Name: **Russell Taylor**
Student ID: **001441098**
Course: **C950 Data Structures and Algorithms II**
Date: **October 31, 2020**

DOCUMENTATION

A: ALGORITHM SELECTION

Identify the algorithm that will be used to create a program to deliver the packages and meets all requirements specified in the scenario.

The problem given is a classic traveling salesman problem (https://en.wikipedia.org/wiki/Travelling_salesman_problem). This program uses a **nearest neighbor algorithm**. Because this is an approximation algorithm, the result is not an optimal route, but it does fall comfortably within the given constraints.

B1: LOGIC COMMENTS

1. Write a core algorithm overview, using the sample given, in which you do the following: 1. Comment using pseudocode to show the logic of the algorithm applied to this software solution.

Stated Problem:
The purpose of this project is to create an algorithm using Python to find an approximate solution to a traveling salesman problem where packages are being delivered from the WGU warehouse to 40 nearby locations in the least amount of time and mileage. The locations are divided between two trucks that run concurrently. In the solution implemented here, each truck divides its locations into two trips and therefore runs through this algorithm twice.

Algorithm Overview:
1. Initialize a list of all locations for a given route
2. Mark the first location (the WGU warehouse) as the current location
3. Find the location that is nearest to the current location
4. Mark the nearest location as the current location
5. Record the distance between the previous location and the current location
6. Delete the previous location
7. Deliver packages at the current location
8. Repeat steps 3–8 until no locations remain
9. Return to the first location (the WGU warehouse)

Pseudocode:

```
Initialize locations (load packages on truck)
```

```
for p in list_of_packages:
     packages_on_truck.add(p)
     locations_to_visit.add(p.location)
     p.set_status('On Truck')
```

Mark the first location as current

```
current = wgu_warehouse.location
```

Find the location that is nearest to the current location

```
distance = maxsize
for loc in current.adjacency_list:
     if loc in locations_to_visit:
          if distance > loc.distance:
               distance = loc.distance
               next_location = loc
```

Mark the nearest location as the current location

```
previous_location = current
current = next_location
```

Record the distance between the previous location and the current location

```
total_distance += distance
```

Delete the previous location

```
locations_to_visit.remove(previous_location)
```

Deliver packages at the current location

```
for p in packages_on_truck:
     if p.location == current:
          p.set_status('Delivered')
          packages_on_truck.remove(p)
```

Repeat steps 3–8 until no locations remain

```
while len(locations_to_visit) > 0:
```

Return to the first location (the WGU warehouse)

```
for loc in current.adjacency_list:
     if loc = wgu_warehouse.location:
          total_distance += loc.distance
          current = loc
```

B2: APPLICATION OF PROGRAMMING MODELS

2. Apply programming models to the scenario.

This program was written and tested in PyCharm Professional 2020.2 using Python 3.8.5.

The data is imported from four csv files containing information about the distances between the package delivery locations (locations.csv), the delivery address, delivery time, and other information about the packages (packages.csv), the routes to be taken by each truck (routes.csv), and alerts about any special circumstances (alerts.csv). The program uses Python's built-in csv module to import the data.

The program is organized using object-oriented methodology and classes of objects representing the data and functions to perform on that data. The program classes are organized using data-access-object methodology. The data is contained in the included csv files. The main.py, interface.py, and deliveries.py modules provide access to the data, and the locations.py, packages.py, routes.py, and alerts.py modules provide the framework for the objects that represent and allow for manipulation of the program data. Three additional utility classes provide essential functionality: the data.py module handles importing data from the csv files, the time.py module handles any time tracking, conversions, and display, and the hashtable.py module performs all hashing functions.

Communications protocols, server application programs, and interaction semantics to control connection, data exchange, and disconnection are not applicable to this project since it runs on the local machine and all data is stored either in csv files on the local machine or in the local machine's computer memory. No connection is required.

B3: SPACE-TIME AND BIG-O

3. Evaluate space-time complexity using Big O notation throughout the coding and for the entire program.

Time Complexity:

The overall time complexity of the program is $O(n^2 \times k)$ where n is the number of locations and k is the number of packages, or **$O(n^3)$** if n=k. The time complexity is dominated by the nearest neighbor algorithm in `get_location()` in deliveries.py.

main.py O(1)

interface.py

`show_options()` O(1) if the user inputs a valid choice, otherwise O(n) where n is the number of times the user enters an invalid choice

`get_choice()` O(1) if the user inputs a valid choice plus the time complexities of data.py and time.py

`run_simulation()` O(1) plus the time complexity of deliveries.py

`status_report()` O(n) where n is the number of packages, plus the time complexities of `input()` and `run_simulation()`

`lookup_packages()` O(1) plus the time complexities of `input()` and `run_simulation()`

`input()` O(n) if the user inputs a valid choice where n is the number of valid choices, otherwise O(n × k) where n is the number of valid choices and k is the number of times the user enters an invalid choice

`quit()` O(1)

deliveries.py O(n^2 × k) where n is the number of locations and k is the number of packages per route

`deliver_packages()` O(n) where n is the number of locations, multiplied by the time complexities of `get_location()` and `make_deliveries()`, plus the time complexities of `load_truck()` and `return_to_start()`

`load_truck()` O(n) where n is the number of packages per route

`get_location()` O(n × k) where n is the number of locations and k is the number of packages per route

`make_deliveries()` O(n) where n is the number of packages per route

`return_to_start()` O(1)

`alert()` O(n) where n is the number of alerts

data.py O(n × k) where n is the number of packages and k is the number of locations

`read_locations()` O(n) where n is the number of locations

`read_packages()` O(n × k) where n is the number of packages and k is the number of locations

`read_routes()` O(n) where n is the number of packages

`read_alerts()` O(n) where n is the number of alerts

time.py O(1)

hashtable.py O(n) where n is the number of entries in the hash table

`get_hash()` O(n) where n is the number of characters in the key string

`add()` O(n) where n is the number of entries in the hash table

`get()` O(n) where n is the number of entries in the hash table

locations.py O(1)

packages.py O(1)

routes.py O(1)

alerts.py O(1)

Space Complexity:

The overall space complexity of the program is **O(n²)** where n is the number of locations. The space complexity is dominated by the adjacency list which, for each location, stores the distances to every other location.

main.py O(1)
interface.py O(1)
deliveries.py O(n) where n is the number of packages
data.py O(n²) where n is the number of locations
time.py O(1)
hashtable.py O(n) where n is the number of entries in the hash table
locations.py O(1)
packages.py O(1)
routes.py O(1)
alerts.py O(1)

B4: ADAPTABILITY

4. Discuss the ability of your solution to adapt to a changing market and to scalability.

The nearest neighbor algorithm "on average yields a path 25% longer than the shortest possible path" and as the dataset expands it is possible that the algorithm will generate "the worst route" (https://en.wikipedia.org/wiki/Travelling_salesman_problem). However, because the traveling salesman problem is a NP-Hard problem, an optimal route is extremely difficult to determine and attempting to do so is not feasible for a program of this sort, even as the dataset grows. Therefore, the chosen algorithm will generally be sufficient unless the dataset grows very large, at which point additional revenue should allow for additional investment in producing a more sophisticated algorithm.

The program itself is specifically designed with growth and adaptability in mind. It is fully object oriented, and the classes are designed to be modular and easy to update with minimal refactoring throughout the rest of the program.

The program is also designed to work with various datasets and in other markets. The user simply needs to update the four csv data files with data specific to their needs. New data should follow the same format as the data in the existing files. The locations.csv and packages.csv files allow the program to adapt to deliveries in new markets. The routes.csv file allows the user to manually optimize the routes and make granular changes where dictated by the circumstances. And changes to the alerts.csv file will allow the user to interrupt the delivery schedule at any time for various reasons including delayed packages, address errors, updates to routes, etc.

B5: SOFTWARE EFFICIENCY AND MAINTAINABILITY

5. Discuss the efficiency and maintainability of the software.

As the time and space complexity analyses above show, this program operates in $O(n^2 \times k)$ time, or $O(n^3)$ if $n=k$, and requires $O(n^2)$ space. The use of hash tables allow data lookup in $O(1)$ time which has reduced the time complexity significantly compared to an implementation that stores the data in simple lists or other data structures.

As the adaptability discussion above has shown, this program was specifically written to be easy to maintain and use in various markets and circumstances. Additionally, the code is well documented through the use of carefully chosen variable names, efficient and clean code that eliminates clutter and duplication, generous comments throughout, and comprehensive documentation accompanying the program. Future programmers will find the code easy to read, maintain, and adapt.

B6: SELF-ADJUSTING DATA STRUCTURES

6. Discuss the self-adjusting data structures chosen and their strengths and weaknesses based on the scenario.

The code implements a graph data structure and stores the graph's vertices and edges and vertices in adjacency lists within a hash table. The packages, routes, and alerts information is also stored in hash tables.

The **hash tables** are self-adjusting because every time a new element is added, it is automatically placed in the correct location, which changes the shape of the data as a whole. Moreover, the data structure is abstracted enough to allow input of various types and sizes. The same data structures is used four different times to accommodate locations, packages, routes, and alerts data.

Likewise the **graph** is self-adjusting because whenever new vertices and edges are added or removed, it automatically changes the way the data may be processes. This adaptation is on display in the functioning of the nearest neighbor algorithm which automatically determines the next location, depending on the data present in the graph.

C: ORIGINAL CODE

Write an original code to solve and to meet the requirements of lowest mileage usage and having all packages delivered on time.

Please see the included code

C1: IDENTIFICATION INFORMATION

1. Create a comment within the first line of your code that includes your first name, last name, and student ID.

Please see the included code

C2: PROCESS AND FLOW COMMENTS

2. Include comments at each block of code to explain the process and flow of the coding.

Please see the included code

D: DATA STRUCTURE

Identify a data structure that can be used with your chosen algorithm to store the package data.

The data for each package is stored as an object in the Package class. The class structure allows for storing each piece of information about the package as an instance variable, and allows for quick access using getter and setter methods.

The Package objects are then stored in a **hash table**. Insertion and lookup are both O(1), so the hash table significantly increases the efficiency of the algorithm compared to other types of data structures.

(Goodrich, Data Structures and Algorithms in Python, 410)

The edge and vertex data for the **graph** data structure is stored in the Location class objects. Each Location object is then stored in another hash table.

D1: EXPLANATION OF DATA STRUCTURE

1. Explain how your data structure includes the relationship between the data points you are storing.

The hash table is able to account for the relationship between data points because the keys used to store the information in the table are sequential ID numbers. Therefore, all items stored in the hash table can be accessed by iterating through the range of ID numbers. And any item can be accessed directly if the ID number is known.

The relationship between the data points within each item in the hash table (package ID, address, weight, etc.) is accounted for by storing the data within objects before adding each object to the hash table. The object's instance variables store the individual pieces of information.

The graph accounts for the relationship between data points using an adjacency list. Each Location object represents a vertex. Each Location object has a 'distances' instance variable which contains a list of distances to every other vertex. The index of each distance in the list corresponds to the location ID of the corresponding vertex. This allows for O(1) lookup of any vertex (since the Location objects are stored in a hash table), any adjacent vertex (using the indices of the 'distances' list), and the weight of the edge connecting the two vertices (the corresponding value in the 'distances' list).

(Goodrich, Data Structures and Algorithms in Python, 620)

E: HASH TABLE

Develop a hash table, without using any additional libraries or classes, with an insertion function that takes the following components as input and inserts the components into the hash table: • package ID number • delivery address • delivery deadline • delivery city • delivery zip code • package weight • delivery status (e.g., delivered, in route)

Please see the `insert()` function in the hashtable.py file

F: LOOK-UP FUNCTION

Develop a look-up function that takes the following components as input and returns the corresponding data elements: • package ID number • delivery address • delivery deadline • delivery city • delivery zip code • package weight • delivery status (e.g., delivered, in route)

Please see the `lookup()` function the hashtable.py file, as well as the `lookup_package()` function in the interface.py file

G: INTERFACE

Provide an interface for the insert and look-up functions to view the status of any package at any time. This function should return all information about each package, including delivery status.

Please see the included code in the interface.py file

G1: FIRST STATUS CHECK

1. Provide screenshots to show package status of all packages at a time between 8:35 a.m. and 9:25 a.m.

Please see the included 'G1 - 2 Status Report at 8.50.00 AM.png' file

G2: SECOND STATUS CHECK

2. Provide screenshots to show package status of all packages at a time between 9:35 a.m. and 10:25 a.m.

Please see the included 'G2 - 2 Status Report at 10.22.00 AM.png' file

G3: THIRD STATUS CHECK

3. Provide screenshots to show package status of all packages at a time between 12:03 p.m. and 1:12 p.m.

Please see the included ‘G3 - 2 Status Report at 12.40.00 PM.png’ file

H: SCREENSHOTS OF CODE EXECUTION

Run your code and provide screenshots to capture the complete execution of your code.

Please see the included files:
‘H - 1A Delivery Simulation.png’
‘H - 1B Delivery Simulation.png’
‘H - 1C Delivery Simulation.png’
‘H - 2 Package Status Reports.png’
‘H - 3 Look Up Package.png’

I1: STRENGTHS OF THE CHOSEN ALGORITHM

Justify your choice of algorithm by doing the following: 1. Describe at least two strengths of the algorithm you chose.

One strength of the nearest neighbor algorithm is its simplicity and **ease of implementation**. This saves programmer time and therefore reduces cost. It furthermore increases the maintainability of the code. It avoids the pitfall of applying a complicated heavyweight solution to a simple problem in small programs with limited datasets or in cases where an optimal solution is not needed or possible.

A second strength of the nearest neighbor algorithm is its **speed of execution**. Algorithms that produce an optimal route can take enormous amounts of time to execute since this is a NP-Hard problem. As an approximation algorithm, the nearest neighbor algorithm can produce a generally acceptable result at very fast speeds.

The trade-off is that it is possible for the nearest neighbor algorithm to produce a poor route. “As a general guide, if the last few stages of the tour are comparable in length to the first stages, then the tour is reasonable; if they are much greater, then it is likely that much better tours exist” (https://en.wikipedia.org/wiki/Nearest_neighbour_algorithm).

I2: VERIFICATION OF ALGORITHM

2. Verify that the algorithm you chose meets all the criteria and requirements given in the scenario.

The total mileage of all trucks after running this program is **129.0 miles**:

Truck 1, Trip 1: 28.2 miles

Truck 1, Trip 2: 17.2 miles
Truck 2, Trip 1: 36.8 miles
Truck 2, Trip 2: 46.8 miles

All packages are delivered on time:

Package 1: Deadline 10:30 AM, Delivered at 8:11:40 AM
Package 6: Deadline 10:30 AM, Delivered at 10:14:00 AM
Package 13: Deadline 10:30 AM, Delivered at 8:57:40 AM
Package 14: Deadline 10:30 AM, Delivered at 8:06:20 AM
Package 15: Deadline 9:00 AM, Delivered at 8:13:00 AM
Package 16: Deadline 10:30 AM, Delivered at 8:13:00 AM
Package 20: Deadline 10:30 AM, Delivered at 8:29:40 AM
Package 25: Deadline 10:30 AM, Delivered at 9:42:00 AM
Package 29: Deadline 10:30 AM, Delivered at 8:27:20 AM
Package 30: Deadline 10:30 AM, Delivered at 8:47:00 AM
Package 31: Deadline 10:30 AM, Delivered at 10:09:00 AM
Package 34: Deadline 10:30 AM, Delivered at 8:13:00 AM
Package 37: Deadline 10:30 AM, Delivered at 8:41:40 AM
Package 40: Deadline 10:30 AM, Delivered at 8:15:20 AM

All other packages are successfully delivered before the end of day

I3: OTHER POSSIBLE ALGORITHMS

3. Identify two other algorithms that could be used and would have met the criteria and requirements given in the scenario.

One possible algorithm is a **brute force search**. It tries every possible route and then chooses the one with the lowest cost in time and mileage. The result is an optimal route, but brute force searches have a very high time complexity of O(n!) where n is the number of locations (https://en.wikipedia.org/wiki/Travelling_salesman_problem).

Another approach is to use **the new algorithm discovered this year by Anna Karlin, Nathan Klein, and Shayan Oveis Gharan**. It is the first time that an algorithm has been shown to improve upon the Christofides's algorithm discovered in 1976, although the improvement is only 0.2 billionth of a trillionth of a trillionth of a percent (https://www.quantamagazine.org/computer-scientists-break-traveling-salesperson-record-20201008).

I3A: ALGORITHM DIFFERENCES

a. Describe how each algorithm identified in part I3 is different from the algorithm you chose to use in the solution.

A **brute force** search tries every possible route before selecting which route to actually take. The nearest neighbor algorithm is a greedy algorithm which doesn't decide which route to take until it is actually traversing the graph. The next location is decided one location at a time. The brute force approach is extremely costly and already borders on being unfeasible even with the limited dataset of 40 packages and 27 locations.

The **newest algorithm** to solve the traveling salesman problem is similar to Christofides's algorithm in that it begins with the minimum spanning tree of the graph which can be found in polynomial time with Prim's algorithm or Kruskal's algorithm. Christofides's algorithm then adds connections until every city has an even number of connections, creating a closed loop.

The new algorithm, randomizes the selection of the minimum spanning tree and also specifically chooses a tree where vertices with an odd number of adjacent vertices have low edge weights. This prevents a common problem in Christofides's algorithm where the last connection can be very long.

Like the brute force approach, this too would find determine the route before beginning the deliveries, unlike the nearest neighbor method. Additionally, both of these alternative methods would produce superior optimization, compared to the nearest neighbor method.

J: DIFFERENT APPROACH

Describe what you would do differently if you did this project again.

If I were to do this project again, it would be enjoyable to implement Karlin, Klein, and Gharan's newest groundbreaking method.

I would need to include methods for finding the minimum spanning tree, which could involve the use of a priority queue and disjoint sets if I were to use Kruskal's algorithm. It would also shift the burden of the processing to prior to the beginning of the package deliveries rather than during deliveries, which would increase the efficiency of the program.

K1: VERIFICATION OF DATA STRUCTURE

Justify your choice of data structure by doing the following: 1. Verify that the data structure you chose meets all the criteria and requirements given in the scenario.

All assignment criteria have been met. The least number of total miles has been added to all trucks, all packages are delivered on time, the hash table with look-up function is present, and the reporting needed is accurate and efficient.

K1A: EFFICIENCY

a. Describe the efficiency of the data structure chosen.

Hash table insertion and lookup are both **O(1)**, so the hash table significantly increases the efficiency of the algorithm compared to other types of data structures (Goodrich, Data Structures and Algorithms in Python, 410).

The edge and vertex data for the graph data structure is stored in an adjacency list. Since the algorithm needs to consider all possible vertices at each step, the graph needs to be a complete graph. Therefore, if the adjacency list were stored in a simple array, the time complexity of both insertion and lookup would be $O(n^2)$.

However, because the vertices are stored in a hash table, the insertion and lookup times are reduced to O(n). It takes O(1) time to access the vertex, then O(n) time to access each adjacent vertex. I could have further reduced the time complexity by storing the list of adjacent vertices and edge weights in a hash table as well. However, because the location IDs in this assignment are contiguous and immutable, I was able to achieve the same effect by inserting the edge weights into a list such that the location IDs correspond to the list indices. This allows for **O(1)** insertion and lookup of both vertices and edge weights.

Small, contiguous integers are used as package IDs and location IDs, and for this assignment they are immutable. These IDs are used as keys for the hash tables. This means that there are no collisions, which allows for true O(1) efficiency.

(https://www.baeldung.com/cs/adjacency-matrix-list-complexity)

K1B: OVERHEAD

b. Explain the expected overhead when linking to the next data item.

As detailed in B3 above, the overall time complexity of the program is $O(n^2 \times k)$ where n is the number of locations and k is the number of packages, or **$O(n^3)$** if n=k. The time complexity is dominated by the algorithm to choose a route using the nearest neighbor algorithm.

The overall space complexity of the program is **$O(n^2)$** where n is the number of locations. The space complexity is dominated by the adjacency list which, for each location, stores the distances to every other location.

Bandwidth is not applicable to this program since it runs entirely on the local machine.

K1C: IMPLICATIONS

c. Describe the implications of when more package data is added to the system or other changes in scale occur.

Since the packages, trucks, and locations are each stored in hash tables, as the number of each of these items increase, there will be a need to increase the size of the hash table. The default hash table size is currently set at 101, which is a prime number approximately twice as large as the largest dataset (the number of packages is 40). However, as the program is used for larger datasets, there will be a need to implement automatic resizing of the hash table.

Additionally, as the program is used for larger and more complex datasets in more complex and fluid situations, it is unlikely that the ID numbers for each of these sets of data will remain contiguous small integers. Therefore, it is likely that collisions in the hash table will begin to occur. It will then be beneficial to implement a more efficient collision resolution algorithm such as linear probing, quadratic probing, or double hashing.

The alert system is robust and abstracted enough at present to handle additional alerts of various types. However, as situations grow more complex, it will become cumbersome to input alerts in the specified format, and it will become necessary to input additional alerts on the fly, as the program is running, not simply at the beginning of each run of the program.

If the program is expanded to include numerous concurrent cities and routes, it will be necessary to implement controller classes to manage the various routes more efficiently. At present, only one route may run at a time. The time then resets before the next truck's route. It would be beneficial to implement functionality for multiple routes to run concurrently. Also, the current program depends on pre-sorted package lists for each route. As the use of the program expands, it will become more essential to automate that process as well.

K2: OTHER DATA STRUCTURES

Identify two other data structures that can meet the same criteria and requirements given in the scenario.

The package information could have been stored in a simple **array**. This would have sufficiently met the assignment requirements (except for the instruction to use a hash table). However, the time complexity for insertion and lookup would have increased to O(n).

If the package delivery order was known in advance, the packages could have been stored in a **queue**. A queue would maintain the order of the packages and would allow for O(1) lookup since the next package is always first in the list.

The graph data could have been stored in an **adjacency matrix**. Since the location and distance data was already provided in this format, it would have simplified the process of importing the data. However, insertion and lookup of items in an adjacency matrix is O(n^2).

K2A: DATA STRUCTURE DIFFERENCES

a. Describe how each data structure identified in part K2 is different from the data structure you chose to use in the solution.

An **array** is a simple list of data elements. Each element can be either a basic data type or a complex data type such as an object or another array. In general, inserting and accessing data requires iterating through the elements in the list, which takes O(n) time. If the index of the element is known, access can be as fast as O(1) time. A hash table requires knowledge of the index/key of the element, and offers more flexibility in terms of what key values are allowed. Rather than requiring contiguous integers starting at zero as indices, the keys can be any value determined by the user. Knowledge of the key allows for O(1) insertion and access time.

A **queue** is a simple list with the added requirement that elements may only be added at the back of the list and may only be removed at the front of the list. The additional constraint means that access time for the element at the front of the list is always O(1). However, a queue prevents random access of elements and would therefore require significant changes to the program structure and primary algorithm.

An **adjacency matrix** stores the graph data such that the list of vertices is on both the x and y axis, while the edge weights are placed in each row and column. In general, inserting and accessing data requires traversing the matrix, which involves nested for loops and O(n^2) time. Also, if the graph is not a complete graph, much of the matrix can be left empty, resulting in wasted memory space. However, since the graph used in this program is complete, this is not a consideration.

L: SOURCES

Acknowledge sources, using in-text citations and references, for content that is quoted, paraphrased, or summarized.

Islenia Mil, "Computer Scientists Break Traveling Salesperson Record," October 8, 2020. Accessed October 30, 2020. https://www.quantamagazine.org/computer-scientists-break-traveling-salesperson-record-20201008

Kenneth H. Rosen, *Discrete Mathematics and Its Applications*, 8th ed., 2019.

Michael T. Goodrich, Roberto Tamassia, and Michael H. Goldwasser, *Data Structures and Algorithms in Python*, 2013.

"Nearest Neighbour Algorithm," March 10, 2020. Accessed October 30, 2020. https://en.wikipedia.org/wiki/Nearest_neighbour_algorithm

"Time and Space Complexity of Adjacency Matrix and List," October 19, 2020. Accessed October 30, 2020. https://www.baeldung.com/cs/adjacency-matrix-list-complexity

"Traveling salesman problem," October 25, 2020. Accessed October 30, 2020. https://en.wikipedia.org/wiki/Travelling_salesman_problem