

C950 Task-2 WG
UPS Write-Up

C950 Task-2 WGUPS Write-Up

(Task-2: The implementation phase of the WGUPS Routing Program).

(Zip your source code and upload it with this file)

Logan Tackett

ID #012308366

WGU Email: ltacke12@wgu.edu

11/12/2024

C950 Data Structures and Algorithms II

A. Hash Table

```
hashTable.py > ...
1 from package import Package
2
3
4 class HashTable:
5     def __init__(self, initial_size):
6         self.table = [None] * initial_size
7         self.size = initial_size
8         self.count = 0
9
10
11     def hashkey(self, id):
12         return int(id) - 1
13
14
15     def resize(self):
16         extension = [None] * self.size
17         self.table.extend(extension)
18         self.size *= 2
19
20
21     #assumes status to be default at hub
22     def insertPackage(self, package):
23         key = self.hashkey(package.id)
24
25         #If key outside range of table
26         if key > self.size - 1:
27             self.resize()
28             return self.insertPackage(package)
29
30         #If key already in table, dont increment count and update info
31         #If key not in table, increment count
32         if self.table[key] == None:
33             self.count += 1
34         self.table[key] = package
35
36         return True
37
38
39     #insert package by information instead of object.
40     def insertPackageInfo(self, id, address, deadline, city, zip, weight, note=None):
41         return self.insertPackage(Package(id, address, deadline, city, zip, weight, note))
42
43
44     #Lookup function
45     def lookup(self, id):
46         key = self.hashkey(id)
47         if key < 0 or key > self.size - 1:
48             return False
49         else:
50             return self.table[key]
```

B. Look-Up Functions

```
#Lookup function
def lookup(self,id):
    key = self.hashkey(id)
    if key < 0 or key > self.size - 1:
        return False
    else:
        return self.table[key]
```

C. Original Code

Truck Class:

Package loading functions

```
#packages are stored by id, and their information will be pulled from the hash table
def loadPackage(self,package):
    if package.id in self.packages:
        return False
    else:
        if self.package_count == 16:
            return False
        #update the packages departure time to be the trucks departure time
        package.departure_time = self.departure_time
        #add package id to correct list
        if package.note == "Wrong Address Listed":
            self.packages_no_address.append(package.id)
        else:
            self.packages.append(package.id)
        self.package_count += 1
        return True

def loadPackageById(self,id):
    if id in self.packages:
        return False
    else:
        if self.package_count == 16:
            return False
        package = self.hashTable.lookup(id)
        package.departure_time = self.departure_time
        if package.note == "Wrong Address Listed":
            self.packages_no_address.append(package.id)
        else:
            self.packages.append(package.id)
        self.package_count += 1
        return True
```

Sort packages into optimal delivery order:

```
#first sorts packages by deadline to ensure that packages with earlier deadlines get delivered first, O(nlogn).
#then sorts each group of packages with the same deadline using nearest neighbor, worst case O(n^2), O(n) best case,
# such that the start of each group is the closest location to the end of the previous group.
#the destination array stores a list of all destinations such that the index of the destination is the vertex label
# in the adjacency matrix
def sortPackages(self, initial=0):
    #sorts packages first by deadline, earlier to latest
    merge_sort(self.packages, 0, len(self.packages)-1, self.id_to_deadline)

    l = 0
    r = 0

    start = initial
    while l < len(self.packages) - 1:
        #adjusting r pointer such that l and r pointer group items with same deadline
        if r < len(self.packages) and self.id_to_deadline(self.packages[l]) == self.id_to_deadline(self.packages[r]):
            r += 1
        else:
            #slice that portion of the packages array
            sliced = self.packages[l:r]
            #sort it by nearest neighbor
            optimal = nearestNeighbor(start, sliced, self.id_to_adj_label, self.adjacency_matrix)
            #copy the sorted items back into the original array
            for i in range(len(optimal)):
                self.packages[l+i] = optimal[i]
            #adjust pointers
            l = r
            start = self.id_to_adj_label(self.packages[r-1])
```

Deliver Packages:

```

#defaults initial location and time to HUB and self.departure_time
#O(n) best case O(n^2)
def deliverPackages(self,init_location="HUB",init_time=None):
    #sort packages for optimal delivery order
    self.sortPackages(self.address_to_label(init_location))
    #preparing to deliver in order using array pop
    self.packages.reverse()

    #set current location to the hub
    curr_location = init_location
    #set curr time
    if init_time == None:
        curr_time = self.departure_time
    else:
        curr_time = init_time
    #boolean that tracks if we have received correct addresses for known problem packages
    correct_info_received = False
    #main loop
    while len(self.packages) > 0:
        #we have received the correct address information for packages that had wrong address
        if curr_time > self.address_update_time and correct_info_received == False:
            #add corrected packages to package array
            self.packages.extend(self.packages_no_address)
            self.packages_no_address.clear()
            #sort them to reoptimize delivery order with new information
            self.sortPackages(self.address_to_label(curr_location))
            #reverse as the sort undoes the correct ordering for array.pop() usage
            self.packages.reverse()
            #set correct_info_received to True so this optimization only happens once.
            correct_info_received = True

        #get package information
        delivering_id = self.packages.pop()
        delivering_package = self.hashTable.lookup(delivering_id)
        to_location = delivering_package.address

        #get address labels in adjacency matrix and get distance
        curr_location_label = self.address_to_label(curr_location)
        to_location_label = self.address_to_label(to_location)
        distance = self.adjacency_matrix[curr_location_label][to_location_label]

        #update truck milage
        self.miles += distance

        #18 mph average
        time_to = datetime.timedelta(hours=(distance/18))

        #calculate delivered time
        delivery_time = curr_time + time_to

        #update package information
        delivering_package.arrival_time = delivery_time

        #update curr pointers for next loop
        curr_time = delivery_time
        curr_location = to_location

```

Deliver packages continued:

```
#all package delivery steps done for this package, decrement self.package_count
#add package id to delivered
self.package_count -= 1
self.delivered.append(delivering_id)

#while loop over, check to make sure there aren't any packages with late address corrections
#if the below condition returns true, then we have delivered all packages except for those
#with incorrect addresses, delivery truck will sit still until that time, and continue delivering
if self.package_count != 0:
    self.packages.extend(self.packages_no_address)
    self.packages_no_address.clear()
    self.break_start = curr_time
    self.deliverPackages(curr_location, self.address_update_time)
else:
    #all packages are delivered, time to return to HUB
    curr_location_label = self.address_to_label(curr_location)
    dist_to_hub = self.adjacency_matrix[0][curr_location_label]
    self.miles += dist_to_hub
    time_to_hub = datetime.timedelta(hours=(dist_to_hub/18))
    self.return_time = curr_time + time_to_hub
```

Function that loads trucks both manually and heuristically (by deadline):

```
#functions taking packages as input are assumed to be package id's requiring lookup.
#mostly manual loading as packages 13,14,15,16,19,20 must go together in earliest leaving truck.
#O(nlogn) because of merge sorts
def loadTrucks(trucks,packages,hashtable):
    #key_f function to be used
    def id_to_deadline(id):
        package = hashtable.lookup(id)
        return package.deadline

    #addressing packages that must be delivered together
    packages_grouped = [13,14,15,16,19,20]
    for id in packages_grouped:
        packages.remove(id)

    #group package ids by whether they have notes or not
    packages_no_notes = []
    packages_notes = []
    for id in packages:
        package = hashtable.lookup(id)
        if package.note == None:
            packages_no_notes.append(id)
        else:
            packages_notes.append(id)

    #sort them by time that way packages with earlier deadlines are loaded first
    merge_sort(packages_no_notes,0,len(packages_no_notes)-1,id_to_deadline)
    merge_sort(packages_notes,0,len(packages_notes)-1,id_to_deadline)
    #to prepare them to load by array.pop()
    packages_no_notes.reverse()
    packages_notes.reverse()
    #to store packages that have a note of wrong address
    packages_wrong_address = []

    #truck 1 will first get grouped packages
    truck1 = trucks[0]
    while truck1.package_count < 16 and len(packages_grouped) > 0:
        id = packages_grouped.pop()
        truck1.loadPackageById(id)

    #truck 1 will then get all packages with no notes up to 16 packages
    while truck1.package_count < 16 and len(packages_no_notes) > 0:
        id = packages_no_notes.pop()
        truck1.loadPackageById(id)

    #truck 2 will have all packages with notes and more up to 16
    #truck 2 will be waiting for late arrivals to arrive before leaving.
    truck2 = trucks[1]
    while truck2.package_count < 16 and len(packages_notes) > 0:
        id = packages_notes.pop()
        package = hashtable.lookup(id)
        if package.note == "Wrong Address Listed":
            packages_wrong_address.append(id)
        else:
            truck2.loadPackageById(id)
```

Load trucks continued:

```
#load any remaining packages onto the trucks now
truck3 = trucks[2]
while truck1.package_count < 16 and len(packages_no_notes) > 0:
    id = packages_no_notes.pop()
    truck1.loadPackageById(id)
while truck1.package_count < 16 and len(packages_notes) > 0:
    id = packages_notes.pop()
    truck1.loadPackageById(id)
while truck2.package_count < 16 and len(packages_no_notes) > 0:
    id = packages_no_notes.pop()
    truck2.loadPackageById(id)
while truck2.package_count < 16 and len(packages_notes) > 0:
    id = packages_notes.pop()
    truck2.loadPackageById(id)
while truck3.package_count < 16 and len(packages_no_notes) > 0:
    id = packages_no_notes.pop()
    truck3.loadPackageById(id)
while truck3.package_count < 16 and len(packages_notes) > 0:
    id = packages_notes.pop()
    truck3.loadPackageById(id)

#we will return these to be loaded onto a truck after sorting, that way it is at the end of the
#list of packages (to be delivered last). They will be resorted once the new address information
#is gotten
return packages_wrong_address
```

Load incorrect address packages:

```
#used to load packages who's addresses need correction to the packages_no_address list
def loadWrongAddressPackages(trucks,extras):
    truck1 = trucks[0]
    truck2 = trucks[1]
    truck3 = trucks[2]
    #try truck 1 first
    while truck1.package_count < 16 and len(extras) > 0:
        id = extras.pop()
        truck1.loadPackageById(id)
        #truck stores when it will be receiving the address for use in delivery function
        truck1.address_update_time = datetime.datetime(2024,11,10,10,20)
    while truck2.package_count < 16 and len(extras) > 0:
        id = extras.pop()
        truck2.loadPackageById(id)
        truck2.address_update_time = datetime.datetime(2024,11,10,10,20)
    while truck3.package_count < 16 and len(extras) > 0:
        id = extras.pop()
        truck3.loadPackageById(id)
        truck3.address_update_time = datetime.datetime(2024,11,10,10,20)
```


C1. Identification Information

```
main.py > ...
1 | # Student ID #012308366
2 | # Logan Tackett
3 |
```

```
#import csv data
csv_packages = loadPackageData('WGUPS Package File - Sheet1.csv')
csv_packages_ids = []
for package in csv_packages:
    csv_packages_ids.append(package.id)

#import distance table
destination_array, matrix = loadDistanceData('WGUPS Distance Table - Sheet1.csv')
#populate empty cells of distance table to make it symmetric
populateUpperTriange(matrix)
#convert all entries of distance table to float values
floatify(matrix)
#convert distance table to all-paths shortest distance table
FloydWarshallAllPairsShortestPath(matrix)

#create package hash table
hash_table = HashTable(10)
#insert all packages into hash table
for package in csv_packages:
    hash_table.insertPackage(package)

#create trucks 1, 2, and 3. Truck 1 leaves at 8:00, Truck 2 leaves at 9:05, and truck 3 leaves when one of the two trucks returns.
truck1 = Truck(0,datetime.datetime(2024,11,10,8,0),destination_array,matrix,hash_table)
truck2 = Truck(1,datetime.datetime(2024,11,10,9,5),destination_array,matrix,hash_table)
truck3 = Truck(2,datetime.datetime(2024,11,10,23,59),destination_array,matrix,hash_table)

Fleet = [truck1,truck2,truck3]

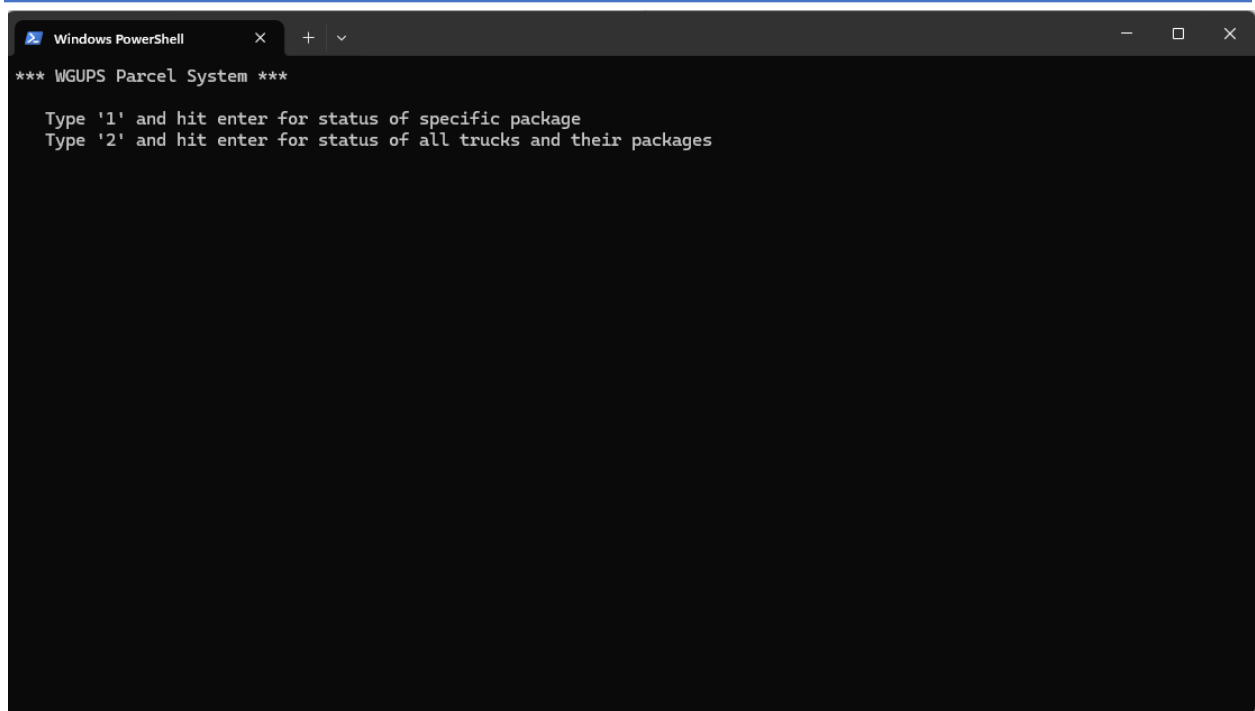
#load packages onto trucks, returns list of packages with incorrect addresses
wrong_addresses = loadTrucks(Fleet,csv_packages_ids,hash_table)

#loads packages with wrong addresses onto trucks
loadWrongAddressPackages(Fleet,wrong_addresses)

#delivers packages for trucks 1 and 2
truck1.deliverPackages()
truck2.deliverPackages()
#truck three starts delivery when one of the two trucks returns
if truck1.return_time < truck2.return_time:
    truck3.updateDeparture(truck1.return_time)
else:
    truck3.updateDeparture(truck2.return_time)
truck3.deliverPackages()
```

C2. Process and Flow Comments

D. Interface



```
Windows PowerShell
*** WGUPS Parcel System ***

Type '1' and hit enter for status of specific package
Type '2' and hit enter for status of all trucks and their packages
```

D1. First Status Check

```
Windows PowerShell
*** Delivery Status At Time: 9:00 ***

All Trucks Current Milage: 18.0

Truck 0 status: Delivering
Current Milage: 18.0

Truck 0 Packages:
ID: 15 Status: Delivered at time: 8:11 (On Time)
ID: 34 Status: Delivered at time: 8:11 (On Time)
ID: 16 Status: Delivered at time: 8:11 (On Time)
ID: 14 Status: Delivered at time: 8:18 (On Time)
ID: 20 Status: Delivered at time: 8:28 (On Time)
ID: 31 Status: Delivered at time: 8:33 (On Time)
ID: 40 Status: Delivered at time: 8:39 (On Time)
ID: 1 Status: Delivered at time: 8:42 (On Time)
ID: 29 Status: Delivered at time: 8:52 (On Time)
ID: 37 Status: En Route
ID: 30 Status: En Route
ID: 13 Status: En Route
ID: 39 Status: En Route
ID: 35 Status: En Route
ID: 19 Status: En Route
ID: 33 Status: En Route

Truck 1 status: At HUB
Current Milage: 0.0

Truck 1 Packages:
ID: 25 Status: HUB
ID: 6 Status: HUB
ID: 17 Status: HUB
ID: 32 Status: HUB
ID: 21 Status: HUB
ID: 28 Status: HUB
ID: 12 Status: HUB
ID: 36 Status: HUB
ID: 27 Status: HUB
ID: 3 Status: HUB
ID: 38 Status: HUB
ID: 24 Status: HUB
ID: 26 Status: HUB
ID: 22 Status: HUB
ID: 23 Status: HUB
ID: 18 Status: HUB
```

```
Truck 2 status: At HUB
Current Milage: 0.0

Truck 2 Packages:
ID: 2 Status: HUB
ID: 7 Status: HUB
ID: 10 Status: HUB
ID: 5 Status: HUB
ID: 9 Status: HUB
ID: 8 Status: HUB
ID: 4 Status: HUB
ID: 11 Status: HUB

Hit ENTER to return to home screen.
```

D2. Second Status Check

```
*** Delivery Status At Time: 10:00 ***  
  
All Trucks Current Milage: 52.5  
  
Truck 0 status: Delivering  
Current Milage: 36.0  
  
Truck 0 Packages:  
ID: 15 Status: Delivered at time: 8:11 (On Time)  
ID: 34 Status: Delivered at time: 8:11 (On Time)  
ID: 16 Status: Delivered at time: 8:11 (On Time)  
ID: 14 Status: Delivered at time: 8:18 (On Time)  
ID: 20 Status: Delivered at time: 8:28 (On Time)  
ID: 31 Status: Delivered at time: 8:33 (On Time)  
ID: 40 Status: Delivered at time: 8:39 (On Time)  
ID: 1 Status: Delivered at time: 8:42 (On Time)  
ID: 29 Status: Delivered at time: 8:52 (On Time)  
ID: 37 Status: Delivered at time: 9:06 (On Time)  
ID: 30 Status: Delivered at time: 9:09 (On Time)  
ID: 13 Status: Delivered at time: 9:23 (On Time)  
ID: 39 Status: Delivered at time: 9:23 (On Time)  
ID: 35 Status: Delivered at time: 9:29 (On Time)  
ID: 19 Status: Delivered at time: 9:49 (On Time)  
ID: 33 Status: Delivered at time: 9:57 (On Time)  
  
Truck 1 status: Delivering  
Current Milage: 16.5  
  
Truck 1 Packages:  
ID: 25 Status: Delivered at time: 9:13 (On Time)  
ID: 6 Status: Delivered at time: 9:36 (On Time)  
ID: 17 Status: Delivered at time: 9:41 (On Time)  
ID: 32 Status: Delivered at time: 9:43 (On Time)  
ID: 21 Status: Delivered at time: 9:48 (On Time)  
ID: 28 Status: Delivered at time: 9:52 (On Time)  
ID: 12 Status: En Route  
ID: 36 Status: En Route  
ID: 27 Status: En Route  
ID: 3 Status: En Route  
ID: 38 Status: En Route  
ID: 24 Status: En Route  
ID: 26 Status: En Route  
ID: 22 Status: En Route  
ID: 23 Status: En Route  
ID: 18 Status: En Route
```

```
Truck 2 status: At HUB  
Current Milage: 0.0  
  
Truck 2 Packages:  
ID: 2 Status: HUB  
ID: 7 Status: HUB  
ID: 10 Status: HUB  
ID: 5 Status: HUB  
ID: 9 Status: HUB  
ID: 8 Status: HUB  
ID: 4 Status: HUB  
ID: 11 Status: HUB  
  
Hit ENTER to return to home screen.
```

D3. Third Status Check

```
*** Delivery Status At Time: 13:00 ***
```

```
All Trucks Current Milage: 119.6
```

```
Truck 0 status: At HUB
```

```
Current Milage: 38.1
```

```
Truck 0 Packages:
```

```
ID: 15 Status: Delivered at time: 8:11 (On Time)
ID: 34 Status: Delivered at time: 8:11 (On Time)
ID: 16 Status: Delivered at time: 8:11 (On Time)
ID: 14 Status: Delivered at time: 8:18 (On Time)
ID: 20 Status: Delivered at time: 8:28 (On Time)
ID: 31 Status: Delivered at time: 8:33 (On Time)
ID: 40 Status: Delivered at time: 8:39 (On Time)
ID: 1 Status: Delivered at time: 8:42 (On Time)
ID: 29 Status: Delivered at time: 8:52 (On Time)
ID: 37 Status: Delivered at time: 9:06 (On Time)
ID: 30 Status: Delivered at time: 9:09 (On Time)
ID: 13 Status: Delivered at time: 9:23 (On Time)
ID: 39 Status: Delivered at time: 9:23 (On Time)
ID: 35 Status: Delivered at time: 9:29 (On Time)
ID: 19 Status: Delivered at time: 9:49 (On Time)
ID: 33 Status: Delivered at time: 9:57 (On Time)
```

```
Truck 1 status: At HUB
```

```
Current Milage: 52.8
```

```
Truck 1 Packages:
```

```
ID: 25 Status: Delivered at time: 9:13 (On Time)
ID: 6 Status: Delivered at time: 9:36 (On Time)
ID: 17 Status: Delivered at time: 9:41 (On Time)
ID: 32 Status: Delivered at time: 9:43 (On Time)
ID: 21 Status: Delivered at time: 9:48 (On Time)
ID: 28 Status: Delivered at time: 9:52 (On Time)
ID: 12 Status: Delivered at time: 10:02 (On Time)
ID: 36 Status: Delivered at time: 10:12 (On Time)
ID: 27 Status: Delivered at time: 10:21 (On Time)
ID: 3 Status: Delivered at time: 10:37 (On Time)
ID: 38 Status: Delivered at time: 10:41 (On Time)
ID: 24 Status: Delivered at time: 11:03 (On Time)
ID: 26 Status: Delivered at time: 11:09 (On Time)
ID: 22 Status: Delivered at time: 11:13 (On Time)
ID: 23 Status: Delivered at time: 11:35 (On Time)
ID: 18 Status: Delivered at time: 11:37 (On Time)
```

```
Truck 2 status: At HUB
```

```
Current Milage: 28.7
```

```
Truck 2 Packages:
```

```
ID: 2 Status: Delivered at time: 10:16 (On Time)
ID: 7 Status: Delivered at time: 10:21 (On Time)
ID: 10 Status: Delivered at time: 10:31 (On Time)
ID: 5 Status: Delivered at time: 10:37 (On Time)
ID: 9 Status: Delivered at time: 10:40 (On Time)
ID: 8 Status: Delivered at time: 10:40 (On Time)
ID: 4 Status: Delivered at time: 10:58 (On Time)
ID: 11 Status: Delivered at time: 11:21 (On Time)
```

```
Hit ENTER to return to home screen.
```

E. Screenshot of Code Execution

```

1  # Student ID #012308366
2  # Logan Tackett
3
4  from util import populateUpperTriange, FloydMarshallAllPairsShortestPath, merge_sort, floatify
5  from hashTable import HashTable
6  from postal import loadTrucks, printAllTrucksStatus, loadWrongAddressPackages, loadPackageData, loadDistanceData
7  from package import Package
8  from truck import Truck
9  import datetime
10 import os
11
12 #import csv data
13 csv_packages = loadPackageData('WGUPS Package File - Sheet1.csv')
14 csv_packages_ids = []
15 for package in csv_packages:
16     csv_packages_ids.append(package.id)
17
18 #import distance table
19 destination_array, matrix = loadDistanceData('WGUPS Distance Table - Sheet1.csv')
20 #populate empty cells of distance table to make it symmetric
21 populateUpperTriange(matrix)
22 #convert all entries of distance table to float values
23 floatify(matrix)
24 #convert distance table to all-paths shortest distance table
25 FloydMarshallAllPairsShortestPath(matrix)
26
27 #create package hash table
28 hash_table = HashTable(10)
29 #insert all packages into hash table
30 for package in csv_packages:
31     hash_table.insertPackage(package)
32
33 #create trucks 1, 2, and 3. Truck 1 leaves at 8:00, Truck 2 leaves at 9:05, and truck 3 leaves when one of the two trucks returns.
34 truck1 = Truck(0,datetime.datetime(2024,11,10,8,0),destination_array,matrix,hash_table)
35 truck2 = Truck(1,datetime.datetime(2024,11,10,9,5),destination_array,matrix,hash_table)
36 truck3 = Truck(2,datetime.datetime(2024,11,10,23,59),destination_array,matrix,hash_table)
37
38 Fleet = [truck1,truck2,truck3]
39
40 #load packages onto trucks, returns list of packages with incorrect addresses
41 wrong_addresses = loadTrucks(Fleet,csv_packages_ids,hash_table)
42
43 #loads packages with wrong addresses onto trucks
44 loadWrongAddressPackages(Fleet.wrong_addresses)

```

PROBLEMS OUTPUT DEBUG CONSOLE **TERMINAL** PORTS

```

All Trucks Current Milage: 119.6

Truck 0 status: At HUB
Current Milage: 38.1

Truck 0 Packages:
ID: 15 Status: Delivered at time: 8:11 (On Time)
ID: 34 Status: Delivered at time: 8:11 (On Time)
ID: 16 Status: Delivered at time: 8:11 (On Time)
ID: 14 Status: Delivered at time: 8:18 (On Time)
ID: 20 Status: Delivered at time: 8:28 (On Time)
ID: 31 Status: Delivered at time: 8:33 (On Time)
ID: 40 Status: Delivered at time: 8:39 (On Time)

```

F1. Strengths of the Chosen Algorithm

I used a greedy nearest neighbor algorithm applied to an optimized distance table using the Floyd Warshall algorithm. The main strength of the use of nearest neighbor is the algorithm's simplicity. It runs in $O(n^2)$ time and does a close to optimal job of delivering the packages on the truck. Compare this to a DFS algorithm, which could give a more optimal or even perfect solution at the cost of runtime. The main strength of using Floyd Warshall is the easy optimization it provides. It runs in $O(n^3)$ time, which for this problem is efficient as it only runs once, and the outcome of running it is a more optimal distance table that can be used in all future processes of the program. I also used merge sort to sort packages by deadline. This was simply to make sure that packages got to their destination on time. The strength of sorting the packages this way is, again, simplicity and an efficient runtime of $O(n \log n)$. Using heuristics like sorting by destination time and then by nearest neighbor greatly simplifies the problem and keeps time and space complexities low when compared to more optimal methods like DFS.

F2. Verification of Algorithm

The algorithm provided above, plus some edge case handlers, satisfies all requirements in the scenario. All packages are delivered by their deadline, and delayed packages don't leave until 9:05; the package with the missing address is loaded onto a truck as early as possible but isn't delivered until the new address is received, to which the delivery order is resorted such that the route stays optimal even with the new address; packages that must

be delivered on the same truck are delivered on the same truck, and packages that must be on truck 2 are all on truck 2. Total truck mileage after all deliveries, including the drive back to the HUB, is 119.6 miles, which is less than 140 miles. Trucks leave no earlier than 8:00 AM and only two trucks are delivering at a time.

F3. Other Possible Algorithms

Two other algorithms that could have been used to solve this problem while meeting the requirements are DFS (Depth-First-Search) and 2-Opt.

F3a. Algorithm Differences

A DFS approach would check all possible ways to deliver the packages, excluding pruned orderings, to see which one is most optimal while satisfying all the conditions. This is very different from nearest neighbor, which is a greedy algorithm that produces a near-optimal result without checking any particular package permutation. The scale of this particular problem is probably small enough that DFS would give a more optimal answer in a tolerable amount of time with the correct optimizations, but if there were to be more locations and packages, DFS could take a very long time. With the optimizations available to us for this problem, the runtime of a DFS approach would still be $O(n!)$. One could use a depth-limited search to calculate a close-to-optimal next package to deliver without going through all permutations of packages, but this is essentially what nearest neighbor is doing with a depth of 1. While this method could yield a more optimal result whilst having a polynomial runtime, it would still be much slower than my algorithm of choice. Specifically, the runtime would be $O(n \text{ choose } l)$, where l is the fixed depth.

2-Opt is a heuristic-based algorithm that finds paths that are close to optimal ones by making sure the path doesn't intersect itself. This could very easily replace the role of nearest neighbor with little refactoring and would have a similar runtime complexity. The question of which one will provide a more optimal result is hard to answer. The path generated by nearest neighbor may intersect itself, which is nonoptimal, but each movement in the path from node A to node B is done such that node B is the closest unvisited node to node A, which is optimal. The path generated by 2-opt would result in a similarly optimal path as the path would never intersect itself, but there is no guarantee that a connection is the shortest one that could have been made.

G. Different Approach

If I had to redo this project, I would load packages using a more sophisticated approach. While it would be more complicated to do so, the most effective way to create a route that is close to the optimal one is to load the trucks by package proximity. Yes, greedy algorithms applied to a set of packages will produce a more optimal result, but this will never come close to loading the trucks such that packages are as close as possible. Packages with early deadlines that are ready to go on time should simply go on the first truck to leave; this doesn't require a sophisticated algorithm. The same applies to packages with an early deadline that arrive late. For packages that have no requirements, one could first find the two packages that are farthest from each other and load them onto two different trucks. Then, the trucks would take turns loading the next closest available package for each truck. If a majority of the packages had an EOD deadline, this optimization would make a large difference in mileage and delivery time

at the same cost as the current implementation ($O(n^2)$). I wouldn't change anything about the way I deliver packages, as packages are simply delivered in the order that they are listed. This ordering in the list is optimized both before and during delivery as new information comes in.

H. Verification of Data Structure

The data structure used to store package data is a hash table. The hash table uses no additional libraries or classes to store packages and does so using the package ID and a package object. The hash table has an insert function that takes an ID and all of the package parameters as input and stores those parameters in the hash table. The hash table also has a lookup function that allows me to pull all package data using the ID alone.

H1. Other Data Structures

Two other data structures that could have been used to store package data are a list (array) or a BST.

H1a. Data Structure Differences

A list implementation would simply append packages to a list to store their data. While lists and hash tables are both implemented using arrays, the key difference is that in a hash table, the index an item is stored at is dependent on a key, for example, an array containing packages such that the index of the package in the array is the package id is a hash table (this is how I implemented my hash table) while a list would simply append new packages to the end of the array assuming it is not already in the array.

This means that the index of packages in a list would be independent of any of the parameters of the package, and lookups would take $O(n)$ time,

which is much less efficient than a hash table that has $O(1)$ lookups. A BST implementation is similar to a list in that the location of the package in the tree is only partially dependent on the ID of the package, as insertion order now affects the location of packages in the tree. In the worst case, a lookup would be $O(n)$, and in the best case, the lookup would be $O(\log n)$, which is better than a list but worse than a hash table.

I. Sources

Lysecky, R., & Vahid, F. (2018, June). *C950: Data Structures and Algorithms II*. zyBooks.

Retrieved November 12, 2024, from

<https://learn.zybooks.com/zybook/WGUC950AY20182019/>

CSV - csv file reading and writing. Python documentation. (n.d.).

<https://docs.python.org/3/library/csv.html>

Retrieved November 12, 2024 from

<https://docs.python.org/3/library/csv.html>

Wikimedia Foundation. (2024, August 15). *2-opt*. Wikipedia.

<https://en.wikipedia.org/wiki/2-opt>

Retrieved November 12, 2024 from

<https://en.wikipedia.org/wiki/2-opt>

Nearest neighbour algorithm (2024) *Wikipedia*. Available at:
https://en.wikipedia.org/wiki/Nearest_neighbour_algorithm.

Retrieved November 12, 2024 from

https://en.wikipedia.org/wiki/Nearest_neighbour_algorithm

J. Professional Communication
