Author: Joshua Sizer

Student ID: 001216849

A.  Identify a named self-adjusting algorithm (e.g., “Nearest Neighbor algorithm,” “Greedy algorithm”) that you used to create your program to deliver the packages.

The algorithm that I used to create my program to deliver the packages was a Nearest Neighbor Algorithm.

B.  Write an overview of your program, in which you do the following:

1.  Explain the algorithm’s logic using pseudocode.

Function get\_package\_status:

Input: graph, start\_vertex, packages, end\_time

Output: packages, total\_distance

current\_time = 8:00 AM

current\_vertex = start\_vertex

graph.trucks.append(Truck(1))

graph.trucks.append(Truck(2))

arrival\_event\_q = PriorityQueue()

arrival\_event\_q.put(ArrivalEvent(current\_time, graph.trucks[0], current\_vertex)

// add 1 minute to ensure that truck 2 is loaded after truck 1

arrival\_event\_q.put(ArrivalEvent(current\_time + 1 minute, graph.trucks[1], current\_vertex)

truck\_load = 0

while arrival\_event\_q is not empty and (end\_time is “EOD” or current\_time < end\_time):

current\_arrival\_event = arrival\_event\_q.get()

if current\_arrival\_event.current\_vertex is the start\_vertex:

packages = load\_truck\_manually(current\_arrival\_event, packages, truck\_load)

truck\_load += 1

else:

delivered\_packages.extend(deliver\_packages(current\_arrival\_event))

traveled\_distance, next\_event = generate\_next\_arrival\_event(current\_arrival\_event, graph, start\_vertex, packages)

total\_distance += traveled\_distance

if next\_event is not None:

arrival\_event\_q.put(next\_event)

all\_packages = []

all\_packages.extend(delivered\_packages)

all\_packages.extend(packages)

for each truck in graph:

for each package on the truck:

all\_packages.append(package)

return all\_packages, total\_distance

This function requires two additional types: ArrivalEvent and Truck. ArrivalEvent is a container for the current time, the truck in question, and the vertex the truck is currently at. Truck is a container holding a HashTable of packages and the Truck ID.

This function also requires the additional functions: load\_truck\_manually, deliver\_packages, and generate\_next\_arrival\_event. In essence, this function runs a simulation of the day, adding simulated events to the arrival\_event\_q and consuming the event with the lowest time at each iteration.

Function load\_truck\_manually:

Input: current\_arrival\_event, packages, truck\_load

Output: remaining packages

If current\_arrival\_event.time > 10:20 am

Change package 9 address

// packages without restrictions delivered to the same address as those with restrictions

// are loaded together

If truck\_load is 0:

Load all the packages with “must be delivered with” special instruction

Else if truck\_load is 1:

Load packages that must be on truck 2

Load packages that have early deadline and are not delayed on flight

Else if truck\_load is 2:

Load packages that were delayed on flight

Else if truck\_load is 3:

Load packages with no condition

Else if truck\_load is 4:

Load package 9

Remove all packages loaded onto truck from the packages list

Set all packages on the truck to have status “en route”

Return packages

Function deliver\_packages:

Input: current\_arrival\_event

Output: a list of packages that were delivered

For each package on the current\_arrival\_event’s truck at the current\_arrival\_event’s current\_vertex:

Remove the package(s) from the truck

Mark the package(s) as delivered

Set the package’s delivery time to the current\_arrival\_event’s time

Set the package’s delivered\_on with the truck’s id

Return each delivered package

Function generate\_next\_arrival\_event

Input: current\_arrival\_event, graph, start\_vertex, packages\_left

Output: the trucks next arrival event and the distance that the truck travels between now and the next arrival event

If the current\_arrival\_event’s truck does not need to deliver any more packages,

If the truck is not at the start\_vertex:

Return distance\_to\_hub and an ArrivalEvent where the truck is at the hub

Else if the truck is at the start\_vertex:

If there are still packages left

Return 0 distance and another ArrivalEvent 5 minutes into the future at the hub (truck needs to wait for packages to become available)

Else:

Return 0 distance and None because the truck’s day is over

Create a set of urgent and non-urgent packages based on delivery\_deadline

If the urgent\_set is non-zero length

Choose to travel to the address that is closest by distance in urgent\_set

Return distance to vertex (address), ArrivalEvent at that vertex at the time the truck will arrive there

Else:

Choose to travel to the address that is the closest by distance in the non\_urgent\_set

Return distance to vertex (address), ArrivalEvent at that vertex at the time the truck will arrive there

2.  Describe the programming environment you used to create the Python application.

The programming environment I used to create the Python application was that of

Visual Studio Code. I used Python 3.6.7 and the Visual Studio Code plugins Python and Visual Studio IntelliCode. To run the program, I used the command line.

The hardware used to develop this program is a Desktop PC with Windows 10, 32 gb of RAM, Intel i7-8700K, and 256gb NVME SSD.

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

|  |  |  |  |
| --- | --- | --- | --- |
| Segment | Big-O Time | Big-O Space | Comments |
| Loading packages from csv | O(p) | O(p) | p is # of packages |
| Loading places from csv | O(n) | O(n) | n is # of places |
| Loading graph with places and distances | O(n) | O(n2) | Time is kept linear through use of HashTable. # of graph weights to store is order of n2 |
| Function load\_truck\_manually | O(p) | O(p) | At most p packages are loaded onto a truck. A HashTable and extra package list is created with space O(p). |
| Function deliver\_packages | O(p) | O(1) | At most p packages are delivered at once. No extra space is used in this function. |
| Function generate\_next\_arrival\_event | O(plog(p)) | O(p) | Time of plog(p) comes from the sort used. Another 2 arrays are created of size O(p). |
| Function get\_package\_status | O(p2log(p) + n) | O(p + n2) | Each function load\_truck, deliver\_packages, and generate\_next\_arrival\_event gets called at least p times, with any excess coming from truck waiting times which is at maximum a constant, so p \* max(O(plog(p)), O(p)) is O(p2log(p)). We cannot forget the O(n) time because it is a different variable. Total space is some multiplicative value of p plus n2 |
| Total | O(p2log(p) + n) | O(p + n2) | The function get\_package\_status is called at most once per program |

4.  Explain the capability of your solution to scale and adapt to a growing number of packages.

The capability of my solution to scale and adapt to more packages is poor in one area and good in another. It is poor because the truck loading process was done by hand. The function “load\_truck” is commented out and can load an arbitrary number of packages onto a truck, but in this case resulted in a total distance traveled of around 180 miles. To maintain the criteria that the trucks travel a total of less than 140 miles, I decided to manually sort the packages onto each truck load. On the other hand, the part of the algorithm that delivers packages can be scaled very effectively. It is easy to add or remove trucks, and any number of packages can be delivered once loaded onto a truck.

5.  Discuss why the software is efficient and easy to maintain.

The software I produced is efficient and easy to maintain because of its code and data structure. The flow of the program is broken down at the highest level:

1. Load data
2. Process data
3. Display processed data

Each segment is sufficiently separate to make changing any layer possible without much refactoring.

In addition, the process data step is broken down further into:

1. Load truck
2. Deliver packages
3. Generate next arrival event

This allows for any part of the process data algorithm to be adjusted independent of the others. Apart from code structure, the data is contained within logical constructs, like Truck, ArrivalEvent, Package, Place, HashTable, and Graph. Changes to each construct is easy if needed, and data integrity is handled within each construct to make interactions easy and reliable.

6.  Discuss the strengths and weaknesses of the self-adjusting data structures (e.g., the hash table).

What makes the HashTable self-adjusting is its ability to resize the internal array when needed. The strength of the HashTable is that it has an O(1) lookup time on average given an arbitrary index (key) type. It also has O(1) inserts and removes on average. A weakness of the HashTable is that these average Big-Oh times are dependent on good hash functions and management of load-factor (amount of used space vs unused space). Improper management of these factors can rid the HashTable of the O(1) time complexities.

D.  Identify a self-adjusting data structure, such as a hash table, that can be used with the algorithm identified in part A to store the package data.

1.  Explain how your data structure accounts for the relationship between the data points you are storing.

What makes a HashTable self-adjusting is its ability to resize the internal array when needed.

A key ability needed for this program is to be able to look up a package by its address. For example, in the deliver\_packages function, the truck’s current location (address) is used to lookup a list of packages needing to be delivered at that location. The use of the HashTable makes this lookup O(1) and not O(n).

G.  Provide an interface for the user to view the status and info (as listed in part F) of any package at any time, and the total mileage traveled by all trucks. (The delivery status should report the package as at the hub, en route, or delivered. Delivery status must include the time.)

1.  Provide screenshots to show the status of all packages at a time between 8:35 a.m. and 9:25 a.m.



2.  Provide screenshots to show the status of all packages at a time between 9:35 a.m. and 10:25 a.m.



3.  Provide screenshots to show the status of all packages at a time between 12:03 p.m. and 1:12 p.m.



These screenshots are available under the folder “screenshots” in my submission.

H.  Provide a screenshot or screenshots showing successful completion of the code, free from runtime errors or warnings, that includes the total mileage traveled by all trucks.





These screenshots are available under the folder “screenshots” in my submission.

E.  Develop a hash table, without using any additional libraries or classes, that has an insertion function that takes the following components as input and inserts the components into the hash table:

•   package ID number

•   delivery address

•   delivery deadline

•   delivery city

•   delivery zip code

•   package weight

•   delivery status (e.g., delivered, en route)

I developed a HashTable and it is in the file hash\_table.py. It can be given an arbitrary key or value. In the program, the way that the components listed above are added is through a Package object. The package object contains all the components listed above and is the value that is stored in the HashTable. The key can be any one of those components, although in my program I used Package ID and delivery address most frequently.

F.  Develop a look-up function that takes the following components as input and returns the corresponding data elements:

•   package ID number

•   delivery address

•   delivery deadline

•   delivery city

•   delivery zip code

•   package weight

•   delivery status (i.e., “at the hub,” “en route,” or “delivered”), including the delivery time

My HashTable get(key) function provides this functionality. You can use any one of the components as keys, and the returned value is a Package, holding the rest of the corresponding data elements.

I.  Justify the core algorithm you identified in part A and used in the solution by doing the following:

1.  Describe at least **two** strengths of the algorithm used in the solution.

One strength of the nearest neighbor algorithm identified in Part A is that no preplanning is required. Trucks make the decision of where to go next at the time that they need to figure out where to go next in the simulation. In this way, it is easy to understand how the algorithm operates and how to modify it. Another strength is that any heuristic imaginable can be used for determining which packages to deliver next. My algorithm uses the heuristic of delivering the closest and most urgent packages first, but any number of other heuristics could be used.

2.  Verify that the algorithm used in the solution meets all requirements in the scenario.

All packages are delivered before their deadlines. Packages that require a specific truck are loaded onto a specific truck. Packages that must be delivered on the same truck are loaded at the same time. Packages are loaded onto trucks only after they have arrived (late arrival special note). You can query the status of any package at any time.

3.  Identify **two** other named algorithms, different from the algorithm implemented in the solution, that would meet the requirements in the scenario.

Two other algorithms that could meet the requirements are a brute-force solution, as well as the Ant Colony system.

1. Describe how each algorithm identified in part I3 is different from the algorithm used in the solution.

The brute-force solution is different than my algorithm because it searches the entire space of possible routes, given a set of packages, and finds the optimal path for the given set of packages. My algorithm, on the other hand, searches only one possible path given a few heuristics.

The Ant Colony system is different than my algorithm because it models the path for a set of packages as a trail that ants follow. As ants are sent out to each vertex in the graph, they leave a “pheromone” on the edge. The next vertex they visit is a combination of the edge weight and the amount of pheromone on the edge. The goal is to have the shortest paths accumulate more pheromones than longer paths, thus providing a close-to-optimal solution (Travelling Salesman Problem, 2021). This is different from my algorithm because it calculates the optimal path before traversing it. My algorithm traverses a path at the same time it is figuring out what vertex to visit next.

J.  Describe what you would do differently, other than the two algorithms identified in I3, if you did this project again.

One thing I would do differently for this project is create a more general system for allocating drivers and trucks. The current solution for this project only loads packages onto 2 of the 3 trucks since the bottleneck is the two drivers. This is because there is no advantage to pre-loading a truck (“The delivery and loading times are instantaneous, i.e., no time passes while at a delivery or when moving packages to a truck at the hub”). If I had implemented a way for all 3 trucks to be loaded, while allowing the drivers to switch trucks, my solution would be more scalable. This is especially true if in the future we stopped including loading times in the average speed of the truck.

K.  Justify the data structure you identified in part D by doing the following:

1.  Verify that the data structure used in the solution meets all requirements in the scenario.

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

The time to look up a package by any of its attributes does not change even as the number of packages to be delivered increases. Given a sufficient implementation, the average case time to look up is O(1).

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

The space usage of the HashTable increases proportionally (linearly) with the number of packages to be delivered. In other words, it is O(n)

1. 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.

A change to the number of trucks would not affect the look up time or space usage of the HashTable. If you need to find what truck a specific package is on, you could maintain a separate HashTable with keys being the package id or address and the value being the truck that is holding it. Then, the package can be retrieved from the truck in a similar fashion. A change to the number of cities would also not affect the look-up time or space usage of the data structure. Again, another HashTable could be created where the key is the city and the value is a HashTable of trucks. The space usage would be equal to the number of cities, also known as O(n).

2.  Identify **two** other data structures that could meet the same requirements in the scenario.

One other data structure that could meet the same requirements is a list. Another data-structure could be a binary-search-tree.

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

A list is different than a HashTable because its keys are always integers. Since this is the case, retrieving an item from a list by any attribute other than its index requires an O(n) search.

A binary-search-tree is different from a HashTable because its underlying structure is a tree and not a list. In a balanced binary search tree, the lookup time is always log(n) instead of a HashTable’s average O(1).

L.  Acknowledge sources, using in-text citations and references, for content that is quoted, paraphrased, or summarized.

The information on the Ant-Colony system comes from Wikipedia:

Works Cited:

Travelling Salesman Problem. (2021, February 14). Retrieved March 09, 2021, from https://en.wikipedia.org/wiki/Travelling\_salesman\_problem#Computing\_a\_solution