C950 Task-1 WGUPS Algorithm Overview

(Task-1: The planning phase of the WGUPS Routing Program)

Sarah Kouhail

ID: 010604128

WGU Email: skouhai@wgu.edu

03/07/2024

C950 Data Structures and Algorithms II

# Introduction

The task at hand is to utilize self-adjusting data structures and algorithms to efficiently deliver packages with optimal mileage. We are tasked to determine an efficient delivery route for distribution given the constraints that the packages may hold. This involves determining the proper algorithms, data structures, and organization of data to find the most optimal solution.

# A. Algorithm Identification

A self-adjusting algorithm that could be used to create the program to deliver the packages would be the nearest neighbor algorithm. This algorithm can be used to repeatedly select the nearest unvisited vertex (packages that have not been delivered yet) from the current vertex (location). In turn, this can optimize the delivery route by iterating over all packages and selecting the nearest package based on the current package’s location and the distance between the rest.

# B. Data Structure Identification

A self-adjusting data structure that could be used with the nearest neighbor algorithm from above could be a HashMap.

# B1. Explanation of Data Structure

HashMaps account for the relationships between the components required for the project due to its key-value relationship. Before implementing the algorithm, a HashMap can be constructed considering the package information provided in the Excel sheets found in the task. Each package can be given a unique ID, which will act as its key, while the object itself serves as the value. Using a HashMap also allows for efficient retrieval. It provides a constant-time average-case complexity, making it efficient to quickly access package information based on a unique ID. With collision handling, it ensures that different packages with the potential to have identical hash values can exist in the same HashMap. Finally, it also allows for dynamic updating of the data loaded into it. As packages are updated (deleted, added, etc.), it can adjust its internal structure to remain efficient for data retrieval.

# C1. Algorithm’s Logic

(Nearest neighbor)

#goes through each package from the passed truck object and compares the distance of the #truck’s initial location to #all packages in the truck, continuously stores the shortest distance #that is found from the distance matrix. After the end of each iteration, the package with the #shortest distance is added to an empty list and that package is removed from the unsorted #list. Once the unsorted list is empty, it will return the sorted list.

**FUNCTION sortPackages (currentIndex, truck):**

ASSIGN unSortedPackages = [

FOR EACH package in truck:

package HashMap GET packageID

END FOR]

ASSIGN sortedPackages = [] < empty list

WHILE unSortedPackages is not empty:

ASSIGN shortestDistance = Infinity

ASSIGN nextPackage = None

FOR EACH package in unOrderedPackages:

packageIndex = GET location Index

IF currentIndex is not None and packageIndex is not None THEN

distance = GET distance

IF distance <= to shortestDistance THEN:

ASSIGN shortestDistance = distance

ASSIGN nextPackage = currentPackage

END FOR

IF nextPackage exists THEN:

ASSIGN nextPackage to sortedPackages []

REMOVE nextPackage from unSortedPackages []

currentIndex = GET next location Index

ELSE:

ENDWHILE

RETURN sortedPackages []

(Delivery process)

**Def deliverPackage (truck):**

ASSIGN currentIndex = GET truck address

ASSIGN sortedPackages = GET sortedPackages

FOR EACH package in sortedPackages:

ASSIGN package truck ID = truck object ID

ASSIGN currentIndex = GET location index of truck address

ASSIGN packageIndex = GET location index of package address

IF currentIndex is not None and packageIndex is not None:

ASSIGN distance = GET distance

ASSIGN deliveryDuration

ASSIGN truck’s miles = truck’s miles + distance

ASSIGN truck’s address = current package address

ASSIGN truck’s time = truck’s time + deliveryDuration

ASSIGN package’s departure time = truck’s departure time

ASSIGN package delivery time = current truck time

END FOR

# C2. Development Environment

**Programming environment:** Visual Studios Code v1.86.0

**OS:** Windows 11

Python version 3.12

**Hardware:**

**Memory:** 32 GB DDR5-6000 CL36 Memory

**Processor:** AMD Ryzen 5 7600X 4.7 GHz 6-Core Processor

# C3. Space and Time complexity using Big-O notation

3.  Evaluate the space-time complexity of *each* major segment of the program and the entire program using big-O notation.

* **sortPackages:**
  + Time complexity: Since the function iterates through each package in the truck passed to it to calculate the distance between them, it uses a nested loop resulting in the time complexity of O(n^2).
  + Space Complexity: The function allocates additional space for the sorted list items to be added to, causing it to grow linearly with the number of packages in the truck object passed in, resulting in the space complexity of O(n).
* **deliverPackage:**
  + Time complexity: The function calls the sortPackages function, which has the time complexity of O(n^2), then iterates through the returned sorted list of packages to deliver. This results in the time complexity of O(n^2).
  + Space complexity: This function only uses a constant amount of additional space for temporary variables when it is called. This results in the space complexity of O (1).
* **readCSV:**
  + Time Complexity: The CSV readers iterate all the rows and columns of the file. There are n rows and m columns. The time complexity would be O(n\*m) or O(n).
  + Space Complexity: The space requirements for the CSV readers are proportional to the number of rows (n) and columns (m) in the CSV, which would also be O(n) (O(n\*m)).
* **loadPackages:**
  + Time complexity: When loading each package, the function iterates over all packages from the package CSV file. The result is a linear time complexity - O(n).
  + Space complexity: Since it only creates package objects and adds them to the HashMap, which does not increase the number of packages and does not store any additional data structures, it has a space complexity of O (1).
* **createLocationDictionary:**
  + Time complexity: This function iterates over all locations in the CSV file, resulting in a linear time complexity of O(n).
  + Space Complexity: The dictionary size depends on the size of the CSV files, so it is also linear O(n).
* **getLocationIndex:**
  + Time Complexity: Getting an index from the dictionary is a constant-time operation since a key is passed. Resulting in O (1) time complexity.
  + Space Complexity: No additional space is required for this function, meaning it has an O(1) space complexity.
* **Distances getter:**
  + Time Complexity: Getting the distance from the CSV file that contains the distance is a constant time operation since it reads as a matrix. This means it has a time complexity of O(1).
  + Space Complexity: No additional space is required for this function so it is O(1) as well.

Overall, the time complexity of this project is majorly influenced by the algorithm used to sort the packages, which has the time complexity of O(n^2). As for the space complexity, it is dependent on all the data being processed and stored during execution. This includes the CSV files, HashMap usage, and other data structures used, which have O(n) space complexity.

# C4. Scalability and Adaptability

The project can scale to accommodate additional packages due to the ability to utilize data structures such as the HashMap for efficient retrieval and updates/changes to package data. Although time complexity may be affected by an increase of packages, such as in the algorithm that sorts the packages, this operation can be optimized to an operation that can handle a larger set of data.

# C5. Software Efficiency and Maintainability

The software is efficient because it utilizes modularization and encapsulation methods by separating the classes and different functionalities. It allows it to be more digestible and easier to debug. It also utilizes object-oriented programming concepts such as inheritance, polymorphism, and classes. This allows the program to be reusable, extendable, and easier to modify without causing major problems if any.

# C6. Self-Adjusting Data Structures

The strengths of the self-adjusting data structure (HashMap) include adaptability to various sizes of datasets, allowing them to scale easily with data. It also offers quicker operations such as insertion, deletion, and lookup functions since the hash function can distribute keys evenly. Some weaknesses involve the possibility of a hash collision. This can occur if more than one key hashes to the same index, which can lead to weaker performance as the number of collisions increases. Along with that, if a hash function is poorly chosen, it will also cause an uneven key distribution, which will also affect performance negatively. Another weakness is that HashMaps may consume more memory than other data structures, especially if chaining is used as a collision resolution.

# C7. Data Key

The choice of the key for delivery management would be the package ID. It is the most suitable choice for efficient look-up and the management of the package objects because it is a unique identifier and is assigned to every package. The other attributes, such as deadline, address, and status can also help organize or filter packages. They may not be the best option as the primary key, but they can be useful as secondary keys.

# D. Sources

Text goes here

An example:

Lysecky, R., & Vahid, F. (2018, June). *C950: Data Structures and Algorithms II*. zyBooks.

Retrieved March 22, 2021, from https://learn.zybooks.com/zybook/WGUC950AY20182019/

# E. Professional Communication

Nothing to write. Run your document through https://www.grammarly.com/