Student Name: Steven Hatch

Student ID: -

Date: 6/5/21

Write-Up

A. Two self-adjusting algorithms were used to deliver packages (see Self\_Adjusting\_Algorithms.py)

* Greedy Algorithm = setPackagesInTrucks()
* Nearest Neighbor Algorithm = setOptimalPackageListAllTrucks()

B. Overview of Program

B1: Explanation of the algorithm logic using pseudocode (also listed in Self\_Adjusting\_Algorithms.py):

* Self-Adjusting Algorithm #1 - Greedy Algorithm - setPackagesInTrucks()
  + Overview: algorithm prioritizes earliest deadline times and fills up first two departing trucks (T1 and T2)
  + algorithm gets list of all packages sorted by earliest deadline from getPackageDeliveryDeadlineList()
  + algorithm places packages in first two trucks until they are full (# 14 package limit for equal distribution)
  + algorithm places remaining packages in truck departing latest (T3)
  + while placing packages, algorithm appends each package to packagesAlreadyPlaced list to prevent duplicates in event that algorithm needs to be called again (if deadlines change, if total number of packages change, etc.)
* Self-Adjusting Algorithm #2 - Nearest Neighbor Algorithm - setOptimalPackageListAllTrucks()
  + Found in Distance.py at line 425
  + Overview: algorithm sets optimal package list order to minimize route mileage for each truck
  + algorithm searches all possible destinations to determine shortest location from hub, then loops to determine shortest distance from each location to the next, then optimally sets address key/names - loopToSetAllAddressesInOptimalRouteOrder() - getShortestDistanceFromHub(), getNextShortestDistanceForRouteList()
  + algorithm then converts optimally sorted address key/names into optimally sorted street addresses - setOptimalPackageListAddressesT1()
  + algorithm then matches optimally sorted street addresses to package delivery destinations and sets package IDs in optimal order - setOptimalPackageListT1()
  + algorithm then replaces packageList with optimalPackageList

B2: Description of programming environment used to create the Python application:

* PyCharm was used for the IDE
* PyCharm was running on Windows 10 64-bit OS on my desktop computer

B3: Space-time complexity of each major segment of program, and the entire program, using big-0 notation included in code comments

* Each major segment included in line
* Entire program included in Main.py line 5 = O(N^2) quadratic

B4: Explanation of the capability of my solution to scale and adapt to a growing number of packages:

* Hash table includes parameter to change initial capacity, allowing for additional package storage
* CSV import is capable of reading and storing additional packages from larger package lists
* Self-adjusting algorithm #1: packages are self-placed into trucks based on delivery deadline and truck departure time
* Self-adjusting algorithm #2: packages are self-sorted to determine optimal delivery route (lowest mileage)
* Main function deliveryAddressUpdateCheck: updates package delivery address based on specified criteria then re-initializes data
* Identified constraints are number of trucks, overall truck package capacity, and number of truck drivers

B5: Discussion of why the software is efficient and easy to maintain:

* Efficiency:
  + Big-O time complexity of entire program = O(N^2) quadratic
  + User Interface:
    - Straightforward prompts for user to get desired data
* Maintainability for Future Developers:
  + Code structure:
    - Each py file follows same formation (ex. initialization, functions)
  + Code comments:
    - Each major section of code is labeled with its purpose
  + Compartmentalization:
    - All major functions have been combined for easier readability (ex. setTotalRouteMileageAllTrucks instead of setForT1, setForT2, setForT3
    - Main file only requires two function calls
      * mainUserInterfaceInit()
      * mainUserInterface()

B6: Strengths and weaknesses of the self-adjusting data structures (e.g., the hash table).

* Hash Table
  + Strengths
    - Allows for fast inserts, searches, and removals (up to O(1))
    - Easy to implement (python has built-in hash() function)
    - Excellent for data integrity verification and cryptography
  + Weaknesses
    - Needs to be designed to avoid collisions
    - Slow if there are a high number of collisions
    - Helps to know all possible item keys beforehand (not always available)

C. Original program written to deliver all packages, meeting all requirements, using the attached supporting documents

C1: Identifying comment included in first line of Main.py

C2: Comments in code have been included to explain process and flow of program.

D. Self-adjusting data structure that can be used with self-adjusting algorithms in part A to store the package data = hash table

D1: Explanation of how data structure accounts for the relationship between the data points being stored

* + Relationship between data points being stored and hash table
    - Data points being stored = package data
    - Package data is stored in hash table as package objects
      * p = Package( pID, pAddress, pCity, pState, pZipCode, pDeadline, pWeight, pSpecialNotes, pStatus)
      * Inserted into hash table as packageHashTable.insert(pID, p)
        + pID = key to search, insert(update/insert new), and remove package object from hash table
    - Hash table retrieves information accurately and more efficiently than linear search due to its bucket design
      * Bucket is determined by hash(key) % len(self.table)
        + ex. package 40

Instead of searching 1-40 to find p40 (linear search) (O(N)), hash table allows for p40 to be placed in bucket 0 and found at O(1)

* + Relationship between self-adjusting algorithm #1 (greedy algorithm) and hash table
    - Algorithm searches through all package objects in hash table to build package list sorted by pDeadline (delivery deadlines)
      * This list (sortedDeadlineList) is then implemented to set packages with earlier delivery deadlines to the trucks departing earliest
  + Relationship between self-adjusting algorithm #2 (nearest neighbor) and hash table
    - After determining optimal routes for all trucks, algorithm searches through package objects in hash table to verify that a match has been found between addresses in optimalPackageListAddresses (optimal street addresses) and pAddresses (addresses of packages from hash table)
      * Once match is found, pID (package ID) is appended to optimalPackageList

E. Hash table developed without using any additional libraries or classes. Hash table has an insertion function that takes all required components as input and inserts all required components in the hash table

* see Hash.py, Hash.py – insert, Package.py - packageHashTable.insert(pID, p)

F. Look-up function developed that uses package IDs as input and returns all corresponding package data elements

* see Hash.py – search, Package.py – getPackageDataByID

G. User interface provided that allows viewing of status and info of any package at any time, and the total mileage traveled by all trucks

* see Main.py – mainUserInterface
* All screenshots are included in zip file submission in project\_screenshots folder
  + Screenshot of the status of all packages at a time between 8:35am and 9:25am (time=8:45am)
  + Screenshot of the status of all packages at a time between 9:35am and 10:25am (time=10:15am)
  + Screenshot of the status of all packages at a time between 12:03pm and 1:12pm (time=1:05pm/13:05 military time)

H. Screenshot provided showing successful completion of the code, free from runtime errors or warnings, that includes total mileage traveled by all trucks

* screenshot included in zip file submission in project\_screenshots folder

I. Justification of core algorithms identified in part A

I1. Strengths of the algorithms used:

* Self-Adjusting Algorithm #1 - Greedy Algorithm - setPackagesInTrucks()
  + Strength 1: Ensures that packages with earliest delivery times are placed on first departing trucks
    - Aims to prevent delivery past delivery deadline
      * Works for this project (unsure about scalability)
  + Strength 2: Prevents duplicate packages being added to trucks (in event that algorithm needs to be called again (if deadlines change, if total number of packages change, etc.)
    - * see Self\_Adjusting\_Algorithms.py, packagesAlreadyPlaced
* Self-Adjusting Algorithm #2 - Nearest Neighbor Algorithm - setOptimalPackageListAllTrucks()
  + Strength 1: Sets optimal package list order to minimize route mileage for each truck
  + Strength 2: Allows flexibility of package list size
    - Based on number of packages in truck == number of unique address + duplicate address
    - see Distance.py, loopToSetAllAddressesInOptimalRouteOrder
  + Strength 3: Accounts for duplicate delivery addresses
    - Appends duplicate delivery address after first instance of address to maintain optimal route / package list order
    - see Distance.py, loopToSetAllAddressesInOptimalRouteOrder

I2: Verification that the algorithm used meets all requirements

* Total combined miles traveled by all trucks = 130.7 miles
  + Can verify through user interface, ‘View Truck Data’
  + Can also verify by enabling the following print functions:
    - Diatance.py (original print not reflective of optimal miles – check console after entering time into user interface to see print of total miles that were set based on optimal route being calculated ex. T2 miles = 58.1 -> 52.8)
      * setTotalRouteMileageAllTrucks(), print(mileageToSetT1)
      * setTotalRouteMileageAllTrucks(), print(mileageToSetT2)
      * setTotalRouteMileageAllTrucks(), print(mileageToSetT3)
* All packages were delivered on time
  + Can verify through user interface, enter time >= 1159
  + Can also verify by enabling the following print functions:
    - Distance.py (float list of route distance ---- last distance = distance to hub)
      * updateTotalMilesAndPopulateDistancesT1(), print(distanceFloatListT1)
      * updateTotalMilesAndPopulateDistancesT2(), print(distanceFloatListT2)
      * updateTotalMilesAndPopulateDistancesT3(), print(distanceFloatListT3)
    - Time.py (time between locations in minutes ---- last time = distance to hub)
      * setTimeBetweenLocationsListT1 (), print(timeBetweenLocationsListT1)
      * setTimeBetweenLocationsListT2 (), print(timeBetweenLocationsListT2)
      * setTimeBetweenLocationsListT3 (), print(timeBetweenLocationsListT3)
    - Time.py (delivery times -- original print not reflective of optimal times – check console after entering time into user interface to see print of delivery times that were set based on optimal route being calculated ex. deliveryTimesToReturnT2 last delivery = 1051 -> 1033)
      * setAndReturnDeliveryTimesT1(), print(deliveryTimesToReturnT1)
      * setAndReturnDeliveryTimesT2(), print(deliveryTimesToReturnT2)
      * setAndReturnDeliveryTimesT3(), print(deliveryTimesToReturnT3)
* All packages were delivered according to their delivery specifications
  + Packages 13, 14, 15, 16, 19, and 20 initialized in truck 1
    - Meets requirement of being out for delivery on same truck
    - To verify, see Self\_Adjusting\_Algorithms.py, packageListT1
    - Can also verify package list by enabling the following print function:
      * Distance.py
        + setOptimalPackageListAllTrucks, print(optimalPackageListT1)
  + Packages 3, 18, 36, and 38 initialized in truck 2
    - Meets requirement of only being delivered by truck 2
    - To verify, see Self\_Adjusting\_Algorithms.py, packageListT2
    - Can also verify package list by enabling the following print function:
      * Distance.py
        + setOptimalPackageListAllTrucks, print(optimalPackageListT2)
  + Packages 6, 25, 28, and 32 initialized in truck 3
    - Meets requirement of not leaving hub until 9:05am
    - To verify, see Self\_Adjusting\_Algorithms.py, packageListT3 and Truck.py, timeLeftHubT3
    - Can also verify package list by enabling the following print function:
      * Distance.py
        + setOptimalPackageListAllTrucks, print(optimalPackageListT3)
  + Package 9 also initialized in truck 3
    - Meets requirement of not being delivered until 10:20am
    - To verify, see package 9 delivery time through UI console output
      * Package 9 is delivered at 10:33am

I3: Two other named algorithms, different from the algorithms implemented, that would meet the requirements in the scenario

* Algorithm 1 = Dijkstra’s Shortest Path Algorithm
  + Could be implemented to minimize total mileage
* Algorithm 2 = Breadth-first search
  + Could also be implemented to minimize total mileage
* I3A: How both algorithms differ from my algorithm
  + My solution is not graph based
  + My solution only finds the shortest path from one location to the next (irrespective of each location’s distance from starting location / hub)
  + Dijkstra’s Shortest Path Algorithm
    - I3A (continued): Dijkstra’s algorithm “finds the shortest path from a start vertex to each vertex in a graph” (Lysecky and Vahid, 2017, section 6.11)
  + Breadth-first search
    - I3A (continued): Breadth-first search “visits a starting vertex, then all vertices of distance 1 from that vertex, then of distance 2, and so on, without revisiting a vertex.” (Lysecky and Vahid, 2017, section 6.5)

J. If I was to do this project again, other than attempting to implement the algorithms mentioned in I3, I would do the following:

* Implement randomization of packages into the truck loading
  + This could allow for more optimal routes as the result of the lowest total mileage through randomization could be logged, then saved, to place the packages in that specific order, thus decreasing total mileage
* Redistribute my functions
  + Most of them ended up being more conveniently placed in the Distance.py file (leaving the other files sparser)

K. Justification of the data structure identified in part D (hash table)

K1: Verification that the data structure used meets all requirements in the scenario

* + Total combined miles traveled by all trucks = 130.7 miles
  + All packages were delivered on time
  + All packages were delivered according to their delivery specifications
  + An ‘efficient’ hash table with a look-up function is present
  + The ‘reporting’ (package statuses and information) can be verified through the user interface
    - All information is accurate

K1A: Time needed to complete the look-up function is affected by changes in the number of packages to be delivered

* + - If additional packages need to be delivered, additional packages will need to be added, resulting in inevitable collisions, thus resulting in longer look-up times unless structural change is made to the hash table (i.e., adding more buckets)

K1B: The data structure space usage is affected by changes in the number of packages to be delivered

* + - If additional packages are added, buckets in the hash table will become fuller, prompting a need for a solution on how to deal with collisions (potentially needing more space)
    - If packages are removed, buckets in the hash table will become emptier, freeing space usage for packages being added in the future to utilize

K1C: Changes to the number of trucks or the number of cities would affect the look-up time and the space usage of the structure

* + - As the hash table is only implemented to store package data, changes specific to the number of trucks or the number of cities would not directly affect the look-up time or the space usage of the data structure
    - However, alongside the assumption that more packages would need to be delivered (based on in increase in the number of trucks and delivery to more cities), the look-up function would become slower due to collisions as more packages would need to be added to the table
    - Look-up speed could be increased at the cost of additional space usage (more buckets would need to be added)

K2: Two other data structures that could meet the same requirements in the scenario:

* Data structure 1 = Dictionary / Nested Dictionary
  + K2A: Rather than storing packages as objects in the hash table, packages could be stored as key/value(s) pair in a dictionary
    - This would allow for convenience of dictionary method usage (ex. update, pop, for key in dictionary loop) (Lysecky and Vahid, 2017, section 9.4)
    - This would add consistency in the project as the distance data is implemented as a dictionary
* Data structure 2 = Set with BST
  + K2A: Contrasting with the hash table, a set would allow for operations such as union, intersection, and difference when comparing two sets, filter to create a subset, and map to create a new set based on a function (Lysecky and Vahid, 2017, section 8.4)
    - Packages could be established as a dynamic set or static set

L. Sources included below

Ashe, Jim. (14 May 2020) *C950 Task Directions Summary Alternative Task Directions and FAQ (NHP2)* [document]*.* C950 Task Directions Summary, Google Docs, docs.google.com/document/d/1Sgcd3blGzU3oQOjFrDfSq1bluxgU02Bv8tzCA9aDuac/edit#

Lysecky, Roman, and Frank Vahid. (December 2017) *C950: Data Structures and Algorithms II* [e-text]. ZyBooks, Zyante Inc., learn.zybooks.com/zybook/WGUC950AY20182019

Tepe, Cemal. (17 Nov. 2020). *C950 - Webinar-1 - Let's Go Hashing* [video], Western Governors University, Panopto, wgu.hosted.panopto.com/Panopto/Pages/Viewer.aspx?id=f08d7871-d57a-496e-a6a1-ac7601308c71