C950 Task-1 WGUPS Algorithm Overview

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

Logan Tackett

ID #012308366

WGU Email: ltacke12@wgu.edu

11/12/2024

C950 Data Structures and Algorithms II

# Introduction

The project entails writing a program that will deliver 40 packages to different locations such that the total miles driven by the delivery trucks is less than 140. Different packages have other requirements, such as deadlines, required groupings, or delays. We are provided a list of packages with all of their information and a pairwise complete distance table. There are three trucks, but only two can be driving at once, and each truck can hold at most 16 packages.

I plan to approach this problem by first manually loading packages with specific requirements onto the trucks. There are certain packages that only make sense to go on certain trucks, this includes packages that must be grouped together and packages that are required to be on a specific truck. I will sort the remaining packages by their deadline (earliest to latest) and load them onto the trucks in sequence. Optimizing the delivery route for this loading order will solve the problem. Packages that have the same delay of 9:05 are put onto the second truck, which will leave at 9:05. The package with a missing address is treated as a delayed package. It will be loaded onto the third truck and will only be delivered after 10:20 when the correct address is received.

Since the order of packages in a truck's package list will be sorted in optimal delivery order, the delivery function will be rather simple. The delivery function will simply deliver packages in the order they are listed in their container. After each package is delivered, the mileage and current time will be updated. The truck with the wrong address package will check the time after every package delivery; if the current time is past 10:20, the wrong address package will be updated with the new address, put into the package list, and the package list will be re-sorted for optimal delivery given the new information.

Package information will be stored in a hash table, which I will implement. The hash table will take a package ID and a package object as input; it will insert the package object into the table with the ID as its key in the table. There will also be a lookup function that all other parts of the program will use to access package data given the package's ID.

I noticed when looking at the distance table that while it is pairwise complete, it is actually not a shortest-paths table. This, to me, indicates that I should use Floyd Warshall's all-pairs shortest path algorithm, as it's easy to implement, and the shortest paths table only needs to be calculated once at the start of the program. It will be easiest to keep packages sorted in order of deadline; there are certainly more optimal routes that deliver packages with later deadlines sooner. However, more computation is required to verify that these routes satisfy all deadline conditions. An optimal delivery route for a list of packages will first sort packages by their deadline. Then, for each group of packages with the same deadline, we will sort those in optimal delivery order using a self-adjusting algorithm of choice. The first package of the next deadline group will be the package in that group closest to the last package in the previous deadline group.

# A. Algorithm Identification

My self-adjusting algorithm of choice will be the nearest neighbor algorithm. It is a very simple algorithm and produces orders that are near optimal for delivery. Its runtime is O(n^2), which is much better than more sophisticated algorithms.

# B. Data Structure Identification

My data structure of choice will be a hash table, as stated in the introduction. I chose this data structure as it has constant time lookups. Insertions are also constant time, with the exception being when a resize is needed.

# B1. Explanation of Data Structure

The hash table will utilize the Package class that I created to group package data together. The insertPackage function will take in a package ID and package information, create a package object with that information, and insert it into the table with the key being the ID. If there is already a package in the hash table at that index, it will be updated with the new package object. The hash table will use direct hashing, so the hashing function will simply subtract one from the ID; if the hash key is larger than the last index in the hash table's array, resizes will be done until there is room for that ID. A resize will simply double the size of the array. The lookup function will take in an ID and return the package object in the table if one exists for that ID; if not, it will return false.

# C1. Algorithm’s Logic

The delivery function is trivial, as packages are delivered in the order that they are stored in the truck's package array, and truck loading is also trivial as it is mostly casework and then loading packages with earlier deadlines onto the truck(s) that leave earlier. The main component/core algorithm of my program is the algorithm that sorts packages in order of deadline and then in optimal delivery order using nearest neighbor, so that is what I will be delving into here.

The core algorithm finds a solution to the problem by first sorting packages in order of deadline, going from earliest to latest. This ensures that packages with earlier deadlines are delivered first; that way, they are delivered on time. This sorting will be done with the standard merge sort algorithm. This is easy as DateTime objects are comparable using < and > operations. Then, for each group of packages that have the same deadline, we sort them in optimal delivery order using nearest neighbor. The first (earliest) group of packages will assume that the truck is starting at the HUB and will be sorted such that package1 is closest to HUB, package2 is closest to package1, and so on. The second group of packages will be sorted the same way, except the starting location for the truck will be the location where the last package in group 1 was delivered. This process will continue for each group of packages with the same deadline until all groups are sorted.

Pseudocode:

function sortPackages (package\_list):

// merge sort is standard, simply compares packages by their deadline

merge\_sort(package\_list,0,package\_list.length,deadline)

left = 0

right = 0

curr\_deadline = package\_list[left].deadline

curr\_location = “HUB”

while left <= right and right < package\_list.length:

if package\_list[right].deadline == curr\_deadline:

right = right + 1

else:

nearest\_neighbor\_sort(package\_list,left,right,curr\_location)

left = right

curr\_time = package\_list[left].time

function nearest\_neighbor\_sort (package\_list,left,right,curr\_location):

sorted\_array = []

// copy package\_list subarray starting at left (inclusive) to right (exclusive)

copy = package\_list.copy[left:right]

curr = curr\_location

while copy.length > 0:

nearest = find\_nearest\_neighbor(curr,package\_list)

curr\_location = nearest

sorted\_array.push(nearest)

copy.remove(nearest)

// copy sorted elements back into package\_list

for i = 0, i < sorted\_array.length, i++:

package\_list[left + i] = sorted\_array[i]

function find\_nearest\_neighbor (curr\_package,package\_list):

min = None

min\_dist = infinity

for package in package\_list:

curr\_dist = distance(curr\_package,package)

if curr\_dist < min\_dist:

min = package

min\_dist = curr\_dist

return min

# C2. Development Environment

I will be using VSCode on my Windows 11 computer to develop my program. While PyCharm is recommended, and I have used PyCharm in the past, I am simply more comfortable using VSCode.

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

The main parts of my program are the hash table, package class, truck class, loading trucks, all-pairs shortest distance table, sorting packages, and delivering packages.

**Hash Table:**

**Insert:** Best case O(1) time complexity, Worst case O(n) time complexity (doubling the size of the array), always O(1) space complexity.

**Lookup:** Always O(1) time complexity and space complexity.

The hash table itself stores items in memory, and thus no matter what subprocess is being run, the hash table’s creation is O(n) time and takes up O(n) space perpetually.

**Package and Truck Classes:**

The package class is used to store properties of packages and thus has O(1) space complexity and O(1) runtime complexity (no class methods). The truck class will have a space complexity of O(1) as well as a truck can hold at most 16 package ids in its storage. Its methods will have varying time complexities.

**Loading Trucks:**

Loading trucks will have O(n log n) time complexity, as packages are sorted using merge sort before being loaded onto trucks one by one, which is O(n). The space complexity will be O(n) as the merge sort's space complexity is O(n), and we won't need any extra storage besides the auxiliary array, which would be the list of packages.

**All-Pairs Shortest Distance Table:**

Floyd Warshall's algorithm has a runtime of O(V^3) where V is the number of vertices or in this case, the number of different delivery locations. This is because this algorithm is effectively running Dijkstra's algorithm for every delivery location and storing that information in a matrix. The space complexity of his algorithm is O(V^2) as the algorithm stores each pair of destinations shortest distance in a matrix, and there are V^2 such pairs.

**Sorting Packages:**

Sorting packages has two parts: the initial merge sort and the multiple nearest neighbor sort.

The merge sort, as stated previously, has a time complexity of O(n log n) and a space complexity of O(n) for the input. The nearest neighbor sort algorithm is O(n^2) time complexity in the worst case and has a space complexity of O(n) as it stores a copy of the input array. It is clear to see that the time complexity is O(n^2): First note that for each package we have to find its nearest neighbor, and finding the nearest neighbor is done in O(n) time. Thus, the time complexity for sorting n packages using nearest neighbor is O(n^2). Let n be the number of packages in the package\_list, then let's assume that there are n\*p packages in each deadline group where p is some percentage. Then, the time complexity would be

O(1/p \* (np)^2 ) = O(p \* n^2). Assuming p is a constant, we can reduce the previous expression to O(n^2), whereas if p = 1/n, the time complexity would be O(n) (best case).

Combining merge sorts complexities with nearest neighbor sorts complexities, we see that the best case runtime for the algorithm as a whole is O(n log n), the worst case (and average) is O(n^2), and the space complexity is always O(n).

**Delivering Packages:**

As packages are delivered in the order they are stored in the package\_list after sorting, the time complexity and space complexity is O(n), as we are only doing constant time operations for each package.

**Whole Program:**

The time complexity of the program as a whole in the worst case is O(V^3 + n^2), where V is the number of delivery locations and n is the number of packages. The space complexity of the program is O(V^2 + n).

# C4. Scalability and Adaptability

The program is scalable and able to adapt to an increasing number of packages due to its use of nearest neighbor and hash table. Both of these algorithms/data structures are self-adjusting such that they will work for any number of packages without modification or refactoring. The same applies to merge sort, which is also a core component of my algorithm. Compare this to the loading function, which may require modification if new packages with different requirements are introduced to the problem. The loading function can handle packages with no notes in a scalable way, but packages with special requirements may need to be implemented into the function. The loading function is rather simple, so adapting it to these new types of packages should be easy.

# C5. Software Efficiency and Maintainability

The program is efficient as it runs in polynomial time. More specifically, it runs in O(V^3+n^2) time, where V is the number of destinations and n is the number of packages. The O(V^3) might not seem ideal, but it is important to remember that V is the number of destinations; when scaling this application, that calculation only needs to be done once. One could even do the destination array calculation once and store that improved table in a CSV file to be used in future program executions. This would completely get rid of the O(V^3) runtime until a new location needs to be added, to which the distance array can be updated with that new location in O(V^2) time.

The program will be easy to maintain/modify through good code organization and commenting. For example, the functions used in sorting, like merge sort and nearest neighbor sort, will be stored in a separate Python file and imported, making it easy to change the functions used to sort items. The truck class and package classes will both have their own Python files, and functions pertaining to multiple trucks, like delivering packages, will also be in their own file. Good naming conventions, abstraction, and well-written, concise comments will also make it easy for people besides the author to understand the flow of the main program or what a complicated function is doing.

# C6. Self-Adjusting Data Structures

The self-adjusting data structure used in my program will be a hash table. The strengths of a hash table include its fast operations, where insertions and lookups are both O(1) constant time. The main weakness of a hash table is the space complexity, as they have a fixed size until they need to be resized. While this will not be an issue for this problem, as packages in the package list have IDs 1,2,3…, and 40, in a scenario where packages can have varying IDs, the hash table implementation being used here would have to change, as doubling the size of the hash table for a single item insertion takes up a lot of excess memory. This would require the use of collision resolution, and this increases the complexity of the hash table implementation.

# C7. Data Key

The key for packages in the hash table will be the package ID. This is the obvious choice as it is the only property of the package that is inherently unique. Let's say that two packages had the same ID; we at least know that this is due to some error in package processing; the same cannot be said about any of the other parameters. When choosing a key to identify items, it's important that the key be unique across items.

# D. Sources

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

Retrieved November 12, 2024, from <https://learn.zybooks.com/zybook/WGUC950AY20182019/>

*CSV - csv file reading and writing*. Python documentation. (n.d.). https://docs.python.org/3/library/csv.html

Retrieved November 12, 2024 from

<https://docs.python.org/3/library/csv.html>

Wikimedia Foundation. (2024, August 15). *2-opt*. Wikipedia. https://en.wikipedia.org/wiki/2-opt

Retrieved November 12, 2024 from

<https://en.wikipedia.org/wiki/2-opt>

*Nearest neighbour algorithm* (2024) *Wikipedia*. Available at: https://en.wikipedia.org/wiki/Nearest\_neighbour\_algorithm.

Retrieved November 12, 2024 from

<https://en.wikipedia.org/wiki/Nearest_neighbour_algorithm>

# E. Professional Communication

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