**Core Algorithm Overview**

**Stated Problem:**

The Western Governors University Parcel Service (WGUPS) needs a program to assist them in determining the best route and delivery distribution for all of their daily deliveries. There has been an issue with consistently delivering packages by their deadlines. There are three trucks, two drivers, and an average of 40 packages that need to be delivered each day. Each package can have a special note containing specific requirements for that package, and a delivery deadline that needs to be met.

The task is to write a program using the best algorithm that will ensure all 40 packages are delivered on time, meeting their specified unique requirements, and doing so in the least number of miles being added to the trucks. All package details and locations are provided along with distances in the two csv files located with the program.

**Assumptions:**

* Each truck can carry a maximum of 16 packages.
* Trucks travel at an average speed of 18 miles per hour.
* Trucks have a “infinite amount of gas” with no need to stop.
* Each driver stays with the same truck as long as that truck is in service.
* Drivers leave the hub at 8:00 a.m., with the truck loaded, and can return to the hub for packages if needed. The day ends when all 40 packages have been delivered.
* Delivery time is instantaneous, i.e., no time passes while at a delivery (that time is factored into the average speed of the trucks).
* There is up to one special note for each package.
* The wrong delivery address for package #9, *Third District Juvenile Court*, will be corrected at 10:20 a.m. The correct address is 410 S State St., Salt Lake City, UT 84111.
* The package ID is unique; there are no collisions.
* No further assumptions exist or are allowed.

**Algorithm Overview:**

The program has several classes created to assist in managing the data.

* Package objects contain all of the package details for that specific package
* Truck objects represent the three trucks available to be used to deliver packages
* AddressNode objects are the vertexes in the graph containing address details
* Graph object contains adjacency list and edge weights
* Hashtable object is the data structure created to store the package data
* Simulation object contains the other objects and preloads data

The first thing the program does after instantiating the simulation object is to import all of the packages from the packages.csv file. Each line gets parsed into a package object that is inserted into the direct hash data structure hashtable object. Decided to use a direct access table since there are only 40 packages. The benefit is that there are no collisions. There are limitations to using a direct access table hash such as the size of the table is equal to the largest key plus one which could be very large depending on the use case. Fortunately, in the case of 40 packages there is an enormous amount of room to scale without cause for concern using the direct access table hash.

**import\_packages()**

open the packages.csv file

skip header

go through and parse each line in the file storing in package variable

insert each package into the hashtable data structure

space-time complexity ***O(n)*** due to running through each line in the csv file

Next is to import all of the addresses and distances. Modified the WGUPS Distance Table Excel file into columns that specify the from address and the to address along with the distance between the two undirected points. Parse each line and check if the address already has a node created and if not then create one and add it to the graph. Once the from and to nodes have been added then the undirected edge and distance can be added.

**import\_distances()**

open the distances.csv file

skip header

go through each line and create from and to address node objects

add undirected edge weights

space-time complexity ***O(n)*** due to running through each line in the csv file

Part of the scope of the project is to follow the requirements of the special notes for the packages. To help ensure each note is handled properly this function sorts the packages into groups based on what the special note is requiring. Special notes include packages that must be delivered with other packages, packages that can only be delivered by truck # 2, packages that are delayed until a certain time, and a package that has the wrong address that will not be corrected until a later time. Two of the groups are loaded directly to the trucks at this stage, and the other two groups are separated into lists that are held in the simulation object until later when the trucks need to reload after delivering all the packages initially loaded onto the truck. This method would need to be adjusted if there were to be new requirements included in future packages.

**handle\_special\_notes()**

loop through each package in hashtable data structure

put package in group based on the special note

space-time complexity ***O(n)*** due to running through each package

After sorting out all packages that have a special note into the four separate groups it makes sense to go through and see if any of the remaining packages that do not have special notes have the same address as any of the packages that do have special notes. Then it would save time to deliver all packages going to the same address at the same time instead of having to make multiple trips to the same address.

**load\_packages\_with\_same\_address()**

loop through each package in hashtable data structure

check the delivery status to see if package has yet to be loaded into a package group

loop through each package already loaded and compare addresses

if addresses are the same then load the new package to same location as other package

space-time complexity ***O(n2)*** due to multiple loops

Now the packages with the special notes and the other packages going to those same addresses have already been loaded into groups. The next priority is to get all the packages that have the early morning deadline and get those loaded onto the first two trucks so they can be sure to meet the deadline. With careful review of the packages given for this project it was found that there was no issue with capacity on the trucks even though they can only hold up to 16 packages each. However, if there is an increase in packages in the future that have early deadlines then modifications would need to be made and possibly include hiring another driver to drive the third truck.

**load\_deadlines()**

loop through each package in hashtable data structure

check the delivery status to see if package has yet to be loaded into a package group

if package has a deadline then load it onto a truck

space-time complexity ***O(n)*** due to running through each package

Run the **load\_packages\_with\_same\_address()** method again to get all the other packages that are going to the same places into the same groups.

Finally, need to go through all remaining packages and load those into groups as well. Thought it would make sense to put the remaining packages in same groups for those with similar zipcodes.

**load\_final\_packages()**

loop through each package in hashtable data structure

check the delivery status to see if package has yet to be loaded into a package group

check the zipcode for each package and put similar zipcodes into same groups

after all packages have been loaded into a group run methods to get distances

call **get\_shortest\_path()** to calculate shortest distance and predecessors

call **list\_sort\_distance()** to sort each package list by least distance ascending

mark each trucks location at the hub

space-time complexity ***O(n2)*** due to multiple loops

The **get\_shortest\_path()** method is a modified version of the Dijkstra’s shortest path algorithm for graphs that visits each vertex comparing the distances and updating the distance and predecessor attributes for the path that is the shortest. The idea here is that for each delivery starting with the hub to calculate the least distance for the package with the next closest address. After each delivery to recalculate the shortest distance with the last delivery location as the starting vertex.

**get\_shortest\_path()**

create an unvisited queue to store each node

loop through each vertex in the graph’s adjacency list

append each vertex to the queue and set distance to infinity and predecessor to None

loop through the unvisited queue to find vertex with lowest distance and pop

go through adjacency list for that vertex and calculate distances and predecessors

space-time complexity ***O(n2)*** due to multiple loops

The program will always run the **list\_sort\_distance()** method immediately after running the **get\_shortest\_path()** method. This is because the remaining packages in the queue now have updated distances based on the new starting point which is the location of the last package that was delivered. Then the package that is closed to that location will be the next package to be delivered and named as the truck’s destination.

**list\_sort\_distance()**

sort list by distance attribute ascending

space-time complexity ***O(n)*** due to the sort of the length of the list

The main program handles calling all of the afore mentioned methods in the first line by instantiating the simulation object which has a call to everything in the init method. The next thing is a while loop that shows the main menu to the user.

The **show\_menu()** accepts the user input in the form of an integer in the range of 1 – 5. Option 1 will simulate the entire day which basically delivers all packages. Option 2 allows the user to just simulate an hour at a time. Option 3 is to lookup a package from the hashtable data structure by package ID to get all the package details including the current delivery status. Option 4 will provide a full report of all truck’s mileage and each package’s delivery status. Then option 5 will exit the simulation.

**show\_menu()**

print statements that create the main menu

space-time complexity ***O(1)*** due to all print statements run in constant time

Both options 1 and 2 call the **run\_simulation(duration)** method. With option 1 adding 9 hours for a full days simulation (8am – 5pm). The method checks to make sure that the time doesn’t exceed 5pm. Option 2 will just increment the time by one hour each call. This method would need adjustment to the for loop range if the company added more drivers and trucks as it currently is set for the two trucks since there are only two drivers. Within each truck loop the

**prepare\_next\_destination()** method calls the **get\_shortest\_path()** method and the **list\_sort\_distance()** method to prep the next destination. Then the **set\_truck\_destination()** is called to update the truck’s destination and status properties before printing to the console the truck status. At this point since the distance to the next destination is known and the truck drives 18 mph we can calculate how much time it will take to arrive at the next destination by calling the **get\_time()** method. If the truck can arrive to the destination before the simulation time then it is ok to proceed with delivering the package by calling the **deliver\_package()** method which handles all the necessary updates of the delivery. After the loop the truck’s mileage is displayed to the console.

**run\_simulation(duration)**

set the simulation time based on the added duration (9 or 1) not to exceed 5pm

loop through each of the two trucks in use (only 2 drivers)

call **prepare\_next\_destination()**

call **set\_truck\_destination()**

call **get\_time()**

if the truck’s time hasn’t exceeded the new simulation time then call **deliver\_package()**

print truck mileage

space-time complexity ***O(n)*** due to the while loop that goes through N packages

The **prepare\_next\_destination()** method just calls the previously explained **get\_shortest\_path()** method and the **list\_sort\_distance()** method to prep the next destination.

**prepare\_next\_destination()**

call **get\_shortest\_path()**

call **list\_sort\_distance()**

space-time complexity ***O(n2)*** due to **get\_shortest\_path()** method

The **set\_truck\_destination()** is called to update the truck’s destination and status properties before printing to the console the truck status.

**set\_truck\_destination()**

set truck destination property

set truck status property

print truck status property

space-time complexity ***O(1)*** due to all statements run in constant time

The **get\_time()** method is used to calculate the time it will take to get to the next destination. If that time added to the current truck time is less than the simulation time then it is ok to proceed with delivery and we can update the truck time. A backup time is created for the case where the truck would not be able to reach the destination before the simulation time so then it would need to revert back to the time before the attempt.

**get\_time()**

set backup time equal to the current truck time

calculate the time it will take to get to the next destination

if the minutes are greater than 59 then we need to increase the hour of the time object

set new truck time

space-time complexity ***O(1)*** due to all statements run in constant time

Once it is confirmed that there is time to reach the next destination prior to the end of the current simulation period then the delivery is set to take place by calling the **deliver\_package()** function. This function updates the package delivery status, updates the trucks mileage, prints to console that the package was delivered, pops the package from the queue and adds it to the delivered list, updates the truck’s last stop property, and finally checks if that was the last package. If it’s the last package then need to head back to the hub and see if there are more packages to pick up and deliver.

**deliver\_package()**

update package status

update truck mileage

print to console package delivery information

remove package from truck’s list of packages to deliver

add package to the truck’s list of delivered packages

update the truck’s last stop property

check if there are any packages left to deliver

if there are no packages left update destination mileage for trip back to hub

space-time complexity ***O(1)*** due to all statements run in constant time

Option 3 is for the package lookup. The user is asked to input the package id number for lookup in the packages hashtable data structure. If the package id is invalid the user will be prompted again for a correct id number. Once there is a valid id number the package details are retrieved and displayed to the console. This option has a space-time complexity ***O(1)*** due to all statements run in constant time.

When the user selects option 4 from the main menu they will get a full report of all trucks and packages. The program starts with a for loop through each of the trucks and printing out the truck number, the number of packages delivered, the total miles for that truck, and the current truck status. After that there is a loop through the package hashtable data structure that displays each package and all of the package’s properties. This option has a space-time complexity ***O(n2)*** due to multiple loops.

To exit the simulation the user can select option 5 from the main menu. This just prints an exit statement and raises SystemExit to end the program. This option has a space-time complexity ***O(1)*** due to all statements run in constant time.

This program’s algorithms were designed to ensure that all of the special note requirements were met along with meeting each of the package deadlines. This meets the project scope established by the WGUPS delivery company. By utilizing the undirected edge weight distances between locations and determining shortest paths the truck routes created were highly competitive. Loading packages going to the same address in the same truck was another strength of this algorithm since that eliminated having to take multiple trips to the same location.

A brute force algorithm could have been used for this task to meet the basic requirements of the special notes and the deadline deliveries, however, that choice would have ultimately led to a larger total of miles driven and ultimately taken longer to prepare as it would be necessary for manual intervention to ensure the deadlines and special note requirements were met. Additionally, a binary search tree algorithm could be implemented for the project which would be great for fast lookups, although in the case of this project with only 40 packages it makes more sense to me to simplify things by using a direct access table hash as even if the number of packages increased 100 X that would be no problem for this hashtable. While keeping the insert, remove, and search functions at a simplified level.

Looking back if I were to do the project again and keeping in mind the potential for future scaling and increase in packages I think I would want to implement some measures to ensure more balance between the trucks in the number of packages and also try to group packages from same zipcodes together if possible dependent on the special note requirements.

Using the direct access table hash for this project was ideal due to the low number of packages. I would feel comfortable using it even up to a million packages. Direct hashing has search, insert, and removal in ***O(1)*** which is highly efficient and there are no collisions. Ramifications of increase in packages is simply a larger table.

Binary Search Trees and AVL Trees are both alternative data structures that could be used. While they are both trees the AVL is a type of BST that rebalances itself making it faster in lookups with worst case ***O(logn)*** while BST worst case is ***O(n).*** The main differences between using the Trees vs the Graph I chose is that with Trees the path is one between each vertex and with graphs you can have multiple paths, even loops and circuits.