Course ID: C950
Course title: Data Structures and Algorithms II
Project Title: Task 1 - WGUPS delivery algorithm
Student name: Lawrence Sanzogni
Student ID: 012143472

Introduction

This written presentation will cover a delivery algorithm that ensures timely and accurate parcel deliveries for an itinerary for the WGUPS delivery service.

Task I

Task A

The self-adjusting algorithm used will be a 'greedy' algorithm. The algorithm will select the next delivery based on priority, which will be determined by the delivery deadline and distance from the current stop.

Task B

There will be three data structures implementing this algorithm. The first is an array of dictionaries, where the array is the truck's delivery queue and each dictionary is a package, each with its attributes. The 2nd data structure is a hash table of delivery distances. The keys are the current destination, and the buckets will contain all other destinations and their relative distance from the current destination. The third is a package hash table, which will keep self-adjusting by keeping track of each package's delivery status.

The three data structures will work together as follows: the first package in the truck array represents the default 'nearest package', based on driving distance. The package will perform a lookup on the hash table, the current delivery address as an index. Once the bucket is found, it will check all destination addresses in the bucket and determine what the next address will be based on the aforementioned priority criteria.

Once the next destination has been selected, the current package will be dequeued from the array, and the next package will be assigned as the current package and moved to the front of the array.

Task C

C_1

This is a pseudocode version of my algorithm:

```
l:1 use our current stop as a reference
l:2 assume the first package in the queue has the shortest driving distance
l:3 look through the rest of the packages in the package list..
l:4   if we still have priority packages left to deliver (non-EOD deadline)
l:5     if the current package is a non-priority package
l:6       do nothing and skip to the next package
l:7     otherwise
l:8       If the current package’s driving distance is nearer to the
          current stop and it hasn’t already been delivered
l:9         assume it’s the nearest package and reassign the ‘nearest
            package’ to the current package
l:10  otherwise
l.11    if the current package’s driving distance is nearer and it hasn’t
        been delivered yet
l.12      assume it’s the nearest package and reassign the ‘nearest
          package’ to the current package
```

C_2

I will be running my program using Python in the PyCharm IDE. The hardware running the program will be a 2020 M1 Macbook Pro with 16 GB of RAM.

C-3

These are the following major sections of my program and their corresponding space-time complexity:

Section: The ‘packages’ hash table and the methods to populate it:

```
#main.py
with open('WGUPS Package File_edited.csv') as package_file_csv:
   read_package_csv = csv.reader(package_file_csv, delimiter=',')

   # skip file headers
   next(read_package_csv)

   for row in read_package_csv:
       # initialize package
       truck_package = package.Package(int(row[0]), row[1], row[2], row[4], row[5],
row[6], row[7], 'at the hub')
       # add package to package hash table
       packages.insert_package(truck_package)
```

```
#packages.py
def insert_package(self, package):
   key = package.delivery_id
   index = self._hash(package.delivery_id)

   if package.delivery_deadline != 'EOD':
       self.other_than_EOD_packages += 1

   if index not in self.packages:
       self.packages[index] = package
   else:
       current = self.packages[index]
       while current.next:
           current = current.next
       current.next = package

   self.size = self.size + 1
```

Methods:

- Populating the ‘packages’ hash table: **O(n)**. A package object is created for each field in the packages.csv file and is added to the packages hash table iteratively.

Section: The ‘distance’ hash table and the methods to populate it:

```
#main.py
distances.create_hash()
distances.fill_hash()

#distance_hash.py
def create_hash(self):
   self.hash = {}
   self.size = 0
   # read distance rows from csv file
   with open('WGUPS Distance Table_edited.csv', 'r', newline='') as
distance_table_csv:
       read_distance_csv = csv.reader(distance_table_csv, delimiter=',')

       # get file header
       header = next(read_distance_csv)

       header_index = 0
       for row in read_distance_csv:
           row_header = row[0]
           try:
               row_header = row[0][0:row[0].index('(')].replace('\n', '')
           except ValueError as e:
               row_header = row_header.replace(' ', '')

           # replace column headers with row headers to get matching keys
           header[header_index] = row_header
           header_index += 1

           ## set bucket keys, with a default empty bucket
           if row_header not in self.hash:
               self.hash[row_header] = {}
               self.size += 1

           ## populate buckets
           n = len(row)
           for j in range(1, n):

               ### check if intercepting distance is in cell
               if row[j] != '':
                   self.hash[row_header][header[j-1]] = row[j]

def fill_hash(self):
   # this method fills out the tables - even if there is redundant data to simplify
looking up and comparing distances for nearest neighbor algorithm
   for outer_key, outer_bucket in self.hash.items():
       for inner_key, inner_bucket in self.hash.items():
           if outer_key in inner_bucket and inner_key not in outer_bucket:
               outer_bucket[inner_key] = inner_bucket[outer_key]
```

Methods:

- Creating the ‘distances’ hash table:
    - Creating the header: **O(n)**. This method begins by iteratively creating the bucket keys using the left-most addresses and then parsing the keys to ensure that they are identical to their column-key counterparts.
    - Populating the hash table with the triangular matrix values: **O(n^2)**. Each header has a list of address distances, and those distances are used to populate buckets in the hash table. The worst-case scenario, one address has distance measurements for all other addresses.
    - Filling the hash table: **O(n^2).** This method is to fill out the empty cross values from the original hash table, transforming the data structure from a triangular matrix to a square matrix. This is to make looking up the nearest next stop easier in the nearest neighbor algorithm. This requires cross-checking every address in the distances table again to make sure that there aren’t any empty cells.

  Total space-time complexity: **O(n^8)**

Section: the delivery algorithm and the nearest neighbor algorithm to calculate the next stop

```
#truck.py
def deliver_package(self, distance_hash):

   #EDGECASE: user input time is before truck's official departure time
   if self.end_time <= self.current_time:
       self.current_time = self.end_time
       return
   if self.user_time and self.user_time <= self.current_time:
       self.current_time = self.user_time
       return

   #EDGECASE: if we're at the hub
   if self.current_stop == 'HUB':
       # find the nearest stop
       nearest_package = self.nearest_neighbor(distance_hash)

       #once we've selected the package with the nearest address:
       self.current_stop = nearest_package['address']
       self.previous_stop = 'HUB'
       # deliver the package at the next stop
       self.deliver_package(distance_hash)

   # if we're en route
   for i in range(0, len(self.packages)):
       current_package = self.packages[i]
       # drop off package at current stop
       if current_package.delivery_address == self.current_stop and
current_package.delivery_status == 'en route':
           # add mileage
           current_distance =
float(distance_hash[current_package.delivery_address][self.previous_stop])
           # update time
           self.update_time(math.ceil((current_distance / self.speed) * 60))
           # EDGECASE: user time entered is between deliveries
           if self.user_time and self.current_time > self.user_time:
               self.current_time = self.user_time
               return
           self.update_mileage(current_distance)
           # update package deliver time
           current_package.delivery_time = self.current_time
           # mark package as delivered
           current_package.delivery_status = 'delivered'
           # update priority packages
           if current_package.delivery_deadline != 'EOD':
               self.priority_packages -= 1
```

```
            # update inventory
            self.update_inventory(current_package)
            #update last delivered package ID
            self.last_packaged_delivered_ID = current_package.delivery_id
            break

    # BASE CASE: if we're at our last top
    if self.current_time > self.end_time or self.inventory == 0:
        return

    # determine the next stop
    nearest_package = self.nearest_neighbor(distance_hash)

    temp = self.current_stop
    self.current_stop = nearest_package['address']
    self.previous_stop = temp

    while self.current_time < self.end_time and self.inventory > 0:
        self.deliver_package(distance_hash)
```

```
#truck.py
def nearest_neighbor(self, distance_hash):
    nearest_package = {}
    i = 0

    # if all priority packages have been delivered
    if self.priority_packages == 0:
        while i < len(self.packages):
            current_package_status = self.packages[i].delivery_status
            if current_package_status == 'delivered':
                i = i + 1
                continue
            nearest_package = {
                'address': self.packages[i].delivery_address,
                'distance':
float(distance_hash[self.packages[i].delivery_address][self.current_stop])
            }
            if i == len(self.packages) - 1:
                break
            j = i + 1
            while j < len(self.packages):
                current_package_address = self.packages[j].delivery_address
                current_package_distance =
float(distance_hash[current_package_address][self.current_stop])
```

```
                if current_package_distance < nearest_package['distance']
and self.packages[j].delivery_status == 'en route':
                    nearest_package['address'] = current_package_address
                    nearest_package['distance'] = current_package_distance
                j = j + 1
            i = len(self.packages)
   else:
   # if there are still priority packages left
        while i < len(self.packages):
            current_package_status = self.packages[i].delivery_status
            priority_package = self.packages[i].delivery_deadline != 'EOD'
            if current_package_status == 'delivered' or not
priority_package:
                i = i + 1
                continue
            nearest_package = {
                'address': self.packages[i].delivery_address,
                'distance':
float(distance_hash[self.packages[i].delivery_address][self.current_stop])
            }
            if i == len(self.packages) - 1:
                break
            j = i + 1
            while j < len(self.packages):
                current_package_address = self.packages[j].delivery_address
                current_package_distance =
float(distance_hash[current_package_address][self.current_stop])
                if current_package_distance < nearest_package['distance']
and self.packages[j].delivery_status == 'en route' and
self.packages[j].delivery_deadline != 'EOD':
                    nearest_package['address'] = current_package_address
                    nearest_package['distance'] = current_package_distance
                j = j + 1
            i = len(self.packages)

   return nearest_package
```

Methods:

- Dropping off the package: **O(n)**. The package list is iterated until the package that matches the address is found.
- Calculating the next stop: **O(n)**. The package list is iterated from beginning to end until the one with the shortest distance is found. This requires checking all the packages.

- Calculating each package distance: **O(1)**. The next package's address and the truck's current stop are used as keys to grab the distance values from the distance hash. The lookup time is instant.
  Total space-time complexity: **O(n^2)**

C_4

With space time complexity of many sections being **O(n^2)**, the program will have performance dips with large datasets. An advantage of the program is the lookup time with a time complexity of **O(1)**, which means that the speed of grabbing and comparing distances will remain the same regardless of the size of the dataset.

C_5

The design could be polished, but overall, it is easily maintained. The program has separation of concerns with 'Truck', 'Packages', 'Interface', and 'Distances' entities. The truck packages are loosely coupled, and making changes to one won't interfere with the other. The program is also modular, as each major section of code is properly separated and contained.

C_6

The strengths of the hash table are that the design is simple. The structure of the hash map doesn't physically change (i.e., nothing is rearranged, only the delivery status values are updated), this makes tracking the packages simple. Lookup time is fairly fast, with a hashing algorithm placing packages into buckets (limiting the size of the first layer of the hash table). A disadvantage is that the buckets themselves are lists and require linear traversal to reach the correct package.

C_7

Using the package id as a key works in that each ID is unique. This makes modifying the package's status - as well as other attributes - simple and prevents data corruption.

Task II

Task F

F_1

Strengths of the algorithm:

1. The lookup time when calculating the next nearest address is O(1), meaning that the algorithm's execution time will be constant regardless of the dataset's size.
2. The algorithm is modular and loosely coupled. Truck entities operate independently, and their attributes can be altered without one affecting the other.

F_2

The algorithm does meet all of the requirements stated in the project scenario - this can be verified in the attached screenshots from Tasks D and E. The requirements met are: all priority packages are delivered on time, all delayed packages are picked up at the correct time, all packages that are assigned to specific trucks are loaded correctly, and the mileage requirements are met (< 140 miles total trip).

F_3

Two other named algorithms that could've achieved similar results are:

1. Dijkstra's algorithm: This could've planned the optimal route by not only finding the nearest immediate package from the current stop (as in my greedy algorithm) but by building a shortest global path through exploring all possible travel paths to each stop and repeatedly selecting the shortest one. A drawback is that it would've incurred more overhead, computing all shortest paths from the hub to every stop in our route prior to actually deciding on a route and delivering the packages. There's also the misuse of the algorithm itself, whose purpose is to find the shortest path from an origin to a destination. This isn't the goal of this project, which is to minimize mileage while still visiting all stops.

2. Bellman-Ford algorithm: Like Dijkstra's algorithm, this algorithm also builds a shortest route by repeatedly computing the shortest path to each stop from the hub. The difference is that it takes a brute force approach by exploring all possible routes to every single stop, exhausting all options (i.e.: relaxing all edges V - 1). Dijkstra chooses to explore paths based on the current minimal travel time, established by previously relaxed edges, which can lead to unexplored paths (e.g, If a route splits between two nodes, we'll always choose the node that has the shortest path. If the other node connects to a node that the chosen node never reaches, this would lead to an unexplored path (O(E*logV) time complexity) (Geeks For Geeks, Dijkstra's Algorithm to find Shortest Paths from a Source to all). Bellman-Ford would guarantee finding the optimal solution, but have a very large time complexity at O(V*E) (Geeks For Geeks, Bellman-Ford Algorithm). It also can't be fully utilized as 'negative' edges wouldn't exist in the context of an itinerary where negative mileage isn't possible.

Task G

What I would do differently is I would make the program more scalable by removing hard coded flags (e.g., "If package must be loaded onto truck 2") and replace them with dynamic flags to handle more versatile constraints. These conditionals would involve parsing the package note strings.
I'd also make better use of the package hash table and have it interact directly with the distance hash table, rather than instantiating 'Truck' entities as intermediaries. If I continued using truck

entities, I'd make them dictionaries instead of arrays for faster package look up time during delivery.

Task H

The implemented data structures do meet all specified project requirements. A custom hash table is initialized with methods to add package data, and packages are ordered based on a hashing alrogirthm; to permit faster lookup time (Ologn). The hash table also includes a look up function that takes a package ID and returns the correct package object. No external libraries have been used in this project.

H_1

The two other data structures that could've met similar requirements are:

1. A two-dimensional array: the item in the '0' index of each subarray could've acted as a hash key, and the item in the '1' index could've been the list of packages. Lookup time would've been affected, bucket key lookup would've worsened to O(n) from O(1) since array search methods are always in linear time.
2. A min heap tree: If this data structure were used, the hash table could've interacted directly with the distance hash to find the next nearest package. Whenever the next destination is determined, the heap tree could self-adjust by moving all package objects top-to-bottom, with the nearest packages at the top.

Sources:

- *1:https://www.geeksforgeeks.org/dsa/dijkstras-shortest-path-algorithm-greedy-algo-7/
- *2:https://www.geeksforgeeks.org/dsa/bellman-ford-algorithm-dp-23/#principle-of-relaxation-of-edges