C950 Project – WGUPS Delivery Router

Mario Silvestri III – 000941631 – Western Governors University

# A – Algorithm Selection

The WGUPS program utilizes a nearest neighbor algorithm to order the delivery of packages. The algorithm will take a starting location and loop through each package to find the next package location, repeating until all packages have been ordered. Additionally, packages are given a priority of high, medium, or low, based on their delivery time constraints. Therefore, each priority level will be separately ordered using the algorithm and then combined so that high priority is delivered first, followed by medium, then low.

# B – Program Overview

## Algorithm Logic (wgups.py, nearest\_neighbor, line 95; generate\_route, line 57)

Nearest Neighbor pseudocode:

def nearest\_neighbor(packages\_list, start\_location, start\_time):

remaining\_packages = packages\_list

current\_location = start\_location

current\_time = start\_time

route: tuple(location, package, distance) = []

if len(remaining\_packages) == 0:

final\_route = [(current\_location, none, 0)]

return final\_route, current\_time

while (len(packages\_list) > 0):

next\_package = none

for each package in packages\_list:

if package.available\_time and package.available\_time > current\_time:

continue

if next\_package is none:

next\_package = package

continue

if current\_location == package.location:

package.delivered\_time = current\_time

route.append(package.location, package.id, 0)

continue

if distance(current\_location, package.location)

< distance(current\_location, next\_package.location):

next\_package = package

distance = distance(current\_location, package.location)

current\_time = current\_time + time(distance, speed)

next\_package.delivered\_time = current\_time

route.append(next\_package.location, next\_package, distance)

current\_location = next\_package.location

remaining\_packages.remove(next\_package)

return route, current\_time

The algorithm takes as parameters a list of packages, a start location, and a start time.

def nearest\_neighbor(packages\_list, start\_location, start\_time):

Starting variables are initialized for use throughout the function.

remaining\_packages = packages\_list

current\_location = start\_location

current\_time = start\_time

next\_location = none

final route: tuple(location, package, distance) = []

As mentioned in Section A, the algorithm runs once for each priority level. The statement below handles the condition of a priority level having no packages.

if len(remaining\_packages) == 0:

final\_route = [(current\_location, none, 0)]

return final\_route, current\_time

Next, the program enters the main body of the algorithm. This is the WHILE loop that cycles through the list of packages, removing one with each cycle and adding it to the route, until there are no packages left. The algorithm finds the package whose destination location is closest to the current\_location. This is done by the inner FOR loop that cycles through packages while storing the current closest package in the next\_package variable.

while (len(packages\_list) > 0):

next\_package = none

for package in packages\_list:

The first conditional statement inside the FOR loop checks to make sure the package is available to be delivered. This is for packages like #9; Package #9 is loaded on the truck, but due to constraints, cannot be delivered until new information is provided at 10:20. Therefore, the package is set to available at 10:20, and the algorithm will skip over a package if the current\_time variable is earlier than the package’s available time:

if package.available\_time and package.available\_time > current\_time:

continue

The first available package will be set to next\_package and then the FOR loop continues to the next passage; This happens once per WHILE to get a distance to compare to.

if next\_package is none:

next\_package = package

continue

The next conditional is an optimization that continues to the next loop if a package is found that is at the current\_location. That package is added to the route at the current location and removed from the package list.

if current\_location == package.location:

package.delivered\_time = current\_time

route.append(package.location, package.id, 0)

continue

The last conditional in the while loop is where the main functionality of the algorithm occurs. The distance of current\_location and next\_package.location is compared to current\_location and package.location. If package.location is closer, then next\_package is reassigned to package.

if distance(current\_location, package.location)

< distance(current\_location, next\_package.location):

next\_package = package

After each iteration of the WHILE loop, variables are updated to the next point in the route, and the next package is added to the route.

distance = distance(current\_location, package.location) # distance from current\_location to package.location

current\_time = current\_time + time(distance, speed) # current time + travel time to next location

next\_package.delivered\_time = current\_time

route.append(next\_package.location, next\_package, distance)

current\_location = next\_package.location

remaining\_packages.remove(next\_package)

return route, current\_time

The nearest neighbor algorithm is encapsulated in generate\_route (wgups.py, line 57). The generate route function runs the algorithm separately for each priority level, concatenating the results. The priority level was implemented to make sure packages with earlier delivery constraints could simply be tagged with a priority level, allowing the algorithm could satisfy delivery time constraints The generate route function also has one optimization; it will automatically bump packages up to a higher priority level if it higher level contains a package with the same destination, to ensure this strategy doesn’t route to a location more than once.

## Programing Environment

IDE: PyCharm Professional 2020.3

Python Version: 3.9

OS: Windows 10 Education, 10.0.19042

Processor: Intel i7-3930K @ 3.20 GHz, 6 cores, 12 logical processors

RAM: 16 GB

GPU: NVIDIA GeForce GTX 1060 6GB

## Space-Time Complexity

The space and time complexity of all major functions, classes, and statements are notated in comments throughout the program.

## Solution Scalability & Adaptability

Scalability

Time Complexity O(N^2), where n = number of packages :

The nearest neighbor algorithm binds the program to an upper-bound time efficiency of O(N^2), because for each package must be routed, all other remaining packages must be considered. The number of packages to consider is reduced by 1 for each package that has already been routed, so a slightly more accurate upper bound would be O(N^2/2).

Space Complexity O(N^2), n = number locations:

The space complexity of the program is upper bounded by the number of that must be handled. Each additional location adds N more distances to space requirements.

The system complexity is upper bound to an N^2 on time and space. Given that it is necessary for the algorithm to loop through packages and during each loop check all other packages, as well as the fact that the location data must be stored for reference, this algorithm should be considered reasonably scalable.

Adaptability

The solution was written in an object oriented and modular manner and with an abstract system for assigning package priorities and running the algorithm without intervention. Given the reasonable space and time complexities of the algorithm, and the flexible program, this solution would be adaptable as required to scale the application to sorting more packages or being used in other cities.

## Software Efficiency and Maintainability

Efficiency

The main goal of the software is to complete a route in under 140 miles. The software was also written with the goal of being automated and flexible. The nearest neighbor algorithm with priority levels allows the algorithm to adapt to package delivery time constraints by assigning priority levels to packages. Additionally, packages have an available\_time variable which provides more flexibility to when a package can be delivered. These high-level goals were implemented to reduce manual input during the dispatching process and adaptability.

Maintainability

The software is written in an object-oriented and modular manner with separate classes for different objects, data structures, and utilities. The WGUPS class integrates these classes together, with the intention being that the modularity creates a separation of concerns and ease of understanding and modifying the program.

## Strengths & Weakness of the Self-adjusting Data Structure

Advantages:

* The chaining hash table allows for O(1) look-up, insert, and delete time when there are no bucket collisions.
* A hash table will scale with the number of packages. Only the bucket size needs to be adjusted to minimize collisions.

Disadvantages

* Hash Tables and linked lists would be more space intensive than a basic list because hash tables must implement buckets using lists and possibly nodes, creating a more space complex data structure.
* Hash Table collisions need to be managed. The hash table size either needs to be adjusted to limit collisions, or hash table lookup function time complexity will start to approach O(N) as the number of collisions approaches the number of packages in the hash table.

# C – The Program

# 

# D – Self-Adjusting Data Structure (chaining\_hash\_table.py)

A chaining hash table class with linked list and node classes for buckets is fully implemented. See program.

## Explain how the data structure accounts for the relationship between the stored data points

A hash table requires a unique key and value pair for each object. Each package has its own set of data values including a unique package ID. Since the package id is unique, it is selected for the key, while the package object containing all package data will represent the value. Therefore, the data structure accounts for the relationship between the stored data points using the package id as the hash map key to insert and lookup packages, and the package object as the hash map value that is stored and retrieved.

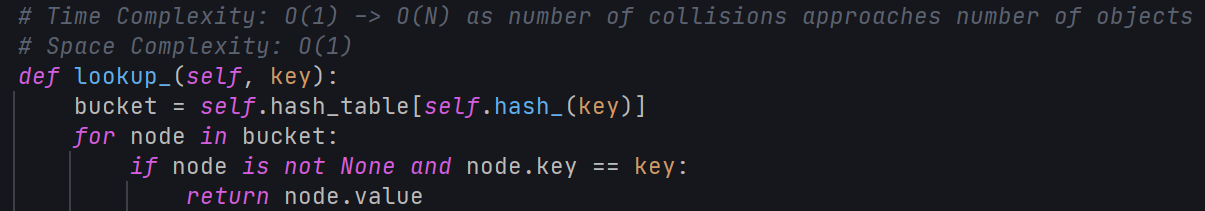
The chaining hash table is used as the central repository for packages. When the program is run, package data is parsed from csv\_reader.py (line 40) and appended into a simple list of packages. Upon starting the program and generating a route, the WGUPS class (wgups.py) instantiates a chaining hash table (chaining\_hash\_table.py) class and calls the chaining hash table insert function on each package, using the package’s id as the key, instantiating the hash table.

# E – Hash-Table (chaining\_hash\_table.py, line 72)

The hash table is instantiated and populated with package objects in wgups.py, lines 20-22. The package id is used as the key and the full package data including id, address, city, zip, weight, notes, and delivery status are included in the object.

# F – Look-up Function (chaining\_hash\_table.py, line 120)

The look-up function takes a package id as a key value argument. The key is hashed (default hash: key % 10) to find the correct bucket. The object is returned, and if there are multiple objects in the bucket, the linked list is traversed until the object with the matching package id is found.



# G – Interface

Screenshots inserted at the end of the document.

## Status of all packages between 8:35am and 9:25am.

## Status of all packages between 9:35am and 10:25am.

## Status of all packages between 12:03pm and 1:12pm.

# H – Successful Code Completion and Total Mileage Traveled by All Trucks

Total miles traveled: 126.8

Last package (23) delivered at 12:02pm



# I – Core Algorithm Justification

## Two Algorithm Strengths

* Nearest neighbor is just about the simplest algorithm for traveling salesman type problems, so it is easy to understand, implement, and use as a benchmark against more complex algorithms. This makes them a great starting point and baseline for solving Traveling Salesman-like problems. This concept is referred to as a naïve algorithm.
* Nearest neighbor algorithms are computationally reasonable, at O(N^2).
* Nearest neighbor, while far from optimal, will find a reasonable solution for how simple the algorithm is.

## Verify Algorithm Meets All Requirements

* Total miles: 126.8 (See screenshot in section H)
* All delivery time requirements are met – see screenshots at end of document
* Algorithm time and space complexity are of N^2 polynomial time, which is efficient and scalable.

## Two Alternative Algorithms

* Pure Greedy Algorithm
* Genetic Algorithm

### Differences between core and alternative algorithms

Pure Greedy Algorithm

The pure greedy algorithm is like the nearest neighbor algorithm in that the shortest distances are used to construct a sequence. However, the greedy algorithm doesn’t have a starting location; The greedy algorithm will find the two closest cities and create a connection between them. Then, it will find the next closest connection, whether it is associated with one of the first two cities or not and connect them but making sure each city is visited only once. This process repeats until all cities have been connected.

Genetic Algorithm

A genetic algorithm is not a naïve algorithm; Its design and implementation are somewhat more complex, inspired from biology and evolution. A genetic algorithm is also classified as a meta-heuristic, a sort of algorithm that develops its own heuristic. After trial-and-error processes, the most successful variants of the algorithm propagate forward and combine, while failures are left behind.

# J – What would I do differently

* Implement multiple algorithms that can be chosen from (‘Strategy’ software design pattern)
* Clean up my codebase and simplify the program wherever possible, to make it more understandable for another programmer.
* Explore other data structures for storing location data and distance data, such as a tree and matrix structures.
* Continue making the solution more systematic and automated. The goal would be to have no manual loading of the trucks; everything from start to finish would be algorithmic. The delivery time constraint could be encoded into the program and a priority level automatically assigned. For the notes section, natural language would be difficult to parse, but there are common constraints and patterns there that could be written into the program. At that point a human can simply enter the constraints into the program and let the program do the rest.

# K – Justify the data structure from part D (chaining\_hash\_table.py)

## Verify the data structure meets all requirements

### Explain how the time needed to complete the look-up function is affected by changes in the number of packages to be delivered.

The lookup function is affected by collisions in the chaining hash table. If the number of buckets is greater than the number of packages, and hash values are evenly distributed across buckets, then time complexity of the lookup function would be approaching O(1). However, as the number of packages increase, if the number of buckets does not increase as well, eventually there will start to be collisions. As the number of collisions approaches the total number of packages, due to not enough buckets or uneven hash distribution, the lookup function time approaches O(N).

### Explain how the data structure space usage is affected by changes in the number of packages to be delivered.

For each delivered package, the space required increases linearly, as the only additional requirements are the data for that one package. Package data space complexity is O(N).

### Describe how changes to the number of trucks or the number of cities would affect the look-up time and the space usage of the data structure.

The lookup time is not affected at all if the number of trucks or cities increase. The data structure is used for packages only; trucks and locations are held in their own basic lists. However, if trucks or locations were stored in hash table, the time complexity scale the same as a package as explained in Section K.1.a: O(N). It is of note, however, that locations do contain distance data to each other location. So, any data structure holding locations would increase in space complexity on the order of N^2, because each additional location N must contain the distance information of N – 1 other distances.

## Identify two other data structures that could meet the same requirements in this scenario.

* A list data structure would serve well in this scenario. Package IDs could be directly mapped to index values of a list.
* A distance matrix could be used for storing distances between locations.

### Describe how each data structure identified in part K2 is different from the data structure used in the solution.

List

A list will have longer delete and insert times than a chaining hash table, with a worst-case approaching O(N) more often than a hash table. This is because the list needs to be rebuilt when an object is removed or inserted. However, the implementation details would be simpler to implement and understand, as there is no need for hash functions, buckets, or key/value pairs.

Distance Matrix

In the solution, each location has a location ID that is associated with its list index in the list of locations. Each location holds the distance to all other locations that have an id lower than itself. This ensures that data isn’t repeated and provides a simple solution to contain distance data and lookup distances without needing to create a new data structure or object for distances. However, the next step in the programs data model would be to abstract distance data from locations for more efficient data manipulation and storage. A distance matrix would do a good job isolating distance data in a way that the data can be passed into algorithms or stored.

# L – Sources

