## C905 WGUPS Documentation

Rebecca McDaniel

Student ID: 001426057

## A: ALGORITHM SELECTION

For this project I chose to implement a version of the nearest neighbor algorithm. From wherever the truck currently is, the algorithm looks at all potential next stops and moves to the nearest one. This repeats until the truck stops at all delivery addresses and then returns to the hub.

This implementation is self-adjusting in two ways. First, the hash map that holds the packages to be delivered is not fixed. You can feed it any CSV list of package information, it counts the lines, and then creates a hash table of that size. The hash function used is a simple modulo of the size of the table, so the table can change without causing collision issues.

Secondly, the part of the algorithm that chooses the route order is self-adjusting. From each stop, the algorithm chooses the nearest stop to the current one, which changes based on the stops available in the list.

## B1: LOGIC COMMENTS

Get report time from user

Create hash table

For packages in package CSV

Read packages into hash table

For packages in CSV

Read package data into hash table

Create distance array

For address in CSV

Read address into dictionary

For distances in CSV

Read distances into array

For address in dictionary

For row in distances array

Add items in row into dictionary {[address1, address2] = distance}

Create 3 truck objects

Print (package 9 address updates)

Hashmap.update package 9 data

Truck1\_list = packages on truck one

Truck2\_list = packages on truck two

Truck3\_list = packages on truck 3

Truck1 start time = 0800

Truck2 start time = 0905

For package in Truck1\_list

For next package in Truck1\_list

Store {[package1, package2] = distance} in temporary array

Sort array

Choose first item of sorted array and save to ordered\_list

Choose first value in sorted array and save to ordered\_distance\_dictionary

Remove first package from list

Return ordered\_list, ordered\_distance\_dictionary, sum(values(ordered\_distance\_dictionary)

For package in Truck1 ordered\_list

Current time = Truck1 start time

Current time = current time + convert\_distance\_to\_time(ordered\_distance\_dictionary)

Hashmap.put “delivered time = {current time} for package

Next ordered\_distance\_dictionary

Return current time

(Repeat last two blocks for truck 2)

Compare Truck1 current time with Truck2 current time

Lowest time = Truck3 start time

(repeat two blocks for truck 3 using Truck3 start time)

Total miles = sum of sum(values(ordered\_distance\_dictionary) for trucks 1, 2, and 3

For package in hash table

Print (data)

Print (total miles)

## B2: DEVELOPMENT ENVIRONMENT

This program was written on an Intel i7, Lenovo Ideapad, laptop. Software used includes Windows 11, Microsoft Office, and PyCharm IDE. Version control was handled by Git, Git Bash, and Github. The program was written in Python 3.

## 

## B3: SPACE-TIME AND BIG-O

See notes within the code for explanations.

|  |  |  |
| --- | --- | --- |
| Method/Part name | Time Complexity | Space Complexity |
| 1. Beginning interface | O(1) | O(1) |
| 2. Package.get\_number\_of\_packages | O(n) | O(n) |
| 3. Package.create\_package\_objects | O(n) - O(n^2) | O(n) |
| 4. HashTable.put | O(1) - O(n) | O(n) |
| 5. Address.put\_addresses\_in\_city\_map\_matrix | O(n) | O(n) |
| 6. Address.put\_distances\_in\_array | O(n) | O(n) |
| 7. Address.put\_distances\_in\_city\_map\_matrix | O(n^2) | O(n) |
| 8. Update package #9 part | O(1) | O(1) |
| 9. Truck.update\_truck\_in\_hashmap | O(log(n)) | O(n) |
| 10. Truck.truck\_start\_time | O(n) | O(n) |
| 11. DeliveryAlgorithm.get\_ordered\_list | O(n log n^2) | O(n log n^3) |
| 12. DeliveryAlgorithm.get\_address1 | O(1) | O(1) |
| 13. DeliveryAlgorithm.get\_distance\_between\_addresses | O(1) | O(1) |
| 14. Time.get\_delivery\_times | O(log(n)) | O(log(n)) |
| 15. Truck.truck3\_start\_time | O(1) | O(1) |
| 16. DeliveryAlgorithm.truck\_mileage | O(1) | O(1) |
| 17. Total Mileage calculation | O(1) | O(1) |
| 18. Ending interface | O(1) | O(n) |
| 19. Time.convert\_distance\_to\_time\_delta | O(1) | O(1) |
| 20. HashTable.update | O(n) | O(1) |
| 21. HashTable.get | O(1) | O(n) |
| 22. HashTable.hash\_function | O(1) | O(1) |
| 23. DistanceMatrix.add\_address | O(1) | O(1) |
| 24. DistanceMatrix.add\_address\_one\_direction | O(1) | O(1) |
| 25. DistanceMatrix.add\_distance | O(1) | O(1) |
| 31. Overall | O(n log n^2) - O(n^2) | O(n log n^3) |

## B4: ADAPTABILITY

This implementation is scalable in two major ways. First, the hash table creates itself to a size matching the list of input data. The hash function itself is also modulo the size of the table. So collisions won’t happen due to size changes.

Secondly, the nearest neighbor part of the algorithm works the same way no matter how many packages are present, and it doesn’t need to know the number ahead of time.

It fails to adapt regarding repeated addresses. The distances matrix does not store distances between an address and itself, nor does the algorithm allow a dictionary lookup for two identical addresses. This prevents the truck from continuously choosing the current stop as the “next closest” stop. This becomes a problem if two separate packages are going to the same address. Repeated addresses will not be dropped off together. The truck will drop off one, go elsewhere, and then come back. In some situations, this algorithm will even drop a repeated package entirely. A more efficient strategy is needed.

## B5: SOFTWARE EFFICIENCY AND MAINTAINABILITY

The program is efficient, in that, worst case scenario, it runs in O(n^2) time.

The program is maintainable due to extensive notes throughout the code. Attempts were made to compartmentalize topics. Also, very descriptive variable names are used.

## B6: SELF-ADJUSTING DATA STRUCTURES

This implementation is self-adjusting in two ways. First, the hash map that holds the packages to be delivered is not fixed. You can feed it any CSV list of package information, it counts the lines, and then creates a hash table of that size. The hash function used is a simple modulo of the size of the table, so the table can change without causing collision issues.

Secondly, the part of the algorithm that chooses the route order is self-adjusting. From each stop, the algorithm chooses the nearest stop to the current one, which changes based on the stops available in the list.

The code fails to be self-adjusting concerning the address matrix. In Address.py, line 40, the loop is based on the “magic number” 27. 27 is the number of addresses in the city. This would have to be adjusted if more addresses were added.

## C: ORIGINAL CODE

All code was written by me, with algorithm help referenced as appropriate in the reference section below. Only the standard python library was used. It runs without errors.

## C1: IDENTIFICATION INFORMATION

Rebecca McDaniel, Student ID: 001426057

See main.py, line 9.

## C2: PROCESS AND FLOW COMMENTS

Comments are scattered throughout the program.

## D: DATA STRUCTURE

The hash table in this program is self-adjusting. It obtains the number of packages from the provided CSV – see main.py line 48. It then automatically creates a hash table to fit that number of packages. See main.py line 50, which takes the result of line 48 as an argument.

The insertion method is called in main.py line 58. It takes the CSV and the hash table created in line 51 as arguments and adds the CSV data to the hash table. This code is also self-adjusting in that it can take packages with any ID number up until the table is full.

The function that looks up the package data by ID number is called from main.py line 233. This prints each line for the user at the end of the program. This can be done via a loop because the packages happen to be in numerical order. It is also referenced by several other methods for various parts of the program.

## D1: EXPLANATION OF DATA STRUCTURE

The hash table is implemented in the HashTable.py file. It consists of two parallel lists called self.key\_buckets and self.data\_buckets found at lines 11 and 12. These lists are set to the size of the package CSV file (table\_size) obtained in a previous method, so there is room for each package. The hash table needs to decide which list index to give to a particular package, and it does this via the hash\_function found at line 65. It takes a numerical ID for the package (the package ID) and performs modulo table\_size on it, and the answer is the index where the package will be stored. The ID is stored at that index in the key\_buckets list, and the data is stored at that index in the data\_buckets list.

No searching is required to access this package information again, because you simply perform the hash\_function on any ID, and you immediately know the index where key and data are stored in their respective lists. If you did not have this direct connection from the ID to the list index it is stored at, you would have to loop through the list and find hits via equality. Hash functions save a search.

## E: HASH TABLE

The insertion function, .put(), is found in HashTable.py line 32. It is described in part D1. The .put() function is called by the create\_package\_objects function in Package.py line 19. Create\_package\_objects spells out which information is passed into the .put() function. The information is parsed from the package CSV as defined by line 29:

hashmap.put(i, [line[1], line[2], line[3], line[4], line[5], line[6], line[7], truck\_num, time\_stamp,  
 status])

Indexes I – 7 are the information items given: ID, address, state, zip, due time, and weight. Truck\_num, time\_stamp, and status are added to keep track of status and delivery times.

## F: LOOK-UP FUNCTION

The hash table loop-up function is found in HashTable.py line 81. Line 82 is truly the only part used in this implementation:

start\_bucket = self.hash\_function(key, len(self.key\_buckets))

This is a recreation of the hash function. It takes the ID provided and calculates the index of the key and data in their respective lists. Line 90 checks to make sure the ID provided does == the key found in key\_buckets, and if so, returns the data in data\_buckets at this same index. The data is a list of package information.

The other work within this method, and the else: clause found at line 94 all pertain to instances where the package ID is not present, or in cases where there were key collisions when adding packages. This is extraneous for this particular use of the hash table because there are no collisions, and every bucket is full.

## G: INTERFACE

A user interface is present. When the code is run in PyCharm, it will welcome you to the program and ask for you to input a time. This time will return a list of all packages and their statuses at that time on that day. After the list of packages, the program will also print the time each of the three trucks returned to the hub, and the total mileage of the 3 trucks that day.

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

Screenshots are provided for 0900, 1000, and 1300, to fit the 3 ranges stated. See the 6 screenshots within the UniversityUPS folder.

## H: SCREENSHOTS OF CODE EXECUTION

See the 6 screenshots within the UniversityUPS folder.

## I1: STRENGTHS OF THE CHOSEN ALGORITHM

Two strengths of the nearest neighbor algorithm are ease of understanding and implementation, and the ability to scale up. This algorithm is easy to implement because there are so many examples of it, and the concept of going to the next nearest stop is easy to grasp. It can scale up well because the algorithm doesn’t grow particularly fast with added packages. Also, I noticed that the trucks I had with fewer packages weren’t necessarily much faster. Density of stops can be a good thing for this algorithm.

## I2: VERIFICATION OF ALGORITHM

Total mileage traveled is printed after the packages print out when the program is run. If you run it with a time past truck 3’s return (after 12:32) it will print 128.9 miles. All packages are delivered on time, and fulfilling the terms of the provided notes, which can clearly be seen in the packages print out.

## I3: OTHER POSSIBLE ALGORITHMS

Two alternative algorithms that can handle this type of traveling salesperson problem are 2-opt and genetic algorithms (Nilsson, n.d.).

## I3A: ALGORITHM DIFFERENCES

2-opt algorithms are an optimization of a nearest neighbor (or other simple path-finder) algorithm (Nilsson, n.d.). 2-opt takes a selection of 2 pieces of the route and switches their end points. If they entire journey is faster after this switch, the algorithm keeps the switch. If it is not, the algorithm takes a different selection of 2 journey pieces and tries again. The algorithm is complete when no further switches can be found that improve the overall run time.

The main advantage over a simple nearest neightbor is that it can be much more optimal. By definition, it is an optimization of it. The disadvantage is in run time. It is also more complicated to implement.

Genetic algorithms are an attempt to mimic natural selection in animals (Nilsson, n.d.). Random pools of potential truck routes would be generated, and then “mated” to see if the resulting route is “more fit.” Obviously, this is far more complicated to implement. You must begin by somehow generating a pool of random routes. Then, you must define how to “mate” the two parent routes. Mutations can also be added to the fittest of the routes to further optimize, which also must be defined and coded. Some implementations have been very optimal (Nilsson, n.d.), but implementations can vary widely. Any algorithm of this kind would have a much greater run time than a simple nearest-neighbor algorithm.

## J: DIFFERENT APPROACH

This program could be improved in the future by making the date changeable. Currently, the time is given by the user, but it is only running for one date: 1/22/2022. See main.py line 35, where the date is hard coded. In theory, this program could accept different dates and port in information from different CSV files as appropriate. The datetime library has a method for retrieving today’s date, and that could be utilized instead of the hardcoded one.

I would also change the criteria on which the main algorithm loops in DeliveryAlgorithm.py get\_ordered\_list(). It currently does not accept two identical addresses in lines 64 and 65 as an attempt to disallow the program from traveling to itself and neglecting to move forward. This is an issue because sometimes two separate packages have the same address. The current algorithm must visit some places two separate times to deliver different packages. An additional If: Else: could be added that linked to a separate method to deal with identical addresses.

## K1: VERIFICATION OF DATA STRUCTURE

Total mileage traveled is printed after the packages print out when the program is run. If you run it with a time past truck 3’s return (after 12:32) it will print 128.9 miles. All packages are delivered on time, and fulfilling the terms of the provided notes, which can clearly be seen in the packages print out. An efficient hash table is present, where items can be updated and retrieved directly. All delivery information is clearly listed when the program is run via the user interface, and it is accurate.

## K1A: EFFICIENCY

The discussion accurately explains how adding packages directly affects the time needed to complete the look-up function.

*The “look-up function” refers to the hash table’s look-up function identified in Part D. Describe how adding more packages affects the time it takes to retrieve package information. The effect could be nil, as in the case of a direct (1-1) mapping. Whatever the case, provide a brief explanation.*

## K1B: OVERHEAD

My hash table implementation’s size, at least in regard to the length of the lists, is constant from its creation. When the program is run, the program uses CSV reader to count the number of the packages in the provided CSV and creates a hash table that size. See Packages.py, get\_number\_of\_packages(), and how that argument is returned and used in the creation of “package\_hashtable = HashTable(number\_packages)”. The only space added as packages are added, is the package data itself. The package data itself is a list of 1 key and 10 items. See Package.py: “hashmap.put(i, [line[1], line[2], line[3], line[4], line[5], line[6], line[7], truck\_num, time\_stamp, status])”. The space taken is the base empty list + 11\*n for each package inserted.

## K1C: IMPLICATIONS

The discussion accurately explains adding trucks or cities would affect look-up time and space usage.

*Look-up and space usage are both referring to the hash-table. Depending on your code, additional cities or trucks may not affect hash-table performance. In which case, you should explain why.*

## K2: OTHER DATA STRUCTURES

The submission identifies two data structures other than the one used in [Part D](#_e5lmjqcaaava) to meet the requirements in the scenario.

*Identify two alternative data structures and justify why they could have been used as your hash-table. The alternatives can include modifications of the hash table listed in* [*Part D*](#_e5lmjqcaaava)*.*

## K2A: DATA STRUCTURES DIFFERENCES

The submission accurately describes attributes of both data structures identified in [part K2](#_g9al3fuucalf) and compares these to [Part D](#_e5lmjqcaaava)’s data structure attributes.

*Attributes and the comparison can include almost anything, e.g., mappings, structure, advantages, disadvantages, etc. The rubric writes “attribute-s.” So you should list at least two attributes per data structure from* [part K2](#_g9al3fuucalf)*.*

## L: SOURCES

AskPython. (2021, June 25). *3 ways to sort a dictionary by value in python*. AskPython. Retrieved January 26, 2022, from https://www.askpython.com/python/dictionary/sort-a-dictionary-by-value-in-python

Drowell, E. (n.d.). *Know thy complexities!* Big-O Cheat Sheet. Retrieved January 26, 2022, from https://www.bigocheatsheet.com/

GeeksforGeeks. (2020, July 28). *String alignment in Python f-string*. GeeksforGeeks. Retrieved January 26, 2022, from https://www.geeksforgeeks.org/string-alignment-in-python-f-string/

Hellman, D. (2011, July 11). *PYMOTW*. datetime – Date/time value manipulation - Python Module of the Week. Retrieved January 26, 2022, from http://pymotw.com/2/datetime/

James, J. (2016). *Python: Creating a HASHMAP using Lists*. Youtube. Retrieved January 26, 2022, from https://www.youtube.com/watch?v=9HFbhPscPU0.

James, J. (2019). *Python: Dates, Times & Timestamps Part-1 | datetime, time libraries*. Youtube. Retrieved January 26, 2022, from <https://www.youtube.com/watch?v=hUkDhkrUNys&t=459s>.

Lusby, J. (2022, January 14). C950 Outreach... Email with overall program structure tips.

Lysecky, R. (n.d.). *ZyBooks My library > C950: Data Structures and Algorithms II home > 6.6: Python: Graphs*. Building the Graph and Vertex classes Retrieved January 26, 2022, from https://learn.zybooks.com/zybook/WGUC950AY20182019/chapter/6/section/6?content\_resource\_id=29588064.

Nilsson, C. (n.d.). *Heuristics for the traveling salesman problem - free*. Link¨oping University. Retrieved January 28, 2022, from http://160592857366.free.fr/joe/ebooks/ShareData/Heuristics%20for%20the%20Traveling%20Salesman%20Problem%20By%20Christian%20Nillson.pdf

phihagphihag 257k6666 gold badges427427 silver badges455455 bronze badges, & martineaumartineau 108k2323 gold badges148148 silver badges263263 bronze badges. (1959, February 1). *How to sum all the values in a dictionary?* Stack Overflow. Retrieved January 26, 2022, from https://stackoverflow.com/questions/4880960/how-to-sum-all-the-values-in-a-dictionary

WGU C950 course documents for handling time steps and the greedy algorithm. https://srm--c.na127.visual.force.com/apex/coursearticle?Id=kA03x000001DbBGCA0. (n.d.). C950 WGUPS Project Implementation Steps - Example. https://srm--c.na127.visual.force.com/apex/coursearticle?Id=kA03x000001DbBGCA0.