Stephen Wong

C950 Performance Assessment

Student ID 011031716

1. Algorithm Identification

The algorithm uses the **Nearest Neighbor/Greedy** algorithm, in the following steps, to solve the performance assessment problem.

1. Load nodes from an array.
2. Iterate through the array to find the shortest edge.
3. Return the attached node to the shortest edge.
4. Set the current location to the new node and removes it from the array of nodes.
5. Repeats steps one to four until the array is empty.

Given an array of nodes, the algorithm is self-adjusting. It searches for the closest node based on the package payload from the current location and moves there. Then it will continuously traverse through the package payload until there are no more nodes. It will also adjust for address changes to find the optimal path at the time.

B.1 Logic comments

CREATE HASH TABLE IMPLEMENTATION

CREATE ADDRESS\_MODEL

CREATE DISTANCE MATRIX

CREATE TRUCK MODEL

CREATE PACKAGE MODEL

CREATE PACKAGE LOG HASH TABLE

FOR EACH CSV FILE: (addresses, distances, packages)

OPEN FILE

CREATE OBJECT

ADD OBJECT TO HASH TABLE

CREATE 3 TRUCKS

ASSIGN PACKAGES TO EACH TRUCKS

TAKE SNAPSHOT OF PACKAGES

INSERT SNAPSHOT INTO PACKAGE LOG

WHILE VARIABLE RUNNING IS TRUE

PRINT WELCOME

PRINT MENU

GET USER INPUT

IF USER INPUT == 1

RUN TRUCK DELIVERY

WHILE TRUCKS HAVE PACKAGES

SET TRUCKS TO DELIVERING

SET PACKAGES TO EN ROUTE

FIND NEAREST PACKAGE

FOR EACH PACKAGE IN PAYLOAD, CALCULATE DISTANCE TO PACKAGE

SET NEAREST PACKAGE TO PACKAGE WITH SHORTEST DISTANCE

SET TRUCK TO NEAREST PACKAGE

DELIVER PACKAGE AT NEAREST PACKAGE

REMOVE PACKAGE FROM TRUCK

PRINT ALL PACKAGE DELIVERY TIME

ELSE IF USER INPUT == 2

PRINT ALL PACKAGE DELIVERY TIME

USER INPUT PACKAGE DELIVERY TIME

USER INPUT PACAKGE ID

PRINT PACKAGE STATUS

ELSE IF USER INPUT == 3

PRINT ALL PACAKGE DELIVERY TIME

USER INPUT PACKAGE DELIVERY TIME

PRINT ALL PACKAGE STATUS

ELSE IF USER INPUT == 4

EXIT PROGRAM

PROMPT USER TO CONTINUE

GET USER INPUT

IF USER INPUT == Y

SET RUNNING TO TRUE

ELSE

SET RUNNING TO FALSE

B.2 Development environment

IDE: Visual Studio Code Version: 1.77.3 (Universal)

Python 3.9.6

Developed on Apple M1 Pro with 16 GB Ram using macOS Ventura 13.3.1

B.3 Space-time and Big-O

Note: All major sections have been noted with space-time complexity in the code.

|  |  |  |
| --- | --- | --- |
| Section | Space-Complexity | Time-Complexity |
| Overall | O(n) | O(n^2) |
| Address DAO | O(n) | O(n) |
| Distance DAO | O(n) | O(n) |
| Package DAO | O(n) | O(n) |
| Hash map | O(n) | O(n) |
| Shortest distance  Nearest Neighbor Algorithm | O(n) | O(n) |
| Print package status | O(n) | O(n) |
| Find distance between two points | O(1) | O(n) |
| Main Truck Function | O(n) | O(n^2) |

B.4 Scalability and adaptability

The overall time complexity of the solution is O(n^2). The solution will accept any number of packages. For each truck, each time it searches and moves to the next node, it will loop to find the shortest distance. While the solution will be adequate for small data sets, a weakness is that it may need excessive resources for very large data sets.

The overall space complexity of the solution is O(n). As the dataset gets larger, additional objects and arrays will be proportionally created to accommodate the algorithm.

B.5 Software Efficiency and Maintainability

The overall solution is efficient and maintainable. The inputs needed from the user are the number of trucks, distance map, addresses, and distribution of packages. The algorithm will determine the next stops based on the shortest distance from the current location without additional inputs. It’s Big-O time complexity is O(n^2). Adjustments may be made to packages mid-route to account for special circumstances.

The code is efficiently written in the Object Orient Programming style. Thus, attributes in the models for the package, truck, and addresses may be updated accordingly. In addition, states are managed in DAO to provide context to modules as needed. Finally, utility functions are separated from the main function to reduce complexity and increase reusability.

B.6 Self-adjusting Data Structures

The hash map data structure was created to manage references to the model instances for the addresses, packages, and trucks. The hash map is instantiated with a predetermined bucket size. In each case, there are 100 buckets for each model to avoid pigeonholing the model instances. The number of buckets is user-defined during instantiation and should be more than the number of model instances.

In most situations, the **strength** is the insertion and lookup methods will be O(1) time complexity if the hashing algorithm and bucket size was implemented efficiently. A **weakness** is in the worst-case solution, that if there were to be hashing collisions, the data will be stored in a dynamic array at the determined bucket. The insertion and lookup methods will traverse through the array with at O(n) time complexity. Overall, the hash map is an efficient data structure.

C.1

main.py:



C.2

Comments example from main.py

A screen shot of a computer program

Description automatically generated with low confidence

Comments example from Hashmap.py

A screen shot of a computer program

Description automatically generated with low confidence

D.1 Self-adjusting Data Structure

The hash map data structure implemented in the solution stores the instances of the address, packages, and trucks so that they can be quickly retrieved.

For the **addresses hash map**, the key values are stored:

* Key: String of Address
* Value: Instance of Address Model with the attributes: id, name, address, city, state, zip.

Instances of the addresses may be retrieved with the string of the addresses so that the lookup of attributes such as address id is effective. The Nearest Neighbor algorithm in the function get\_shortest\_distance of calc\_utils.py inputs an address string. With the address string, an address instance is retrieved from the hash map. Of which an id is retrieved from the address that corresponds to a 2D array of distances between nodes.

For the **packages hash map**, the key values are stored:

* Key: Package ID
* Value: Instance of the Package model with the attributes: id, address, city, state, zip, deadline, weight, notes, truck, status, departure, and delivery time.

The three trucks’ package payload each consists of an array of package ids. To retrieve the package instance, the hash map’s get\_value method is called with the string of the package id.

E & F HashMap Implementation

The look-up function uses a key as a parameter through the hashing function to return an index. Then the index of the array is retrieved to return the value stored at that bucket. If there are multiple arrays in the bucket, the HashMap will traverse through the array to find the key and return the value.

A screen shot of a computer screen

Description automatically generated with low confidence

G.1 All Packages 9:08 AM

A screen shot of a computer

Description automatically generated with medium confidence

G.2 All Packages 10:00 AM

A screenshot of a computer

Description automatically generated with medium confidence

G.3 All Packages 12:09 PM

A screenshot of a computer

Description automatically generated

H. All Packages by Time.

A screen shot of a computer

Description automatically generated with medium confidence

I.1 Algorithm Strengths

The first strength of the Nearest Neighbor algorithm is its flexibility. It will scale to larger datasets and will handle unexcepted data such as the updated address for package #9. The second strength is that the math model is easy to understand and implement. It only requires simple math to determine the edges between the nodes and then determine the minimum distance.

I.2 Verification of algorithm

The Nearest Neighbor algorithm in the will:

1. Deliver all packages on time. (Given constraints)
2. Start delivery for delayed packages when received (Given constraints)
3. Adjust to updated addresses. (Package #9)
4. Deliver within 140 miles limit. (90.1 miles)
5. Meet other constraints as defined by the problem.

I.3 Algorithm differences

A different algorithm in comparison to the Nearest Neighbor Implementation is the Dijkstra. Instead of finding the nearest path to the next nearest node, Dijkstra finds the shortest path to all the nodes. The Nearest Neighbor could be significantly more efficient in time complexity, as Dijkstra is less efficient with more modes.

Another algorithm is brute force. It will find the most efficient path at the expense of the time complexity of O(n!). In a situation where there are 40 packages, there are about 40! permutations to check for the optimal path. It may be not feasible to compute the most efficient path to solve the problem.

J. Different approach

I would have stored instances of the addresses in the package instance instead of storing the address information directly in the package instance. It would have made the code more efficient by increasing reusability and giving it more flexibility to retrieve address information directly from the package model. Second, I would have liked to try a different algorithm such as the Brute Force method at the expense of being a resource hog. For my curiosity, I am interested to know the actual most efficient path.

K.1 Verification of data structure

K.1.A Efficiency

The time for the look-up function is not affected by the packages delivered. The algorithm looks up the bucket with the hash function at O(1) time. If the HashMap was not instantiated properly with enough buckets, then will a linear iteration of an array will be required to find the desired value at O(n) time.

K.1.B Overhead

Space usage for the HashMap will increase linearly to the number of packages. More data in memory will be needed to be allocated to the package instances. As packages are delivered, it will have no effect on memory allocation. The space complexity of HashMap implementation is O(n).

K.1.C Implications

Additional trucks would not affect the time usage. Regardless of the number of trucks, the algorithm will traverse through each one at O(n) and its associated packages at O(n). Combined at O(n^2). It will affect space usage, as additional instances of data structures would be created to accommodate packages.

K.2. Other data structures and differences

A **linked list** is an alternative to store the package data. Overall, most operations such as an item lookup will be at O(n). However, when inserting or deleting an element at the front of the list, it could potentially be faster than a HashMap with collisions at O(n). Otherwise, operations at the end of the list could be the same or worse than the HashMap’s lookup, delete, or insert.

An **array** is also another data structure alternative to the HashMap. It will be slightly slower when deleting an element in the middle of the list, but overall same or better at some operations than a linked list. For example, to remove an item at the end of the list, it is at a constant time of O(1) vs the linked list of O(n), simply because it does not need to traverse through all the elements to find the last element. In comparison to the HashMap, it may be slightly slower to look up elements due to the traversal operation that the HashMap mostly can avoid.

M. Citations

No citations for needed outside of the WGU-provided material.