors, and it is easy to implement. The downside of separate chained hash table is wastage of storage space, and in worst case scenario search can take O(n) time.

Because a hash table does not preserve the order of the keys, it not possible to iterate over the keys in a sequential manner. This limitation usually necessitates use of another data structure, which increases complexity and space requirements.

**Overhead**

Separate-chained hash tables use extra space for storing links, and a lot of space is wastage due to unused space. The use of additional data structure to order data in a hash table is extra cost to the program.

**Implication**

The use of a hash table enhances the running time of the program. Storing and retrieving data is relatively faster compared to alternative data structures such as lists and arrays. Because of the less sensitivity to load factor of the separate-chained hash table, performance is enhanced due to minimal collision. However, there is memory and resource wastage that is inherent of the separate chaining method.

1. **IDENTIFICATION OF A SELF-ADJUSTING DATA STRUCTURE**

A hash table is used with the 2-opt algorithm to store package data. The hash table is utilized as an underlying data storage for a graph structure. Each vertex of the graph is stored as a key. The value corresponding to a key contains a pair of the connected vertex and the weight of the edge.

1. **HASH TABLE DEVELOPMENT**

The code below is the implementation of the hash table.

|  |
| --- |
| class HashTable(object):  """  Represents a hashtable data structure.  """  def \_\_init\_\_(self, length=4):  # for storing items. Initially it's empty.  self.\_buckets = [None] \* length  def \_\_hash(self, key):  """  Returns hash that represents the index in the buckets  array for the given key  """  length = len(self.\_buckets)  return hash(key) % length  def put(self, key, value):  """ Inserts add a value in this hashtable. """  index = self.\_\_hash(key)  if self.\_buckets[index] is not None:  # The index has a bucket; chech whether to update  # or add as new.  for kvp in self.\_buckets[index]:  # The key exists; so replace value.  if kvp[0] == key:  kvp[1] = value  break  else:  # No matching key was found; add it with its value to the end.  self.\_buckets[index].append([key, value])  else:  # The hash index is empty. Create a list and insert the key/value.  self.\_buckets[index] = []  self.\_buckets[index].append([key, value])  def get(self, key):  """ Returns the value pointed to by the given key """  index = self.\_\_hash(key)  if self.\_buckets[index] is None:  return None  else:  # Return the associated value if the given key exists  # in the bucket pointed by the hash index.  for kvp in self.\_buckets[index]:  if kvp[0] == key:  return kvp[1]  # No matching key was found.  return None  def \_\_contains\_\_(self, key):  for (k, \_) in self.\_buckets[self.\_\_hash(key)]:  if k == key:  return True  return False  def \_\_iter\_\_(self):  for bucket in self.\_buckets:  yield from bucket |

1. **JUSTIFICATION OF USING 2-OPT ALGORITHM AND ALTERNATIVE ALGORITHMS**
2. **Strengths**

The 2-opt algorithm has visible strengths is solving the package delivery and routing problem. One of its strength is the ability to generate optimal solutions (Nalepa & Xhafa, 2019, p. 103). In the case of the truck routing problem, it is able to discover efficient routes and distribution such all the 40 packages are distributed by two trucks in under 140 miles. The other advantage of the algorithm is its feasibility in terms of computation requirements. The algorithm runs in O(n2) time, which better than that of the Greedy algorithm (O(n!).

1. **It meets the scenario requirements**

The 2-opt algorithm met all the requirements of the scenario as observed by running the program. The program is able to discover efficient routes and deliver all the 40 packages in less than 140 miles. It took approximately 116.5 miles to deliver the packages.

1. **Alternative algorithms**

Other algorithms that can also be used in place of 2-opt algorithm include the Nearest Neighbor (NN) algorithm and the Greedy algorithm. The Nearest Neighbor algorithm also

1. **WHAT I WOULD DIFFERENTLY**

If I did the project again, I would use alternative data structures such as an adjacency list or an adjacency matrix to store package data. I would use the structure as the underlying storage for the graph structure. The structure would be configured to all the information about the packages.

1. **JUSTIFICATION OF THE DATA STRUCTURE (HASH TABLE)**
2. **Data structure and meeting requirements**

The hash table data structure used in the solution is suitable for storing package data. The structure is used as the underlying storage in a graph structure. Look-up action is fast. On average, it takes O(1). As such, changes in the number of packages does not affect the look-up function. In the worst case, look-up action can take O(n). It means that changes in the number of packages affect the look-up function in a linear manner. Similarly, changes in the number of packages affect the hash table space usage in a linear manner. It takes O(n) space to store package data using the hash table data structure.

The relationship between changes to the number of trucks or number of cities and look-up time or space usage is also linear.

1. **Other data structures that could meet requirements**

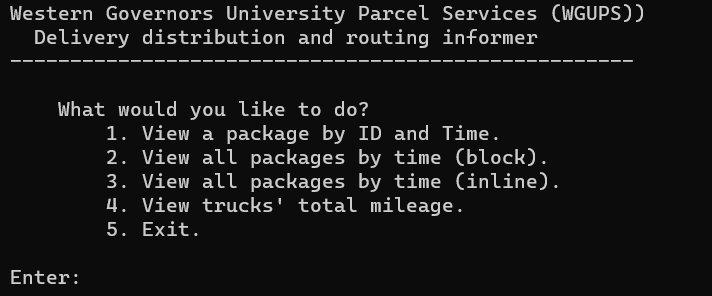
Alternative data structures that could be used in place of a hash table include an adjacency list and an adjacency matrix. Both structures can be used as the underlying data structure of a graph. An adjacency list is an array of linked lists, while an adjacency matrix is a square matrix. These structures are different from the hash table used in the solution.

The insertion and look-up in an adjacency matrix takes O(1) time. However, in a adjacency list, it takes O(1) time to do insertion and O(V) time to look up. In a hash table both look up and insertion are O(1) on average, but O(n) in the worst case.

The space complexity of an adjacency list is O(V+E), on average and O(V2) in the worst case. V and E refer to the number of vertices and the number of edges respectively. Space complexity of matrix is O(V2). The space complexity of a hash table of O(n) is comparable to that of a adjacency matrix and to the worst case for adjacency list.

1. **INTERFACE AND SCREENSHOTS**

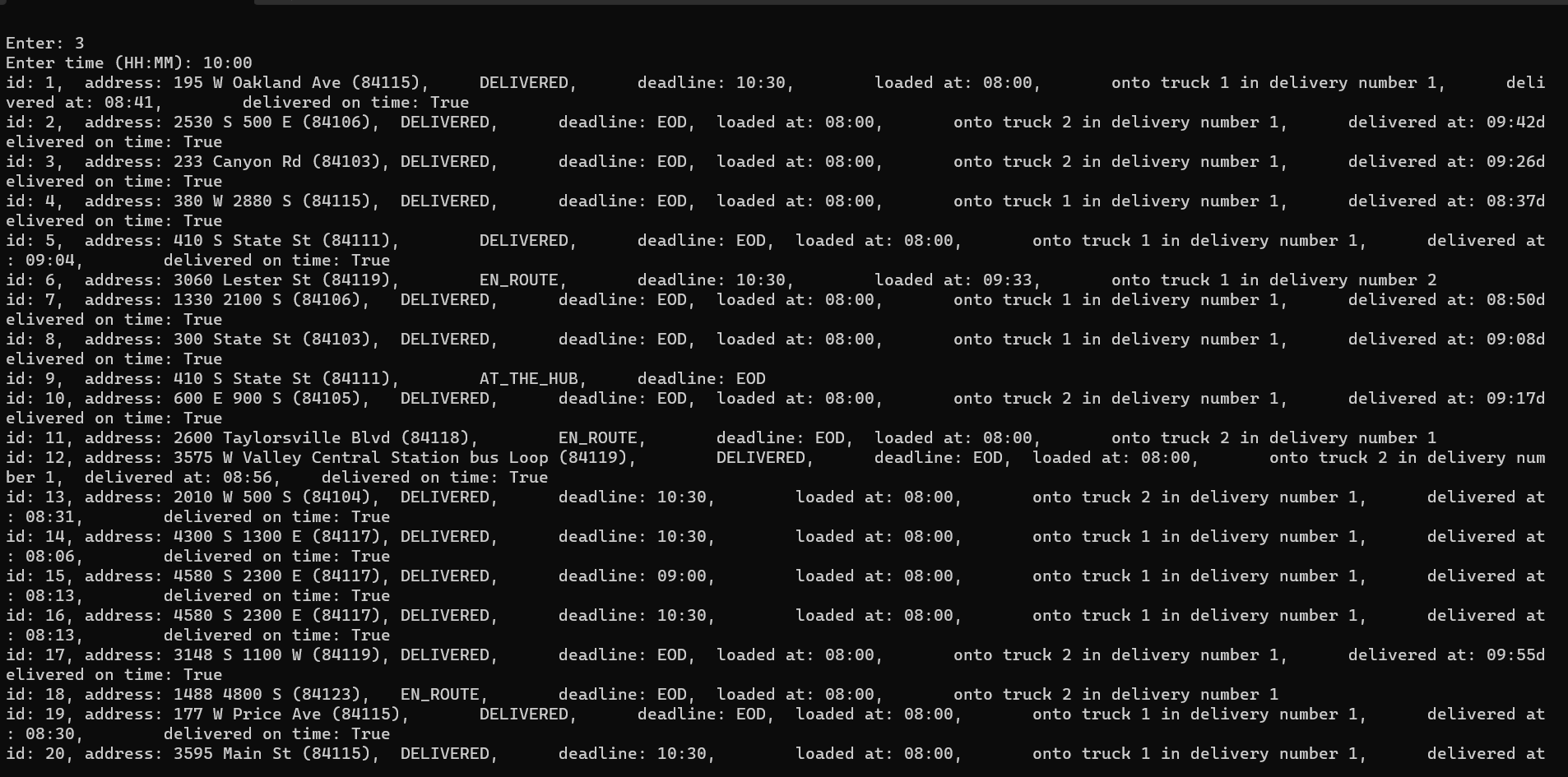
The following screenshot shows the programs user interface that allows a user to run the program and query specific information.



The following screenshots show the status of all packages at various time periods:

Time: 8:40



Time: 10:00

Time: 1:00



**References**

Gittleman, A. (2003). *Computing with C# and the .NET framework.* Sudbury, MA: Jones and Bartlett Publishers.

Gutin, G., Yeo, A. & Zverovich, A. (2002). Traveling salesman should not be greedy: domination analysis of greedy-type heuristics for the TSP. *Discrete Applied Mathematics* *117*(6), 81-86.

Nalepa, J. & Xhafa, F. (2019). *Smart delivery systems: solving complex vehicle routing problems.* Netherlands: Elsevier.

Tapia, S., García, D. and Hernandez, P. (2022). Key Concepts, Weakness and Benchmark on Hash Table Data Structures. *Algorithms, 15*(3), 100.