**A: Identify a named self-adjusting algorithm (e.g., “Nearest Neighbor algorithm,” “Greedy algorithm”) that you used to create your program to deliver the packages.**

The algorithm that I chose for this project is the Greedy algorithm (Lysecky, C950: Data Structures and Algorithms II, 2018). The first location from the hub is the one closest. Then from there, all distances are parsed until the closest position from the new current position is found. I had to make some adjustments because of the requirements of the packages such as deadlines and special notes. I accomplished this using a “white” list and a “black” list.

For the whitelist, I added the priority items and truck-specific items. The algorithm was only allowed to select from the whitelist pool, and I was able to generate a route. After all whitelist items were added, I then selected a blacklist which included all the items that couldn’t go on that specific truck. I combined these two to make a route. I also had to include stopping back at the hub for additional packages.

**B: Write an overview of your program, in which you do the following:**

**1: Explain the algorithm’s logic using pseudocode.**

All Package Numbers = A list of all package numbers

Package Black List = Packages that have already been assigned

White List = Priority locations

Black List = Locations that should not be visited

Visited List = Locations already visited. Can be adjusted to a whitelist, which adds all packages instead of whitelist packages forcing the algorithm to choose only whitelist locations, or to a blacklist which adds items with different requirements whose locations shouldn’t be selected

Temporary location = represents starting location which would be 0. This is updated throughout the algorithm to show the current location

Input – Starting Location? (Inputting 0 will find the closest location to the hub)

Visited List – Append input

Low Distance = 100

Input = X

Input = Y

While X > 0

X -= 1

If X is found in visited list, skip

If distance between input and new, lower X location < Low Distance

Low Distance = New lower distance

Location number = X

X = Input again

While Y < Size of All Location Numbers – 1 (size is 27 but 0 is a location number)

Y += 1

If Y is found in visited list, skip

If distance between input and new, lower Y location < Low Distance

Low Distance = New lower distance

Location number = Y

Return Lowest distance and either X or Y (whichever is lower)

Determine all packages at location by searching

If package is in package blacklist, skip

Display location, distance to location, packages at location

Add packages to package blacklist

This algorithm allows me to generate a list of locations to determine a route. I manually load the trucks with packages and route information. The X and Y refer to how I have my distance table set up and represent coordinates in the distance table. This saves on complexity because instead of running through the whole table, only one row and one column is run.

**B2: Describe the programming environment you used to create the Python application.**

This python was created in the community edition of PyCharm as well as Online-Python.com

**B3: Evaluate the space-time complexity of each major segment of the program, and the entire program, using big-O notation.**

|  |  |  |
| --- | --- | --- |
| Creating chaining hash tables and loading data | O(N) | 2 tables must be added (packages and addresses) but the complexity doesn’t change |
| Lookup function | O(N) |  |
| Table functions including search, reverse search, package from location and address from package number | O(N) |  |
| Loading distance data | O(N2) | Distance data is 2 dimensional so a N \* N locations must be added |
| Greedy algorithm to find the closest location. | O(N) | Will only need to run through a column and a row instead of the whole distance table. This makes the complexity 2O(N) which is equal to O(N) |
| Package update | O(N) | Runs through multiple lists and compares to a single number |
| Clock object and functions | O(1) |  |
| Loading trucks, routes, distances from lists | O(1) |  |
| Display all packages function | O(N) |  |
| Display trucks and statuses | O(1) |  |
| Set clock | O(1) |  |
| **TOTAL** | O(N2) | Program gains complexity as number of locations increase. Package number has no impact on complexity |

**B4: Explain the capability of your solution to scale and adapt to a growing number of packages.**

A growing number of packages will not add complexity. An increase in locations will add row data and column data to the distance table, although only half of it is necessary. A solution could be to only read half of the distance table, but my program reads the entire thing.

**B5: Discuss why the software is efficient and easy to maintain.**

The software is efficient because the number of packages will increase much faster than the increase in locations. It is easy to maintain because it is very flexible. A user can choose their own route and can decide which locations they want available for the algorithm to choose from

**B6: Discuss the strengths and weaknesses of the self-adjusting data structures**

**(e.g., the hash table).**

The hash table that I have incorporated into my program works very well because there are multiple items associated with one key and storing those items as an object as the value allows a search for an item in the hash table to be less complex and faster. A weakness of this structure is collisions that can occur when multiple items belong in the same bucket which can increase complexity. I circumvented this issue in my program by using a chaining hash table where each package number is a key in the table, which prevents any collisions (Lysecky, C950: Data Structures and Algorithms II, 2018).

**C: Write an original program to deliver all the packages, meeting all requirements, using the attached supporting documents “Salt Lake City Downtown Map,” “WGUPS Distance Table,” and the “WGUPS Package File.”**

**1. Create an identifying comment within the first line of a file named “main.py” that includes your first name, last name, and student ID.**

**2. Include comments in your code to explain the process and the flow of the program.**

# First Name: Jeff

# Last Name: Fenwick

# Student ID: 001549011

# Update mileage on trucks

import csv

# Hash table class

location\_size = 27

visited\_list = []

class ChainingHashTable:

# Hash table constructor with capacity for 40 packages

def \_\_init\_\_(self, initial\_capacity=40):

self.table = []

for i in range(initial\_capacity):

self.table.append([])

# Hash table insert as well as update function

def insert(self, key, item):

bucket = hash(key) % len(self.table)

bucket\_list = self.table[bucket]

for kv in bucket\_list:

if kv[0] == key:

kv[1] = item

return True

key\_value = [key, item]

bucket\_list.append(key\_value)

return True

def update\_package\_status(self, pnumb, status):

temp = str(myHashPackage.search(pnumb))

temp\_str = temp.split(', ')

package = Package(temp\_str[0], temp\_str[1], temp\_str[2], temp\_str[3], temp\_str[4], temp\_str[5],

temp\_str[6], status)

myHashPackage.insert(pnumb, package)

# Remove item from hash table

def remove(self, key):

bucket = hash(key) % len(self.table)

bucket\_list = self.table[bucket]

for kv in bucket\_list:

if kv[0] == key:

bucket\_list.remove(kv[0], kv[1])

# Input package number and return full string of package information

def search(self, key):

bucket = hash(key) % len(self.table)

bucket\_list = self.table[bucket]

for kv in bucket\_list:

if kv[0] == key:

return kv[1]

return None

# Lookup function with separate UI that allows users to look up a package using different criteria

def lookup(self):

print('\nHow would you like to look up package?')

inp = int(input(

'\n0) Exit: \n1) Number: \n2) Street Address: \n3) Deadline: \n4) City: \n5) Zip code: \n6) Weight: \n'

'7) Status: \n'))

while inp != 0:

if inp == 1:

numbInp = int(input('\nPackage number?: '))

print(myHashPackage.search(numbInp))

return

if inp == 2:

streetInp = input('\nStreet address?: ')

for i in range(len(myHashPackage.table)):

streetStr = str(myHashPackage.search(i + 1))

splitStr = streetStr.split(', ')

if streetInp in splitStr[1]:

print(streetStr)

return

if inp == 3:

deadline\_inp = input('\nDeadline?: ')

for i in range(len(myHashPackage.table)):

deadline\_str = str(myHashPackage.search(i + 1))

splitStr = deadline\_str.split(', ')

if deadline\_inp in splitStr[5]:

print(deadline\_str)

return

if inp == 4:

city\_inp = input('\nCity?: ')

for i in range(len(myHashPackage.table)):

city\_str = str(myHashPackage.search(i + 1))

splitStr = city\_str.split(', ')

if city\_inp in splitStr[2]:

print(city\_str)

return

if inp == 5:

zip\_inp = input('\nZip code?: ')

for i in range(len(myHashPackage.table)):

zip\_str = str(myHashPackage.search(i + 1))

splitStr = zip\_str.split(', ')

if zip\_inp in splitStr[4]:

print(zip\_str)

return

if inp == 6:

weight\_inp = input('\nWeight(kg)?: ')

for i in range(len(myHashPackage.table)):

weight\_str = str(myHashPackage.search(i + 1))

splitStr = weight\_str.split(', ')

if weight\_inp in splitStr[6]:

print(weight\_str)

return

if inp == 7:

status\_inp = input('\nStatus(AT\_HUB, EN\_ROUTE, DELIVERED?: ')

for i in range(len(myHashPackage.table)):

status\_str = str(myHashPackage.search(i + 1))

splitStr = status\_str.split(', ')

if status\_inp in splitStr[7]:

if status\_str == 'DELIVERED':

print('Time Delivered: {}'.format(status\_inp)) # Fill in with time delivered

print(status\_str)

return

return

# Reverse search function for determining the location number based on the address for inputting coordinates into

# distance dictionary

def rev\_search(self, address\_inp):

for i in range(location\_size):

address\_search = str(myHashAddresses.search(i))

ret\_numb = myHashAddresses.table[i]

# Needed because some address data is slightly different between package data and distance data

if address\_inp[0:7] in address\_search:

return int(ret\_numb[0][0])

# Input package number and get back location number

def loc\_from\_pnumb(self, pnumb):

street\_str = str(myHashPackage.search(pnumb))

split\_str = street\_str.split(', ')

return myHashAddresses.rev\_search(split\_str[1])

def pnumb\_from\_loc(self, loc):

temp\_list = []

for i in range(len(myHashPackage.table)):

temp\_address = str(myHashPackage.search(i + 1))

str\_address = temp\_address.split(', ')

if loc == myHashAddresses.rev\_search(str\_address[1]):

temp\_list.append(int(str\_address[0]))

return temp\_list

def add\_from\_pnumb(self, pnumb):

street\_str = str(myHashPackage.search(pnumb))

split\_str = street\_str.split(', ')

return split\_str[1]

# Defines the package type of object

class Package:

def \_\_init\_\_(self, pnumber, street\_address, city, state, zipcode, deadline, weight, status):

self.pnumber = pnumber

self.street\_address = street\_address

self.city = city

self.state = state

self.zipcode = zipcode

self.deadline = deadline

self.weight = weight

self.status = status

# Adds strings so that hash table data is returned instead of storage locations

def \_\_str\_\_(self):

return "%s, %s, %s, %s, %s, %s, %s, %s" % (self.pnumber, self.street\_address, self.city, self.state,

self.zipcode, self.deadline, self.weight, self.status)

# Loads package data and inserts into table myHashPackage

def load\_package\_data(filename):

with open(filename) as WGUPSPackageFile:

package\_data = csv.reader(WGUPSPackageFile, delimiter=',')

for i in range(8):

next(package\_data)

for package in package\_data:

pID = int(package[0])

pstreet\_address = package[1]

pcity = package[2]

pstate = package[3]

pzipcode = package[4]

pdeadline = package[5]

pweight = package[6]

pstatus = 'AT\_HUB'

package = Package(pID, pstreet\_address, pcity, pstate, pzipcode, pdeadline, pweight, pstatus)

myHashPackage.insert(pID, package)

# Loads distance data from file and inserts into dictionary with 2 dimensional keys

def load\_distance\_data(filename):

with open(filename) as WGUPSDistanceTable:

distance\_data = csv.reader(WGUPSDistanceTable, delimiter=',')

# Needed to advance the row count to start at the data portion

for i in range(8):

next(distance\_data)

count = 0

for row in distance\_data:

for i in range(27):

if row[i + 2] == '':

continue

else:

myDistanceData[i, count] = float(row[i + 2])

count += 1

# Populates a hash table with just addresses

def load\_address\_data(filename):

with open(filename) as WGUPSPackageFile:

address\_data = csv.reader(WGUPSPackageFile, delimiter=',')

j = 0

for i in range(8):

next(address\_data)

for package in address\_data:

l\_address = package[0]

location = Address(l\_address)

myHashAddresses.insert(j, location)

j += 1

# Creates the address class so that just address data can be added to a hash table

class Address:

def \_\_init\_\_(self, loc\_address):

self.loc\_address = loc\_address

def \_\_str\_\_(self):

return "%s" % self.loc\_address

# Returns a distance given two location numbers

def dist\_locations(inp\_loc1, inp\_loc2):

# Only half of the distance table is populated to avoid redundancy, this ensures that only half can be selected

if inp\_loc1 > inp\_loc2:

inp\_temp = inp\_loc1

inp\_loc1 = inp\_loc2

inp\_loc2 = inp\_temp

return round(float(myDistanceData[(inp\_loc1, inp\_loc2)]), 1)

def closest\_location(c\_loc):

loc\_number = 0

loc = int(c\_loc)

low\_dist = 100.0

x = loc

y = loc

while x > 0:

x -= 1

if x in visited\_list:

continue

if dist\_locations(x, y) < low\_dist:

low\_dist = dist\_locations(x, y)

loc\_number = x

x = loc

while y < location\_size - 1:

y += 1

if y in visited\_list:

continue

if dist\_locations(x, y) < low\_dist:

low\_dist = dist\_locations(x, y)

loc\_number = y

return '{}, {}'.format(low\_dist, loc\_number)

def package\_update():

# Loads truck 3, adds route location, changes status of packages, and adds distance after truck 2 returns to hub

for key, value in truck\_list[1].route.items():

if key == 0 and total\_miles\_traveled() > value:

for x in truck3\_packages:

truck\_list[2].load\_truck(x)

myHashPackage.update\_package\_status(x, 'EN\_ROUTE / TRUCK\_3')

for x in truck3\_route:

truck\_list[2].add\_route\_locations(x)

cur\_loc = 0

for keys in truck\_list[2].route:

truck\_list[2].route[keys] = dist\_locations(cur\_loc, keys) + truck\_list[1].route[0]

cur\_loc = keys

# Reloads truck 1 with the second set of packages as well as updating the status

for key, value in truck\_list[0].route.items():

if key == 0:

if total\_miles\_traveled() >= value:

for x in truck1\_reload\_packages:

truck\_list[0].load\_truck(x)

myHashPackage.update\_package\_status(x, 'EN\_ROUTE / TRUCK\_1')

# Changes status for and removes packages that have been delivered by truck 1

for key, value in truck\_list[0].route.items():

if total\_miles\_traveled() >= value:

for x in myHashPackage.pnumb\_from\_loc(key):

if x in truck\_list[0].packages:

myHashPackage.update\_package\_status(x, 'Delivered at {} by Truck 1'

.format(clock.show\_time\_delivered(value)))

truck\_list[0].packages.remove(x)

# Changes status for and removes packages that have been delivered by truck 2

for key, value in truck\_list[1].route.items():

if total\_miles\_traveled() >= value:

for x in myHashPackage.pnumb\_from\_loc(key):

if x in truck\_list[1].packages:

myHashPackage.update\_package\_status(x, 'Delivered at {} by Truck 2'

.format(clock.show\_time\_delivered(value)))

truck\_list[1].packages.remove(x)

# Changes status for and removes packages that have been delivered by truck 3

for key, value in truck\_list[2].route.items():

if total\_miles\_traveled() >= value:

for x in myHashPackage.pnumb\_from\_loc(key):

if x in truck\_list[2].packages:

myHashPackage.update\_package\_status(x, 'Delivered at {} by Truck 3'

.format(clock.show\_time\_delivered(value)))

truck\_list[2].packages.remove(x)

# Creates the truck class and functions

class Truck:

def \_\_init\_\_(self, number, packages, mileage, route):

self.number = number

self.packages = packages

self.mileage = mileage

self.route = route

# Prints the truck information

def truck\_number(self):

print('Truck Number: {}\n Packages: {}\n Mileage: {}\n Route: {}\n'.format(self.number, self.packages, self.mileage, self.route))

# Loads a truck with a package number

def load\_truck(self, package\_number):

self.packages.append(package\_number)

def add\_route\_locations(self, location):

self.route[location] = 0

# Creates 3 truck objects with truck number, an empty list to populate with packages, and starting mileage of 0

truck\_list = []

for i in range(3):

truck\_list.append(Truck(i + 1, [], 0, {}))

# Creates the clock object so I can show the time and package status

class Clock:

def \_\_init\_\_(self, hours, minutes):

self.hours = hours

self.minutes = minutes

def show\_time(self):

temp\_clock\_minutes = ''

temp\_clock\_hours = clock.hours

if clock.minutes < 10:

temp\_clock\_minutes = '0{}'.format(clock.minutes)

else:

temp\_clock\_minutes = clock.minutes

global pm

pm = 0

if clock.hours > 12:

temp\_clock\_hours -= 12

if clock.hours > 11:

pm = 1

return '{}:{} {}'.format(temp\_clock\_hours, temp\_clock\_minutes, clock.am\_pm())

def show\_hours(self):

return clock.hours

def show\_minutes(self):

return clock.minutes

def am\_pm(self):

if pm == 1:

return 'PM'

else:

return 'AM'

def show\_time\_delivered(self, distance):

pm = 'AM'

a = int(((distance // 18) + 8))

if a > 11:

pm = 'PM'

if a > 12:

a -= 12

b = int(((distance % 18) / 18) \* 60)

if b < 10:

b = '0' + str(b)

if int(b) > 60:

b -= 60

a += 1

if a > 11:

pm = 'PM'

if a > 12:

a -= 12

return '{}:{} {}'.format(a, b, pm)

def advance\_10\_minutes(self):

clock.minutes += 10

if clock.minutes > 59:

clock.minutes -= 60

clock.hours += 1

def total\_miles\_traveled():

total\_hours\_elapsed = clock.hours - 8

total\_minutes\_elapsed = (total\_hours\_elapsed \* 60) + clock.minutes

return total\_minutes\_elapsed \* 0.3

# Creates the clock object and assigns a starting time

clock = Clock(8, 0)

# Creates the distance data list and the hash table objects

myDistanceData = {}

myHashPackage = ChainingHashTable()

myHashAddresses = ChainingHashTable()

load\_address\_data('WGUPSDistanceTable.csv')

load\_package\_data('WGUPSPackageFile.csv')

load\_distance\_data('WGUPSDistanceTable.csv')

truck1\_packages = [14, 15, 16, 34, 20, 21, 19, 1, 13, 39]

truck1\_route = [20, 21, 17, 4, 5, 6, 0, 24, 13, 14, 11, 9, 25, 1, 23, 10, 16]

truck1\_reload\_packages = [25, 26, 6, 17, 28, 2, 33, 10, 27, 35, 23, 11, 12]

truck2\_packages = [4, 40, 31, 32, 36, 18, 7, 29, 5, 37, 38, 8, 30, 3, 24, 22]

truck2\_route = [18, 15, 7, 3, 2, 19, 12, 8, 22, 26, 0]

# Package 9 has no deadline and can only be loaded after 10:20 AM so the driver of truck 2 will deliver it after returning from their route

truck3\_packages = [9]

truck3\_route = [12]

# Initial load of trucks

for x in truck1\_packages:

truck\_list[0].load\_truck(x)

myHashPackage.update\_package\_status(x, 'EN\_ROUTE / TRUCK\_1')

for x in truck2\_packages:

truck\_list[1].load\_truck(x)

myHashPackage.update\_package\_status(x, 'EN\_ROUTE / TRUCK\_2')

#Initial load of routes

for x in truck1\_route:

truck\_list[0].add\_route\_locations(x)

for x in truck2\_route:

truck\_list[1].add\_route\_locations(x)

# Adds the distance

cur\_loc = 0

for keys in truck\_list[0].route:

truck\_list[0].route[keys] = dist\_locations(cur\_loc, keys)

cur\_loc = keys

cur\_loc = 0

for keys in truck\_list[1].route:

truck\_list[1].route[keys] = dist\_locations(cur\_loc, keys)

cur\_loc = keys

# Changes distances in dictionary to cumulative distances

temp\_value = 0

for key, value in truck\_list[0].route.items():

temp\_value += value

truck\_list[0].route[key] = round(temp\_value, 1)

temp\_value = 0

for key, value in truck\_list[1].route.items():

temp\_value += value

truck\_list[1].route[key] = round(temp\_value, 1)

print('\nTime: {}\nTotal mileage: {}'.format(clock.show\_time(), total\_miles\_traveled()))

inp = int(input('\n0) Exit: \n1) Display all packages: \n2) Display all trucks: \n3) Determine route: \n'

'4) Set time: \n5: Look up package: \n'))

while inp != 0:

if inp == 1:

package\_update()

# Split into 2 sections to make screenshots look better

print('\n{} {}\n'.format(clock.show\_time(), total\_miles\_traveled()))

for i in range(len(myHashPackage.table) // 2):

print(myHashPackage.search(i + 1))

print('\n{} {}\n'.format(clock.show\_time(), total\_miles\_traveled()))

for i in range(len(myHashPackage.table) // 2):

print(myHashPackage.search((i + 1) + (len(myHashPackage.table) // 2)))

if inp == 2:

package\_update()

for i in range(3):

for keys, values in truck\_list[i].route.items():

max\_miles = values

if max\_miles < total\_miles\_traveled():

truck\_list[i].mileage = max\_miles

else:

truck\_list[i].mileage = round(total\_miles\_traveled(), 1)

if truck\_list[2].mileage > truck\_list[1].mileage:

truck\_list[2].mileage = round((truck\_list[2].mileage - truck\_list[1].mileage), 1)

truck\_list[i].truck\_number()

print('Total Mileage Combined: {:.2f}'.format(truck\_list[0].mileage + truck\_list[1].mileage +truck\_list[2].mileage))

if inp == 3:

all\_package\_numbers = []

for i in range((len(myHashPackage.table)) + 1):

all\_package\_numbers.append(i)

# Package black list will make it so the packages at address don't show already delivered

package\_black\_list = []

# White list to prioritize deadline packages first, then specific truck packages (the grouped up packages

# will go on truck 1)

# 2 late deadline packages will go on third truck, the rest will be split in half between truck 1 and truck 2

white\_list = []

# Preload visited list to blacklist locations

black\_visited\_list = []

a = set(all\_package\_numbers)

b = set(white\_list)

c = black\_visited\_list

# To use the white list, visited\_list = d, to use the black list, visited\_list = c

d = list(a - b)

visited\_list = c

temp\_loc = 0

print('\nDetermine Route...\n')

inp\_1 = int(input('Starting location?: '))

while inp\_1 != 'q':

if -1 < inp\_1 > 27:

print('Invalid location')

inp\_1 = 'q'

continue

visited\_list.append(int(inp\_1))

temp\_loc = closest\_location(inp\_1).split(', ')

print(temp\_loc[1])

print(package\_black\_list)

temp\_pack\_list = myHashPackage.pnumb\_from\_loc((int(temp\_loc[1])))

for i in range(len(temp\_pack\_list)):

if temp\_pack\_list[i] in package\_black\_list:

temp\_pack\_list.remove(temp\_pack\_list[i])

temp\_pack\_list.insert(i, '--')

else:

package\_black\_list.append(temp\_pack\_list[i])

print('Dist: {},Location: {}, Packages at address: {}'.format(temp\_loc[0], temp\_loc[1],

temp\_pack\_list))

inp\_1 = int(input('Starting location?: '))

if inp == 4:

hours\_inp = int(input('\nSet Hour: '))

if hours\_inp > 12 or hours\_inp < 0:

print('\nInvalid Selection\n')

continue

minutes\_inp = int(input('\nSet minutes: '))

if minutes\_inp > 59 or minutes\_inp < 0:

print('\nInvalid Selection\n')

continue

ampm\_inp = int(input('\n1 = AM, 2 = PM'))

if ampm\_inp == 2 and hours\_inp != 12:

hours\_inp += 12

clock.hours = hours\_inp

clock.minutes = minutes\_inp

if inp == 5:

package\_update()

myHashPackage.lookup()

print('\nTime: {}\nTotal mileage: {}'.format(clock.show\_time(), total\_miles\_traveled()))

inp = int(input('\n0) Exit: \n1) Display all packages: \n2) Display all trucks: \n3) Determine route: \n'

'4) Set time: \n5: Look up package: \n'))

**D: Identify a self-adjusting data structure, such as a hash table, that can be used with the algorithm identified in part A to store the package data.**

The self-adjusting data structure I used in my program is a chaining hash table. I based my implementation on the chaining hash table found in the supplemental resources Let’s Go Hashing webinar (Tepe, 2020). This allowed the algorithm to extract addresses from package numbers to convert to locations which were compared on the distance table to determine the next closest location.

**D1: Explain how your data structure accounts for the relationship between the data points you are storing.**

My data structure allows the extraction of package information from a package number key which is converted to a location number. The location numbers are compared to each other to determine the absolute closest location.

**E: Develop a hash table, without using any additional libraries or classes, that has an insertion function that takes the following components as input and inserts the components into the hash table.**

class ChainingHashTable:

# Hash table constructor with capacity for 40 packages

def \_\_init\_\_(self, initial\_capacity=40):

self.table = []

for i in range(initial\_capacity):

self.table.append([])

# Hash table insert as well as update function

def insert(self, key, item):

bucket = hash(key) % len(self.table)

bucket\_list = self.table[bucket]

for kv in bucket\_list:

if kv[0] == key:

kv[1] = item

return True

key\_value = [key, item]

bucket\_list.append(key\_value)

return True

# Defines the package type of object

class Package:

def \_\_init\_\_(self, pnumber, street\_address, city, state, zipcode, deadline, weight, status):

self.pnumber = pnumber

self.street\_address = street\_address

self.city = city

self.state = state

self.zipcode = zipcode

self.deadline = deadline

self.weight = weight

self.status = status

# Adds strings so that hash table data is returned instead of storage locations

def \_\_str\_\_(self):

return "%s, %s, %s, %s, %s, %s, %s, %s" % (self.pnumber, self.street\_address, self.city, self.state,

self.zipcode, self.deadline, self.weight, self.status)

# Loads package data and inserts into table myHashPackage

def load\_package\_data(filename):

with open(filename) as WGUPSPackageFile:

package\_data = csv.reader(WGUPSPackageFile, delimiter=',')

for i in range(8):

next(package\_data)

for package in package\_data:

pID = int(package[0])

pstreet\_address = package[1]

pcity = package[2]

pstate = package[3]

pzipcode = package[4]

pdeadline = package[5]

pweight = package[6]

pstatus = 'AT\_HUB'

package = Package(pID, pstreet\_address, pcity, pstate, pzipcode, pdeadline, pweight, pstatus)

myHashPackage.insert(pID, package)

**F: Develop a look-up function that takes the following components as input and returns the corresponding data elements.**

# Lookup function with separate UI that allows users to look up a package using different criteria

def lookup(self):

print('\nHow would you like to look up package?')

inp = int(input(

'\n0) Exit: \n1) Number: \n2) Street Address: \n3) Deadline: \n4) City: \n5) Zip code: \n6) Weight: \n'

'7) Status: \n'))

while inp != 0:

if inp == 1:

numbInp = int(input('\nPackage number?: '))

print(myHashPackage.search(numbInp))

return

if inp == 2:

streetInp = input('\nStreet address?: ')

for i in range(len(myHashPackage.table)):

streetStr = str(myHashPackage.search(i + 1))

splitStr = streetStr.split(', ')

if streetInp in splitStr[1]:

print(streetStr)

return

if inp == 3:

deadline\_inp = input('\nDeadline?: ')

for i in range(len(myHashPackage.table)):

deadline\_str = str(myHashPackage.search(i + 1))

splitStr = deadline\_str.split(', ')

if deadline\_inp in splitStr[5]:

print(deadline\_str)

return

if inp == 4:

city\_inp = input('\nCity?: ')

for i in range(len(myHashPackage.table)):

city\_str = str(myHashPackage.search(i + 1))

splitStr = city\_str.split(', ')

if city\_inp in splitStr[2]:

print(city\_str)

return

if inp == 5:

zip\_inp = input('\nZip code?: ')

for i in range(len(myHashPackage.table)):

zip\_str = str(myHashPackage.search(i + 1))

splitStr = zip\_str.split(', ')

if zip\_inp in splitStr[4]:

print(zip\_str)

return

if inp == 6:

weight\_inp = input('\nWeight(kg)?: ')

for i in range(len(myHashPackage.table)):

weight\_str = str(myHashPackage.search(i + 1))

splitStr = weight\_str.split(', ')

if weight\_inp in splitStr[6]:

print(weight\_str)

return

if inp == 7:

status\_inp = input('\nStatus(AT\_HUB, EN\_ROUTE, DELIVERED?: ')

for i in range(len(myHashPackage.table)):

status\_str = str(myHashPackage.search(i + 1))

splitStr = status\_str.split(', ')

if status\_inp in splitStr[7]:

if status\_str == 'DELIVERED':

print('Time Delivered: {}'.format(status\_inp)) # Fill in with time delivered

print(status\_str)

return

return

**G: Provide an interface for the user to view the status and info (as listed in part F) of any package at any time, and the total mileage traveled by all trucks. (The delivery status should report the package as at the hub, en route, or delivered. Delivery status must include the time.)**

inp = int(input('\n0) Exit: \n1) Display all packages: \n2) Display all trucks: \n3) Determine route: \n'

'4) Set time: \n5: Look up package: \n'))

if inp == 1:

if inp == 2:

if inp == 3:

if inp == 4:

if inp == 5:

The while loop of the user interface allows users to search a specific package or view all packages as well as view the status of all trucks which include mileage at any given time (Lysecky, C949: Data Structures & Algorithms I, 2018).

**G1: Provide screenshots to show the status of all packages at a time between 8:35 a.m. and 9:25 a.m.**

Text

Description automatically generated

**G2: Provide screenshots to show the status of all packages at a time between 9:35 a.m. and 10:25 a.m.**

Text

Description automatically generated

**G3: Provide screenshots to show the status of all packages at a time between 12:03 p.m. and 1:12 p.m.**

A close-up of a document

Description automatically generated with medium confidence

**H: Provide a screenshot or screenshots showing successful completion of the code, free from runtime errors or warnings, that includes the total mileage traveled by all trucks.**

Text

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Text

Description automatically generated

**I: Justify the core algorithm you identified in part A and used in the solution by doing the following:**

**I1: Describe at least two strengths of the algorithm used in the solution.**

The Greedy algorithm is simple to implement and code as well as being simple to understand. It is also very flexible and can adapt to different data pools

**I2: Verify that the algorithm used in the solution meets all requirements in the scenario.**

The algorithm was able to determine a route and package delivery order to get all packages delivered in time meeting all special requirements. This was done efficiently at less than 140 miles (101 miles).

**I3: Identify two other named algorithms, different from the algorithm implemented in the solution, that would meet the requirements in the scenario.**

A heuristic algorithm could be used to reduce complexity and runtime. This algorithm finds a fast but not optimal solution. A Dijkstra algorithm finds the shortest path between vertices (Lysecky, C950: Data Structures and Algorithms II, 2018). This is different than my Greedy algorithm because it looks at the entire route and compares that to other acceptable routes to find the optimal solution. This is more complex and harder to implement but finds a better solution.

**I: Describe what you would do differently, other than the two algorithms identified in I3, if you did this project again.**

I would have a cleaner layout of code. I was learning as I was going along, so I’m sure I have some redundancies in my code, and it could be organized better. I have good comments in my code, but I would provide more and highlight the process of using the algorithm and show how the algorithm can be adapted to suit different data sets

**K: Justify the data structure you identified in part D by doing the following:**

**K1: Verify that the data structure used in the solution meets all requirements in the scenario.**

**K1a: Explain how the time needed to complete the look-up function is affected by changes in the number of packages to be delivered.**

The lookup function searches through every package and compares the string of the value of the package key against the user input. More packages will make the lookup take longer, but not exponentially longer. This could be optimized by using fewer buckets in a different type of data structure

**K1b: Explain how the data structure space usage is affected by changes in the number of packages to be delivered.**

The space usage is also increased by each package. It is increased by double the packages because each package has a unique key and value. This could also be reduced by having fewer buckets.

**K1c: Describe how changes to the number of trucks or the number of cities would affect the look-up time and the space usage of the data structure.**

The changes to the number of trucks would increase the lookup time slightly because more trucks mean different statuses associated with loading each truck, but this change would be negligible. A change in the number of cities or locations would affect the lookup time exponentially since the distance table is two-dimensional which would add x and y values to finding locations based on addresses. This would also have an exponential effect on space usage.

**K2: Identify two other data structures that could meet the same requirements in the scenario.**

A linear probing hash table and a dictionary.

**K2a: Describe how each data structure identified in part K2 is different from the data structure used in the solution.**

Different types of hash tables can be used, such as a hash table with linear probing. This would involve storing the attributes associated with each package into different buckets. This is more efficient for space usage but can add to look-up time to calculate positions after collisions. A dictionary can also be used and is similar to chaining hash table but is simpler to implement, although a dictionary is limited to the Python language

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

# Works Cited

Lysecky, R. &. (2018). *C949: Data Structures & Algorithms I*. Retrieved from zyBooks: https://learn.zybooks.com/zybook/WGUC9492018

Lysecky, R. &. (2018, June). *C950: Data Structures and Algorithms II*. Retrieved from zyBooks: https://learn.zybooks.com/zybook/WGUC950AY20182019/

Tepe, C. (2020). *Let’s Go Hashing [Webinar]*. Retrieved from WGU: https://wgu.hosted.panopto.com/Panopto/Pages/Viewer.aspx?id=f08d7871-d57a-496e-a6a1-ac7601308c71