C950 Task-1 WGUPS Algorithm Overview

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

David Mitchell

ID # 000273705

WGU Email: dmitc19@wgu.edu

12/08/23

C950 Data Structures and Algorithms II

# Introduction

The Goal of this project is to implement a program to route WGUPS package delivery trucks in an efficient manner while meeting the needs of the clients. I will be implementing iteratively building a portion of the system and testing before moving onto the next until there is a complete deliverable product.

# A. Algorithm Identification

The self adjusting algorithm being implemented is the nearest neighbor algorithm. While it is likely not the most optimal possible solution it will strike a good balance between a globally optimal solution and reasonable time spent computing the delivery routes.

# B. Data Structure Identification

The Data structure used for storing package data will be a hash table. This data structure will serve the purpose of the application very well. Given that hash tables have a fixed lookup time they do not significantly add to the programs time complexity.

# B1. Explanation of Data Structure

A hash table takes a key value, in this case the provided package ID, creates a hash value of the key and stores the keys corresponding data in an easily retrievable table of values. The hash table can quickly be searched and updated as needed using the key value to find the relevant data. In the event of a data collision the hash table will use a chaining technique allowing multiple values to be stored in the same bucket as a list of values.

# C1. Algorithm’s Logic

PackagesTable = read from csv

DistanceTable = read from csv

LocationsTable = read from csv

Sort Packages into an array by delivery deadline using python sort

deadlineList = []

function sortPackagesByDeadline():

for i in range(PackageTable.size)

get packageID

get address

get location from locationTable using address to find matching location

get deadline

listObject = (ID, address, location, Deadline)

append list object to deadlineList

sort deadlineList by deadline

return deadlineList

Place Packages onto trucks placing 20 packages per truck to ensure even distribution

some packages have already been placed per instruction included with packages

truck1List = [13, 14, 15, 16, 19, 20]

truck2List = [3, 18, 36, 38]

truck3List = [6, 9, 25, 28, 32]

packagesPlacedOnTrucksList = [3, 6, 9, 13, 14, 15, 16, 18, 19, 20, 25, 28, 32, 36, 38]

packagesToBeDeliveredTogetherList = [13, 14, 15, 16, 19, 20]

optimalRouteList = []

function loadTrucks():

deadlineList = sortPackagesByDeadline()

for package in deadlineList:

if package id not in packagesPlaced in Trucks:

if truck1List.Len < 20:

truck1List.append(packageID)

packagesPlacedOnTrucks.append(packageID)

if truck2List.Len < 20:

truck2List.append(packageID)

packagesPlacedOnTrucks.append(packageID)

Algorithm to sort packages on trucks using next nearest neighbor algorithm. This is accomplished with 4 independent functions. A function that looks up the distance between two provided locations, one locates the location closest to the hub, one locates the next nearest location, and finally a master function that coordinates them for sorting the packages according to next nearest neighbor.

The function to get the distances between two locations takes the names of the locations as arguments and returns the distance between them from the distances table.

Function getDistance(location1, location2):

x = look up index of location1 in locations table

y = look up index of location2 in locations table

distance = information in distance table at (x,y)

The function to retrieve the location nearest to the hub takes a truck list as an argument. Looks up the distances between each location and the hub, returning the location with the shortest distance.

function GetNearestLocationToHub(list):

PackageDistanceList = []

clear packageDistanceList = []

for packages in truckList

getDistance(Hub, package)

append package with distance to PackageDistanceList

Locate minimum distance from PackageDistanceList

return packages with smallest distance

the function to Locate the next stop takes one location as an argument as well as the list currently being sorted, calls the get distance function for the given location all packages excluding the given package and any packages already added to the optimal route list. The function returns the package with the shortest distance.

Function GetNextStop(list, location):

packageDistanceList = []

for packages in list:

if location not given location and location not on optimalRouteList:

getDistance(location, package location)

append package with distance to packageDistanceList

Locate minimum distance from PackageDistanceList

Return packages with smallest distance

the function to fully sort the truck lists utilizes the above described functions to return a sorted list following the next nearest neighbor algorithm

function sortPackagesByDistance(list):

addressToAppend = “”

for packages in list:

firstStop = getNearestLocationToHub(list)

if Length of optimalRouteList = 0:

append first stop to optimalRouteList

elseif optimal route >=1

addressToAppend = getNextStop(list, firstStop)

append addressToAppend to optimalRouteList

firstStop = addressToAppend

return optimalRouteList

# C2. Development Environment

The WGUPS Routing application was written in Python 3.11.6-1 on an Arch Linux machine with a Ryzen 9 CPU and 64 gigabytes of ram. Arch Linux uses a rolling release strategy and as such does not have versioning; an update was performed 1 day prior to this writing. The text editor Neovim version 0.9.2 was used alongside a number of plugins that emulate the functions found in an IDE such Pycharm. The plugins and their purpose are as follows:

* Autoclose.nvim – Automatically adds the closing bracket when openting bracket is typed.
* Conform.nvim – Automatically formats whitespace in accordance with style guides for python.
* Lazy-Lsp.nvim – Automatically installs LSP servers for coding languages according to opened file type.
* Lsp-config.nvim – Configurations for LSP behavior.
* Surround.nvim – Provides shortcuts for automatically commenting out lines or blocks of code.
* Treesitter.nvim – Provides syntax highlighting functionality.

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

The space and time complexity of each major section of the code is as follows:

* the algorithm that sorts the packages by deadline has a complexity of O(n^2) due to appeding the location from the address table for later use in the Nearest Neighbor algorithm. The space complexity of this function is 0(n) as the size of the package list is proportional to the space needed for the final sort.
* loading the packages that have notes requiring specific trucks or onto the trucks has a complexity of O(log n). The space complexity of this function O(1) as it does not use any additional space to perform.
* The algorithm to load the remaining packages has the same complexity at 0(log n). The space complexity of this function O(1) as it does not use any additional space to perform.
* The nearest neighbor algorithm has a complexity of O(n^2). The space complexity of this function is 0(n) as the space used is proportional to the size of the input.

# C4. Scalability and Adaptability

The program is adaptable, it would not take a lot of time or effort to adjust the program to changing business needs or the needs of other similar businesses. The nearest neighbor algorithm may become a problem on sufficiently large data sets and may need to be changed. The Hash table is easily adjusted to the size of the inputs, and the implemented chaining function reduces problems caused by key collisions as the business grows. The function to load the trucks could easily be adapted to automatically adjust to the number of trucks or the number of packages per truck should either requirement change.

# C5. Software Efficiency and Maintainability

The software is likely not at its optimal efficiency but there no egregiously slow portions of code, and there could be efficiency gains in future updates. The most complex portion of the code base is the Nearest Neighbor algorithm with a time complexity of O(n^2), the rest of the code is a linear or better complexity, which meets general efficiency requirements and would only need improvement on especially large data sets. The code is relatively small as such it should be easy to maintain given good code documentation. The code was also written with an eye towards readability above the inherent readability of python, I aim to choose function and variable names that spoke to their use in the code base wherever possible. The code base is also thoroughly commented with a comment block in main.py explicitly explaining the flow of the program and its functions.

# C6. Self-Adjusting Data Structures

Hash tables have a few strengths and weaknesses. The importance of those vary based on the specific application of the table. One advantage of a hash table is that it has a constant look up time for any data in the table. Another advantage being its ability to store large amounts of data. There are not many disadvantages of hash tables but they are important to consider. One example being data collisions which can be costly to resolve in terms of processing time and should be considered ahead of time. Another disadvantage is that hash tables are not ordered which can be a problem for some applications.

# C7. Data Key

I chose to use the packages ID as the key for my data as it is a unique identifier for the packages. The Package ID key is also easily expanded upon in future revisions of the software if necessary to account for growing needs of the business.

# D. Sources

No Sources Used