Task A:

For my project, I will be using the nearest neighbor algorithm. My project will have a truck entity with a field called packages containing all the packages the truck has to deliver. The algorithm will loop through all the packages and find the nearest delivery address and update the from address with the nearest delivery address after every loop. The delivered package will be removed from the list of packages prior to the next iteration of the loop.

Task B:

A self-identifying data structure that can be used with the algorithm identified in part A to store the package data would be a modified chaining hash table (*ZyBooks*, n.d.). I decided to use this data structure for my project based on guidelines and webinars provided in the course materials (2023). The hash table is modified to store key-value pairs in a list for each bucket in the hash table. The buckets in the hash table can be accessed by their unique ID field values.

#### Task C1. ALGORITHM’S LOGIC:

#### My implementation of the nearest neighbor algorithm is implemented in two functions. The first function is min\_distance\_from. The function finds the nearest address from a list of packages on the truck from the current address. An explanation of the algorithm using pseudocode:

#### PROCEDURE min\_distance\_from(from\_address, truck\_packages, hash\_table):

from\_address\_index EQUALS index RETURN by passing in from\_address parameter to helper function get\_address\_index(street)

min\_distance EQUALS positive infinity

next\_package\_id EQUALS -1

FOR each package in truck\_packages:

package EQUALS package RETURN by searching in hash\_table with package id

package\_address\_index EQUALS index RETURN by passing in package street address into helper function get\_address\_index(street)

distance EQUALS result RETURN by calling function distance\_between(address1\_index, address2\_index) with from\_address\_index and package\_address\_index

IF distance is less than min\_distance:

UPDATE min\_distance with distance

UPDATE next\_package\_id with package id

ENDIF

ENDFOR

The second function truck\_deliver\_packages utilize min\_distance\_from to find the closest package starting from the hub, then the closest package to the found package, rinse, and repeat.

PROCEDURE truck\_deliver\_package(truck, hash\_table):

not\_delivered EQUALS empty list

FOR each package in truck\_packages:

package EQUALS package RETURN by searching in hash\_table with package id

UPDATE package departure time with truck time

APPEND package TO not\_delivered

ENDFOR

EMPTY packages in the truck

WHILE not\_delivered is not empty:

next\_package\_id EQUALS package id RETURN by calling min\_distance\_from with the truck current location and the list of undelivered packages as parameters

next\_package EQUALS package RETURN by searching in hash\_table with next\_package\_id

GET the index number for truck location in the address table by calling get\_address\_index and passing in the truck location as the parameter

GET the index number of the next package address in the address table by calling get\_address\_index and passing in the next package address as the parameter

distance EQUALS result RETURN by calling distance\_between with indexes of the truck location and next package address as parameters

APPEND package TO truck

REMOVE package FROM not\_delivered

UPDATE truck mileage TO truck mileage PLUS distance

UPDATE truck location TO next\_package street address

UPDATE truck time TO truck time PLUS distance DIVIDED by 18

UPDATE package delivery time TO truck time

UPDATE package status with time delivered

ENDWHILE

Task C2:

The programming environment I will be using is Microsoft Visual Studio Code and Python 3.10.11. The hardware I am using is Intel Core i7-13700K and 32GB DDR5 6000 MHz.

#### Task C3 - Space-Time Complexity Using Big-O Notation:

Overall program runs in O(n\*m^2). Please see comments in code for explanations.

main.py

|  |  |  |
| --- | --- | --- |
| Functions | Time Complexity | Space Complexity |
| load\_package\_data | O(n) | O(n) |
| load\_distance\_data | O(n) | O(n) |
| load\_address\_data | O(n) | O(n) |
| distance\_between | O(n) | O(n) |
| min\_distance\_from | O(n\*m) | O(1) |
| get\_address\_index | O(n) | O(n) |
| truck\_load\_packages | O(n) | O(n) |
| load\_package\_set | O(n\*m) | O(n) |
| truck\_deliver\_pacakges | O(n\*m^2) | O(n) |
| return\_to\_hub | O(n) | O(n) |
| print\_menu | O(1) | O(1) |
| print\_all\_package\_status\_and\_total\_mileage | O(1) | O(1) |
| get\_single\_package\_status\_with\_time | O(n) | O(1) |
| get\_all\_package\_status\_with\_time | O(1) | O(1) |
| convert\_user\_time | O(n) | O(1) |

HashTable.py

|  |  |  |
| --- | --- | --- |
| Functions | Time Complexity | Space Complexity |
| insert | O(n) | O(1) |
| search | O(n) | O(1) |
| remove | O(n) | O(1) |

Package.py

|  |  |  |
| --- | --- | --- |
| Function | Time Complexity | Space Complexity |
| update\_status | O(1) | O(1) |

#### Task C4 - SCALABILITY AND ADAPTABILITY:

The data structure that I chose for my project is capable to scale and adapt to a growing number of packages. Even though I initially set the hash table size to 40 since we are working with 40 packages in the project scenario, a hash table can be dynamically resized if the number of packages were to be increased (*ZyBooks*, n.d.). However, the dynamically resizing of the hash table will incur some operating overhead. As the number of packages grow, the space complexity will increase as we are using the hash table as the data structure for our algorithm. More packages mean more space must be created to accommodate the additional packages. Adding packages will increase run time but won’t affect our worse case time complexity scenario of O(n\*m^2).

#### Task C5 - SOFTWARE EFFICIENCY AND MAINTAINABILITY:

My software is easy to maintain because it approaches the solution in an object-oriented way of programming. Separate .py files are created for each entity. This promotes encapsulation, which keeps data and methods related to specific entity together, making the code organized and maintained (Janssen, 2017). Functions and variable names are also chosen to provide easy readability and understanding. Every task is designated to a specific function with comments explaining their functionality. The main program is kept short and concise. If needed, the different functions in main.py can be separated and refactored to different entities with common purposes. It is efficient in that the program performs error handling. The code includes error handling mechanisms, such as input validation and handling exceptions, which can prevent the program from crashing and provide meaningful error messages to users. The requirement for the scenario is handled by the software efficiently as all 40 packages are delivered by their requirements. Each truck had no more than 16 packages and traveled at the assume speed of 18 mph. All three truck total mileage was under 140 mph.

Task C6:

Strengths of the chosen data structure (hash table) include relatively fast time complexity for insert, search and delete functions. The hash table has an average constant time complexity and a worse case of O(n) (*Algorithm - Hash Table Runtime Complexity (Insert, Search and Delete)*, n.d.). The modified chain hash table can handle collisions. It can also be resized if needed. Though the ability to resize is also a weakness as it can incur additional programming overhead. At the same time, if the hash table size is too big for the required task, space would be wasted.

Task C7 – DATA KEY:

Regarding the choice of a key for efficient delivery management, it largely depends on the specific requirements and priorities of the delivery management system. Each component (delivery address, delivery deadline, delivery city, delivery zip code, package ID, package weight, delivery status) serves a different purpose, and the choice of a key should align with the primary goals of the system.

From the list of components to choose as a key for efficient delivery management of our program, we want a key that is unique between packages. No two packages can have the same key to guarantee each package can be identified uniquely within the system. For delivery address, deadline, city and zip code, these attributes are essential for routing and scheduling deliveries, but they may not be suitable as the primary key. Multiple packages could be going to the same address or city, and there may be variations in deadlines. Using them as keys might lead to complexity and potential performance issues. Package weight is an important attribute for load planning and logistics, but it's not suitable as a primary key. It doesn't provide a unique identifier for each package, and multiple packages can have the same weight. Delivery status is dynamic and change as packages move through the delivery process. While they are crucial for tracking and monitoring, they are not unique identifiers and may not be efficient as primary keys. Therefore, package ID would be the best choice of key for our program as package ID can be designed to be globally unique to allow for fast lookup from the hash table and scalability by simply incrementing the package ID value for each new package.

Task D - SOURCES:

*zyBooks*. (n.d.). Learn.zybooks.com. Retrieved October 28, 2023, from <https://learn.zybooks.com/zybook/WGUC950Template2023/chapter/6/section/6>

*algorithm - Hash table runtime complexity (insert, search and delete)*. (n.d.). Stack Overflow. <https://stackoverflow.com/questions/9214353/hash-table-runtime-complexity-insert-search-and-delete>

(2023). Force.com. <https://srm--c.vf.force.com/apex/coursearticle?Id=kA03x000000e1fuCAA>

*zyBooks*. (n.d.). Learn.zybooks.com. Retrieved October 31, 2023, from <https://learn.zybooks.com/zybook/WGUC950Template2023/chapter/6/section/2>

Janssen, T. (2017, November 30). *OOP Concept for Beginners: What is Encapsulation*. Stackify. https://stackify.com/oop-concept-for-beginners-what-is-encapsulation/