# QuickShip Overview

Author: Michael Bros

### Stated Problem:

Western Governors University Parcel Service (WGUPS) has not consistently delivered packages on time. This project aims to write a program with a self-adjusting algorithm to find an optimal delivery route according to the project constraints (see below section ‘Constraints’).

### Constraints:

1. Three trucks and two drivers are available for deliveries; each driver stays with the same truck for as long as the truck is in service.
2. Each truck may only carry a maximum of 16 packages.
3. Drivers may leave the hub no earlier than 8:00 am.
4. There may be up to one special note associated with a given package.
5. The correct delivery address for package #9 is not known until 10:20 am.
6. Each package must be delivered by its specified deadline.
7. The day ends when all packages have been delivered.
8. Trucks must complete their day with a total mileage of under 140.

### Development Environment:

QuickShip was developed in Python 3.10 with the PyCharm Professional IDE on a Windows 11 desktop PC.

### Program Overview:

The QuickShip program is highly adaptable and maintainable by separating functionality from implementation. A day is set up and run in the application's main method using the RouteService class. The RouteService provides methods to interact with all of the program's objects via other services and models. HTML documentation for all models and services can be found in the QuickShip/docs/build/index.html.

The intention is to use RouteService to 'set up the world' and then manipulate the pertinent objects via methods in RouteService. When the RouteService is initialized, all other required services are created, and data from distance.csv and packages.csv are ingested. The initialization is only meant to provide a default option; most of the program's base functioning can be changed by manipulating the attributes in RouteService.

After using RouteService to simulate an entire day of delivering packages, the truck models will have the data history needed to see all packages delivered, route lengths, delivery times, delivery status, and any other model attributes available. Using this data, we can update/refresh the RouteService->PackageService; from there, we can now efficiently lookup and return package models.

#### Adaptability

If this application is to scale to more packages, the main limitation will be the initial assignment/seeding. Currently, RouteService contains several seeding attributes that are lists of package IDs. Each attribute is responsible for one truck and one trip. Currently, there are attributes for two trucks and two trips. There are also a few areas of the code that use hard-coded values that would need to be updated; for example, if the package list increases by x, then the check to see that all packages are delivered would raise an exception since 40 + x ≠ 40.

To add more trucks or more trips will require adding additional attributes. However, if more trucks or trips are not needed, then it is possible to write over the initialized attributes with new lists of package IDs. This will scale without any drawbacks as the primary optimization algorithm will handle the current max load of 16 packages without modification, and the package hash table used for lookups after simulation will also adjust, only changing the current time complexity by a constant and not affecting current Big-O.

#### Maintainability

The most complex operation performed in the application is the optimization of a given route, which, as discussed below in the section 'Primary Algorithm Overview-> Time Complexity' is O(n!). Interaction with the program's objects via RouteService mostly happens at O(1), and the package lookup function utilizes a chaining hash table to allow lookups with worst case O(n) and best case O(1).

The program utilizes a familiar-looking object-oriented design pattern that is inspired by the popular MVC pattern using models and services to extract abstract functionality. This allows for easy additions, say in the case a new model or service needs to be created. Because they are compartmentalized, whole models or services could potentially be replaced/modified without the need to modify any other classes.

The codebase is also already set up with sphinx documentation. However, it would require a developer some level of initial environment setup to regenerate or add documentation. Currently, every class and method in the models or services are fully documented with docstrings. Most of the major methods are also commented with logical steps to provide easier readability.

#### Efficiency

The primary data structure for package lookups is a chaining hash table. This hash table, in particular, has a default of 10 buckets and uses lists to store each bucket. As more packages are added to the data structure, the more inefficient the lookups become. There is still a chance that adding a package and a subsequent lookup would be done in O(1), but it is more likely O(n). This particular implementation uses lists as the base data structure and is therefore bound by the space requirements of lists in python, meaning a space complexity of O(n) where n is the number of items.

On the other hand, adding additional cities or trucks would not increase the time or space complexity of the hash table. That is because the hash table specifically works with Package objects. Adding additional cities or trucks may affect a package's attributes but would not add complexity.

#### Self-Adjusting Data Structure / Explanation of Data Structure

The only self-adjusting data structure in this program is a hash table that is used to store packages for efficient lookups. Specifically, the data structure is a chaining hash table with a default of ten buckets. Each package is stored in the data structure as a key-value pair where the key is the package ID, and the value is the Package object.

A strength of this data structure is the excellent best case lookup complexity of O(1). One disadvantage of using the hash table is that we cannot use operations like permutations on it. The primary algorithm used for optimizing routes is not compatible with a table structure; it has to be a list.

The second strength specific to chaining hash tables is that the table can adjust to store any number of items without the need to adjust the number of buckets. This can go both ways; however, a major disadvantage of chaining hash tables is the reliance on a list structure to hold multiple items in a single bucket. This reliance on a list data structure means that even for our program with only 40 packages, each bucket by default will have ten items and a lookup operation of O(1) is not as likely as it being O(n) where n is the size of the bucket list.

#### Other Data Structures

One possible alternative to the chaining hash table that is implemented would be a modification to the existing structure. Currently, the limiting factor of the table is the reliance on lists; if the buckets used linked lists, then functionality would be improved. This is because a linked list has more efficient sorting and insertion operations. This would theoretically decrease the time to search in a given bucket as the insertions could be made so that the list stays sorted by key value. Although the average case would improve with this modification, the overall time and space complexity would remain the same.

Another possibility would be some type of tree structure like a binary search tree. A binary search tree has a complexity of O(log n), which would be more than good enough for this program even at mid-scale. This might have been a better data structure to use for this particular implementation because traversing all nodes in a binary search tree is quite easy once the data structure is set up correctly, whereas a chaining hash table could become unwieldy at scale.

#### Data Structure Differences

Compared to the current implementation, a chaining hash table with a linked list structure would be nearly identical. The overall structure is identical, and the primary difference is in the data insertion. During the insertion of data currently, the table appends the package to the bucket list, but the insertion of data into a table with a linked list would allow low complexity sorted insertion.

A binary search tree, on the other hand, would be an entirely different structure. Instead of packages being stored within buckets, they would be added to the tree as nodes. This could require quite a bit of refactoring to implement. In addition to that disadvantage, the lookup efficiency would not improve.

#### Different Approach

If I were to remake this program again, I would replace the command line-based UI with a full GUI. Python has a standard library called Tkinter that is quite powerful and lightweight. Providing a graphical interface would greatly improve the user experience and usability. If I were also to implement a different algorithm that utilized a graph data type, a graphical interface would also open the door to graph visualization.

In order to do this, the main class would need to be rewritten quite extensively. The other services and models would also need to be modified, and methods that currently print to the command line would need to be replaced with their graphical equivalents.

### Primary Algorithm Overview:

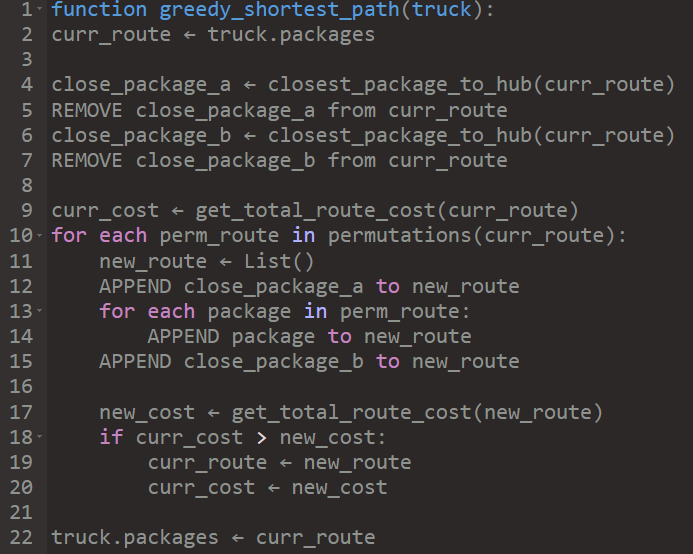
The primary algorithm is within a method named greedy\_shortest\_path, which is contained in the DistanceService class. This algorithm uses manageable-sized lists of permutations to find an optimized path assuming the start and end positions are both always the hub.

#### Detailed Steps

1. If the given current route length is two or less, no optimization can be done.
2. If the given current route length is three, then there may be an optimization step.
3. Find the two packages closest to the hub in the route, arbitrarily assign one package as the first delivery and the other as the last.
4. If the given current route length is greater than three but less than nine, there is almost certainly optimization that can be done.
5. Find the two packages closest to the hub in the route, arbitrarily assign one package A and package B, and take both of these out of the current route.
6. For each permutation of the current route, there is a potential route; add A and B to the start and end of the permutation and then calculate the total route cost. If the cost is the current best, then the potential route becomes the current route to be assigned to the truck.
7. If the given route is more than nine, then the above optimization step will take too long to complete.
8. Same as step 3.a find the two packages closest to the hub in the route, arbitrarily assign one package A and package B, and take both of these out of the current route.
9. Split the current route into two more manageable-sized lists, list A and list B.
10. Optimize list A using logic from step 3.b, except this time, only add package A to the beginning of the route before calculations; do not add package B to the end.
11. Find and remove the package in half B closest to the last package in optimized half A; this will be the first package in half B.
12. Optimize list B using logic from step 3.b except that the package A that would be added to the start of the route before calculating cost should be the package found in the previous step.
13. Both halves now being optimized are now combined to make the current route assigned to the truck.

#### Core Pseudocode

Depending on the size of the list as described above this pseudocode is modified in the actual implementation to find an optimized route.



#### Time Complexity

The core algorithm uses permutations of a list of packages and then calculates each permutation cost. The algorithm uses the python List data structure and only uses list operations that are within O(n). The calculation step is also done with list manipulations that are within O(n). Most of these operations are within the permutations iterations. The time complexity of the iteration step is O(n!), where n is the length of the permutation being calculated.

There is no other section of the program that has a worse Big-O than O(n!) from the permutations step, meaning the overall program complexity is O(n!).

When considering time complexity for this algorithm, it is also useful to acknowledge the constraints of the input. Because the input to this method is a list of packages, we know that we are capped at a max length of 16. It is also true that before checking any permutations for any length > 2 will remove the start and end packages from the list, meaning our actual max length is 14. That max length is still quite large in terms of n!, therefore, for any lists that have a length > 8, the list is split, and permutations are only calculated for a max length of n/2 or 7 since n=14 at most.

#### Strengths

Most important to this project is that delivery routes can be optimized efficiently and that the code is easy to understand and maintainable. This algorithm accomplishes that by carefully considering key connection points and the number of items in a given route to adjust. By accounting for the start and end goal being the hub, the algorithm pulling the two closest packages and placing them at either end of the route means the path taken regardless of packages in between is circular. In this implementation, I went even further to make sure that when splitting a route and optimizing the two halves separately, the connection point between each half is optimal.

This algorithm uses permutations that, while not the most efficient possible algorithm, is straightforward. This means that another developer would be able to understand and maintain the code. It is also implemented in such a way that changing the algorithm at a later date would be easy and not require modification to more than the RouteService and DistanceService.

#### Verification

* Total mileage traveled by all trucks:
* All packages were delivered before their deadline
* All packages were delivered to their delivery specifications
* The lookup function is implemented with a hash-table in PackageService
* Package statuses and information can be verified through the UI

In the UI, when the simulated day has finished, there will be a lot of output on the command line. At the end of this output is a representation of every package with its ID, status, deadline, and delivery time, among a couple of other properties. Below this, there will be the output from several checks. If any of the above rules and any of the constraints listed in the ‘Constraints’ section are violated, the program will raise an exception and will stop running. If the program continues to run, then all checks have passed. The lookup function is used several times in the simulation setup, but it is also used when a user uses the package lookup in the UI.

#### Other Possible Algorithms

One possible alternative algorithm to find an optimized route would be the nearest neighbor. This algorithm is a type of greedy algorithm that finds the shortest path to a new node or neighbor until all nodes have been visited and then returns to the starting node. This algorithm can be implemented with several self-adjusting data structures, for example, a graph.

Another alternative would be the farthest insertion. This algorithm starts by connecting the start to the furthest node. Then it will repeatedly find the next furthest unvisited node and place it between whichever two nodes would yield the lowest overall cost/weight. Like the nearest neighbor, this can be implemented with several self-adjusting data structures such as a linked list.

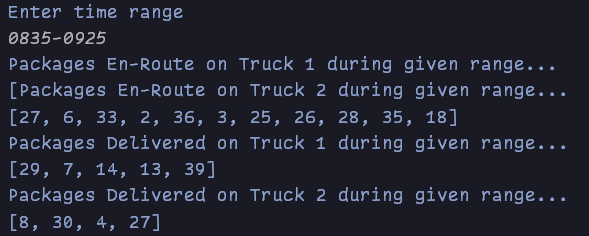
#### Algorithm Differences

Compared to my algorithm, the nearest neighbor algorithm is quite similar. This is because my algorithm actually started as a nearest neighbor algorithm, but without good heuristics, I found the results to be lacking. In my algorithm, the path does not depend solely on heuristics and luck because it attempts to capture key data points like the closest start and end to the hub. The time complexity for the nearest neighbor algorithm is O(n^2) which means it is a much more efficient algorithm than mine, which is O(n!). Both are exponential, but O(n!) is a much stronger curve that reaches towards infinity faster.

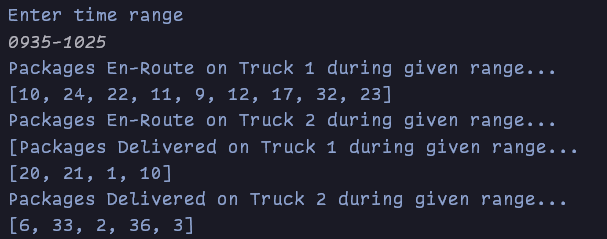
Completely unlike my algorithm, farthest insertion first identifies not the closest data points by the farthest and works its way backward. Farthest insertion would also lead to the real best path, not just an optimization of a given path like mine. The time complexity for farthest insertion is the same as the nearest neighbor algorithm with O(n^2). Compared to my algorithm, again, it is much better in terms of time complexity.

### Status Checks:

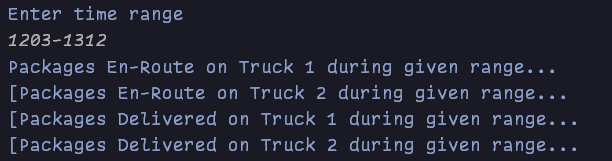
#### First (8:35am - 9:25am)



#### Second (9:35am - 10:25am)



#### Third (12:03pm - 1:12pm)



### Code Execution Screenshots:

These screenshots are located in the root directory of the project.

Path: QuickShip/captures

### Sources:

No sources used that warranted an in-text citation.