A: ALGORITHM SELECTION

*Identify the name of the algorithm used to create a program to deliver the packages and meet all requirements specified in the scenario.*

The core algorithm used for routing is a method of the Stop class found in Stop.py and is called core\_routing\_algorithm. This method uses a greedy algorithm to find the next closest address among the packages loaded on the trucks. It will then self-adjust by removing the address from the list of required stops and iterate until every stop is accounted for.

B1: LOGIC COMMENTS

*The comments accurately explain the logic applied to the solution.*

The program consists of importing the data from CSV, manually loading trucks, a greedy algorithm for determining route. A greedy algorithm was used because with optimally loaded trucks, it provides an adequate route with low travel distance among stops that are close geographically. I achieved this greedy algorithm in linear time by making sure the distances list was sorted in ascending order. Therefore, iterating through the distances list and checking if “in” the truck’s list of addresses at which it needs to stop will return the lowest distance from the current address.

core\_routing\_algorithm:

* Create a dictionary of Stop objects keyed on ascending integers beginning at 0 for hub
* Beginning at the hub and then for each unique address on the truck:
  + Call greedy\_next, passing most up-to-date list of addresses that must be visited
  + Add resultant nearest stop to dictionary
  + Remove visited address from list that must be visited
  + Continue

greedy\_next:

* Create list of string addresses from list of addresses on truck that must be visited
* For each tuple in the current stop’s address’s sorted list of target addresses paired with distance:
  + If target address exists in the list of addresses that must be visited by truck
    - Because the list is sorted, this is the nearest. Construct and return stop object
  + Continue

B2: APPLICATION OF PROGRAMMING MODELS

*Provide a description of the communications protocol used to exchange data, the target host environment used to host the application server application, and the interaction semantics defined by the application to control connect, data exchange, and disconnect sequences.*

This program is written in Python 3.8. All data required is imported from provided CSV files stored locally in /res directory. As such, no communications protocols are required.

B3: SPACE-TIME AND BIG-O

*The evaluation shows the space-time complexity using Big-O notation for each major block of code and the entire program.*

Space and time complexities are found in comments. Of note: list.sort is used to ensure distance data enters the core algorithm sorted and runs in O(n log n) time; everything else in the program runs in linear time at worst (provided the sorted input) and therefore the overall complexity is driven by this sorting operation.

B4: ADAPTABILITY

*The discussion includes the chosen algorithm’s ability to handle a growing number of packages (scalability).*

The routing algorithm will scale well for any number of properly loaded trucks because the maximum number of packages on a truck is 16. The approach doesn’t scale particularly well with manually loading the trucks but given the horrendously inadequate and inconsistent location data provided for the project, manually loading is the only way that makes much sense.

B5: SOFTWARE EFFICIENCY AND MAINTAINABILITY

*The discussion addresses how the software’s efficiency and why it is efficient and easy to maintain.*

Entire program runs in O(n log n). This complexity is driven by the need to list.sort the distance data

The program is maintainable due to reasonable use of classes and decoupling functionality across data structures.

B6: SELF-ADJUSTING DATA STRUCTURES

*The discussion of the self-adjusting data structure(s) includes the ability of the data structure(s) to adapt when data is inserted, removed, or accessed; and how that adaptation affects running time.*

See section D for information regarding self-adjusting hash table data structure

C: ORIGINAL CODE

*The original code runs properly and delivers all packages within their requirements while adding the least number to the combined mileage total of all trucks in less than 145 miles.*

Program user interface provides all information required to verify that program meets specified

C1: IDENTIFICATION INFORMATION

*The initial comment is located within the first line of code and includes the candidate’s first name, last name, and student ID.*

Comments for identification located in Main.py and in the header of this document

C2: PROCESS AND FLOW COMMENTS

*Include comments at each major block of code explaining the process and flow of the code.*

Submitted code is commented.

D: DATA STRUCTURE

*Identify a data structure written by the student using only primitive data structures, lists, tuples, or sets, used by your program to store and retrieve package data.*

The custom hash table implementation for Package data is located in HashTable.py

D1: EXPLANATION OF DATA STRUCTURE

*The submission accurately explains the data structure and how that data structure accounts for the relationship between the data points to be stored.*

A custom hash table implementation is used to store package data. The hash table is an optimal choice for this use case because every package has an integer id. The id is hashed using a mid-square function, and the pointer to the package’s data is stored in the list located at the index of the result of the hash function in the hash table’s array.

E: HASH TABLE

*The hash table has an insertion function that includes, as input, all of the given components.*

HashTable.put located in HashTable.py accepts a key and a value. In this implementation, the key must be the integer package id, and the value must be of type Package as defined in the Package class found in Package.py.

F: LOOK-UP FUNCTION

*The look-up function includes all of the given data elements, completes searches via package ID, returns the data corresponding to the provided ID including the package’ status (at the hub, en route, or delivery time).*

HashTable.get located in HashTable.py accepts a key and returns a Package object if found.

G: INTERFACE

*Provide an interface for the user to view the status and info of any package at any time.*

The user interface located in UI.py provides status and info of packages and ability to input time at runtime.

G1-G3: 1st, 2nd, and 3rd status checks.

*Provide screenshots showing the info (outlined in part F) and statuses at a time between:*

* *8:35 a.m. and 9:25 a.m.*
  + See /project\_docs/Screenshot3\_Bohanon\_Jacob\_C950.png
* *9:35 a.m. and 10:25 a.m.*
  + See /project\_docs/Screenshot4\_Bohanon\_Jacob\_C950.png
* *12:03 p.m. and 1:12 p.m.*
  + See /project\_docs/Screenshot5\_Bohanon\_Jacob\_C950.png

H: SCREENSHOTS OF CODE EXECUTION

*Provide a screenshot or screenshots showing successful completion of the code free from runtime errors or warnings.*

See /project\_docs/Screenshot0\_Bohanon\_Jacob\_C950.png,

/project\_docs/Screenshot1\_Bohanon\_Jacob\_C950.png, /project\_docs/Screenshot2\_Bohanon\_Jacob\_C950.png

I1: STRENGTHS OF THE CHOSEN ALGORITHM

*The description includes at least two specific strengths of the chosen algorithm as they apply to the scenario.*

The strengths of the greedy algorithm as chosen for this project are:

* Time-complexity efficiency:
  + Because the distance data is sorted once at program initialization (O(n^3 log n)), any greedy lookup runs in sublinear time.
* Optimization for minimizing distance traveled on well-loaded trucks:
  + Because well-loaded trucks have a low median distance from greedy-acquired address to greedy-acquired address, the overall mileage comes out very low.

I2: VERIFICATION OF ALGORITHM

*The verification includes the total miles added to all trucks, and it states that all packages were delivered on time.*

Total mileage of 93 miles and package delivery time/truck provided via the console output.

I3: OTHER POSSIBLE ALGORITHMS

*The submission identifies two other algorithms that could meet the requirements of the scenario.*

Two alternate algorithms that could have been used are a dynamic programming approach and an A\* path-finding algorithm

I3A: ALGORITHM DIFFERENCES

*The description includes attributes of each algorithm identified in part I3 and how the identified attributes compare to the attributes of the algorithm used in the solution.*

* Dynamic programming
  + Breaks the problem down into smaller problems to be solved
  + Could have attempted programming the truck loading
  + This would likely be much more time complex but would scale better
  + Would be more appealing with better, more consistent input data, specifically regarding locations
* A\* path-finding algorithm (Patel, 1997)
  + This algorithm is likely to find a more optimal route
  + Manual truck loading required to achieve optimal route due to specific package requirements and the inconsistent location data
  + Much higher space complexity due to storing all possible routes

J: DIFFERENT APPROACH

*The description includes at least one aspect of the process that the candidate would do differently and includes how the candidate would modify the process.*

This process would be greatly improved by retrieving the data from a properly set up relational database to ensure integrity, or simply maintaining integrity throughout the csv files. Also, GPS coordinates for the addresses would make loading the trucks programmatically much, much more feasible. Having raw scalar distances with no planar relationship between locations made manually loading the trucks a required step.

K1: VERIFICATION OF DATA STRUCTURE

*The verification shows all the criteria have been met: the least number of total miles added to all trucks, all packages were delivered on time, the hash table with look-up function is present, and the reporting needed is accurate and efficient.*

The route completes in 93 miles and all packages are delivered on time as evidenced by the user interface and the screenshots. The hash table with look-up function is found in HashTable.py and discussed in sections D, E, & F.

K1A: EFFICIENCY

*The description of the efficiency of the data structure (hash table) used in the solution includes what type of data is being used and how that data is being used.*

The hash table is implemented via a list of hash buckets. Each hash bucket is a list containing all of the values whose key hashes to that particular bucket’s index. The hash table is built with a maximum bucket length of 10 items to minimize the amount of linear searching needed. At the moment any bucket would exceed 10 items, the resize method doubles the number of buckets and rehashes all elements in the hash table to the new buckets. The mid square function for ascending integer ids provides a nicely distributed table when using an appropriate amount of middle bits. As such, the hash function is designed to raise the number of bits from 20 to 32 when the number of elements exceeds 1000, keeping the number of elements in each bucket very low and maintaining the constant time property of operations such as get and put.

K1B: OVERHEAD

*The explanation of the data structure (hash table) includes the computational time, memory, and bandwidth aspects when handling data in this program.*

As discussed in K1A, time complexity maintains nearly the ideal O(1) for reading and storing package object data. Memory allocation will scale linearly with the number of packages as long as the mid-square function is properly distributing hashes. If hashes cluster into specific buckets, resize will be called more and the space and time complexity of adding elements gets much worse. Bandwidth is not a concern as all code is executed locally and all files are stored locally.

K1C: IMPLICATIONS

*In regards to the data structure (hash table), describe the implications when more packages are added to the system or other changes in scale occur.*

As discussed in K1B and consistent with all hash tables, the scalability of the data structure hinges entirely on the hashing algorithm. The mid-square function selected works really well for distributing elements as long as the number of bits chosen to represent the max size of the key allows for the input key to have a fairly random selection of middle bits. As such, the function used in this program enlarges the bit count from 20 to 32 when 1000 unique ids are exceeded. This is operating off of the assumption that the keys start at 1 and ascend iteratively. If another id schema were used, the bit determination would require alteration

K2: OTHER DATA STRUCTURES

*The submission identifies two data structures other than the one used in the solution that meets the criteria and requirements in the scenario.*

A custom hash table implementation was required per the project description. The package data were stored in objects of a custom-built Package class. Alternately, package data could be stored in dictionaries using fields as keys, or as tuples using position as the field indicator

K2A: DATA STRUCTURES DIFFERENCES

*The description includes the attributes of each data structure identified in part K2 and compares these attributes to the attributes of the data structure used in the solution.*

The main benefit of using a class for package data is the ability to associate methods. The package class in this program uses with\_status and with\_address methods to return new package objects that contain the provided attributes as well as a print method used to format the output of a package. These capabilities would be possible using a dictionary or tuple, but the code would be much less clear and maintainable

L: SOURCES

*The submission includes in-text citations for sources that are properly quoted, paraphrased, or summarized and a reference list that accurately identifies the author, date, title, and source location as available.*

Lysecky, R., & Vahid, F. (2018). 7. Hashing. In *C950: Data Structures and Algorithms II* (June 2018 ed., p. 7.6). Zyante (zyBooks.com). Retrieved December 23, 2020, from <https://learn.zybooks.com/zybook/WGUC950AY20182019/chapter/7/section/6>

Patel, A. (1997). Introduction to A\*. Retrieved December 23, 2020, from https://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html

M: PROFESSIONAL COMMUNICATION

*The content reflects an attention to detail, is organized, and focuses on the main ideas as prescribed in the task or chosen by the candidate. Terminology is pertinent, is used correctly, and effectively conveys the intended meaning. Mechanics, usage, and grammar promote accurate interpretation and understanding.*