# **NHP2 TASK 1: WGUPS ROUTING PROGRAM**

Elexis Rox, ID#001478546

Computer Science, Western Governors University

C950: Data Structures and Algorithms II

August 5, 2023

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

This program relies heavily on a Nearest Neighbor algorithm to load the packages according to their distance from each other. After prioritizing special instructions in the notes for each package, the program searches for the closest delivery point, or Nearest Neighbor, next to a package and attempts to ensure those packages are loaded together on the same truck. The packages are then sorted according to closest distance from one another. If delivery deadlines are not met after this sorting, the loading is optimized. This optimization removes packages from trucks that could be slowing down the route and re-loads them onto a more appropriate truck, again basing its loading decisions largely on packages’ distances from one another.

1. *Write an overview of your program, in which you do the following:*
   1. *Explain the algorithm’s logic using pseudocode.*
2. **Load the Packages**

Iterate through all packages for the first time:

Packages are split into two categories: restricted and unrestricted.

Restricted packages have special notes. Unrestricted do not.

If the package is restricted:

If the package is flagged:

Add the package into the hold compartment.

If the package is delayed:

Add the package onto Truck 3.

If the package is required to be loaded on a specific truck:

Add the package onto that truck.

If the package is required to be delivered with other packages:

Check if any sibling package has been loaded.

If a sibling package has been loaded:

Load this package onto the same truck.

If no sibling package has been loaded:

Load this package onto Truck 1.

If the package is not restricted:

Wait until the next loop to process.

Iterate through all packages for the second time:

If the package is not restricted:

If the package is not added to any truck yet and it is being delivered to Holladay:

Add the package to Truck 1.

If the package is not added to any truck yet and its nearest neighbor is already loaded onto a truck:

Add the package onto that truck.

If the package is not added to any truck yet and it has a deadline:

Try adding the package to Truck 1:

If Truck 1 is full:

Add the package to Truck 2.

If the package is still not loaded onto any truck:

Add the package to the first available truck.

Once the package is added to a truck:

Check if there are any packages with identical addresses:

If there are packages with identical addresses:

Add those packages to the same truck as well.

1. **Create the Routes**

Iterate through the 3 delivery trucks and perform the following for each truck:

**Put the Packages in Order**

Iterate through each package on the truck’s package list.

Loop through the other packages.

Get the distance between the first package and the next in the loop.

Keep track of the package that has the smallest distance.

Group packages that share the same address.

Store the package with the nearest distance at the end of the loop.

Iterate through the addresses on the package list starting with the starting location.

Loop through the other addresses until the closest address is found.

Add that address and the correlating packages next in the route.

Iterate through the route that is now complete. At each stop:

Store the location.

Store the current time.

Store the distance traveled.

Store each package’s delivery time when it reaches its destination.

After that loop is complete, find the last package in the route.

Find the distance between the package and the HUB.

Add that distance to the total distance traveled.

**Check for Late Deliveries**

For a max of 3 iterations, loop through the truck’s package list

Compare the delivery deadline and the delivered time.

If delivery deadline is earlier than delivered time:

Move the package to the front of the route.

Break out of the loop to check for late deliveries again.

If delivery deadline is later than delivered time:

Continue looping through the packages.

If 3 loops are attempted and there are still late packages:

**Optimize the Routes**

Add packages whose destination city is ‘Holladay’, ‘Murray’, or ‘West Valley City’ to list ‘packages\_to\_move\_cities”.

Add packages from truck 1 that do not have deadlines and are not restricted to list ‘packages\_to\_move”.

Add packages from truck 2 that are not restricted to list ‘packages\_to\_move”.

Add packages from truck 3 that have deadlines and are not restricted to list ‘packages\_to\_move\_deadlines”.

Loop through each of the three lists made:

Remove each package from its respective truck.

Loop through the ‘packages\_to\_move\_cities’.

If destination city is ‘Holladay’:

Add to truck 1.

If destination city is ‘Murray’:

Add to truck 3.

If destination city is ‘West Valley City’:

Add to truck 2.

Loop through the ‘packages\_to\_move\_deadlines’.

Try to add the package to truck 1.

If truck 1 is full:

Add to truck 2.

Loop through the ‘packages\_to\_move’.

Find the package’s nearest neighbor.

Check if the nearest neighbor is already on a truck.

If it is:

Add the package to that truck.

If the package was not added yet:

Add the package to truck 1.

If truck 1 is full:

Add it to truck 3.

If truck 3 is full:

Add it to truck 2.

At this point, all routes are complete and delivery deadlines will have been met.

* 1. *Describe the programming environment you used to create the Python application.*

The program was created in PyCharm 2022.3.3 (Community Edition) using Python 3.11. The software environment was run on Windows 11 on a desktop computer.

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

Overall program space-time complexity: O(N^2)

Classes listed in alphabetical order. Additional explanations and a breakdown of individual methods can be found in the comments within the code. The number of trucks is considered constant in the context of this project.

**Dispatch.py**

|  |  |
| --- | --- |
| Method/Code segment | Space-time complexity |
| Dispatch class | O(N^2) |
| def load\_hash\_tables | O(N) |
| def get\_package\_hash\_table | O(1) |
| def load\_trucks | O(N^2) |
| def process\_packages | O(N) |
| def process\_restricted\_package | O(1) |
| def process\_unrestricted\_package | O(1) |
| def create\_all\_truck\_routes | O(N^2) |
| def create\_route | O(N^2) |
| def put\_pkgs\_in\_order | O(N^2) |
| def store\_route\_data | O(N) |
| def check\_late\_deliveries | O(N) |
| def optimize\_routes | O(N) |
| def earliest\_return\_to\_depot\_time | O(1) |
| def add\_package\_to\_truck | O(N) |
| def remove\_package\_from\_truck | O(1) |
| def correct\_flagged\_packages | O(N) |
| def move\_package\_up | O(N) |
| def get\_loaded\_addresses\_by\_truck | O(1) |
| def set\_loaded\_addresses\_by\_truck | O(N) |
| def get\_flagged\_packages | O(1) |
| def set\_flagged\_packages | O(1) |
| def get\_truck\_index\_by\_id | O(1) |
| def get\_truck\_from\_package\_id | O(1) |
| def get\_current\_truck\_object | O(1) |
| def get\_package\_by\_id | O(N) |
| def get\_total\_distance | O(1) |
| def set\_total\_distance | O(1) |
| def get\_number\_or\_drivers | O(1) |
| def set\_number\_of\_drivers | O(1) |
| def get\_truck\_location\_at\_time | O(N) |
| def assess\_delivery\_status | O(1) |
| def compare\_two\_times | O(1) |
| def assess\_time\_input | O(1) |
| def read\_time | O(1) |
| def convert\_to\_datetime | O(1) |
| def print\_truck\_stats | O(N) |
| def print\_full\_package\_info | O(N) |
| def print\_all\_packages\_details | O(N^2) |

**Distance.py**

|  |  |
| --- | --- |
| Method/Code segment | Space-time complexity |
| Distance class | O(N^2) |
| def parse\_addresses\_csv | O(N) |
| def parse\_distances\_csv | O(N^2) |
| def lookup\_distance | O(N) |
| def retrieve\_distance\_wgups | O(N) |
| def nearest\_neighbor | O(N) |

**HashTable.py**

|  |  |
| --- | --- |
| Method/Code segment | Space-time complexity |
| HashTable class | O(N) |
| def get\_hash | O(1) |
| def get\_hash\_table | O(1) |
| def insert | O(1) |
| def modify | O(N) |
| def search | O(N) |
| def remove | O(1) |

**Main.py**

|  |  |
| --- | --- |
| Method/Code segment | Space-time complexity |
| Main class (Program overall) | O(N^2) |
| Creates an instance of the Dispatch class | O(N^2) |
| Creates an instance of the UserMenu class | O(N^2) |
| Calls exit\_choice | O(1) |

**Package.py**

|  |  |
| --- | --- |
| Method/Code segment | Space-time complexity |
| Package class | O(N) |
| All attribute getters/setters | O(1) |
| def is\_package\_restricted | O(1) |
| def identical\_packages\_are\_restricted | O(N) |

**PackageCSVExtractor.py**

|  |  |
| --- | --- |
| Method/Code segment | Space-time complexity |
| PackageCSVExtractor class | O(N) |
| def extract\_pkg\_csv | O(N) |
| def get\_pkg\_hash\_table | O(1) |
| def pkg\_process\_row | O(1) |
| def parse\_notes | O(N) |

**Schedule.py**

|  |  |
| --- | --- |
| Method/Code segment | Space-time complexity |
| Schedule class | O(N) |
| def load\_schedule\_data | O(N) |

**Truck.py**

|  |  |
| --- | --- |
| Method/Code segment | Space-time complexity |
| Truck class | O(1) |
| All attribute getters/setters | O(1) |
| def is\_package\_on\_truck | O(1) |
| def add\_package | O(1) |
| def remove\_packages | O(1) |
| def get\_last\_package\_in\_route | O(1) |
| def print\_package\_list | O(1) |

**UserMenu.py**

|  |  |
| --- | --- |
| Method/Code segment | Space-time complexity |
| UserMenu class | O(N^2) |
| def ask\_time\_input | O(1) |
| def begin\_main\_menu | O(N^2) |
| def offer\_exit\_choice | O(1) |

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

The algorithm described in B.1 is built to sort packages based on their properties rather than the specific package itself, so it can easily adapt to a greater number of packages. There are safeguards built in, such as in process\_unrestricted\_package in the Dispatch class, to print an error message to the user if all trucks are full before all packages are loaded. This would occur with more than 48 packages, assuming the company still has only 3 trucks and each truck can still only hold 16 packages.

* 1. *Discuss why the software is efficient and easy to maintain.*

The software utilizes classes for both trucks and packages, ensuring that both could grow in number easily. Most of the company’s standards may be set through class attributes, such as the truck speed, number of trucks, number of drivers, etc. This ensures flexibility as the company grows and changes.

The software’s space-time complexity is O(N^2), but despite this, it is efficient in that its algorithm is broken up into smaller methods that perform different functions. If the company were to expand to the point that the number of packages were causing unnecessary lag in the program, the developer could easily change or update only a specific part of the algorithm, such as targeting specifically how unrestricted packages are processed. This ensures that the program is both easy to maintain and promotes longevity.

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

The strength of a self-adjusting data structure such as a hash table is that it can easily expand to fit any number of packages. While there are a set number of buckets within the table, each bucket is an array, which can hold an indefinite amount of items. The bucket structure helps mitigate collisions, which can degrade a database over time. However, the weakness of a hash table is that the more collisions there are and the greater the size of the arrays in the buckets, the more inefficient the hash table becomes.

1. *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.*

A hash table is the self-adjusting data structure utilized in this program. While there are a set number of buckets within the table, each bucket is an array, which can hold an indefinite amount of packages. The bucket structure helps mitigate collisions. The packages are stored as objects that each hold their data in the form of attributes. This structure ensures that the number of packages may change, as well as ensuring that each package object contains all the data that needs to be accessed.

1. *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.)*

Screenshots are included in program folder.

1. *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.*

Screenshots are included in program folder.

1. *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.*
      1. The algorithm can handle any amount of packages because it is based on the properties set within the package object, rather than the amount of packages themselves.
      2. The algorithm accounts for the special notes of the packages by prioritizing space for any special requirements first.
      3. The algorithm is able to check to make sure the delivery deadlines are met and readjust the loaded packages if necessary using the optimize\_routes method in the Dispatch class.
   2. *Verify that the algorithm used in the solution meets*all*requirements in the scenario.*

The algorithm meets all requirements. Each truck is an object that tracks its total miles. The mileage varies each time due to the program’s flexibility, but the mileage is always under 140 miles. This is largely due to the grouping of packages going to different cities, which occurs in the optimize\_routes method in the Dispatch class. All packages are delivered on time as well. After the loading process, all delivery deadlines are checked. If they are not met, optimize\_routes ensures that they will be moving forward.

* 1. *Identify****two****other named algorithms, different from the algorithm implemented in the solution, that would meet the requirements in the scenario.*
     1. *Describe how*each*algorithm identified in part I3 is different from the algorithm used in the solution.*

1. Greedy Algorithm: A greedy algorithm finds a solution in the smallest amount of time possible. It makes the choice that seems best in the moment, rather than prioritizing one specific quality, such as the distance between packages in the Nearest Neighbor algorithm.

2. Brute Force Algorithm: A brute force algorithm iterates through every possibility and chooses the best solution afterwards. While this may have worked for a package company this small, the processing power would become too cumbersome as the amount of packages grows.

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

I would have used less classes and refined the communication between classes. This was my first program I have created of this size, and while I planned extensively at the beginning, I learned much better means of organization as I went along. For example, I would limit static methods to just one class so they are more easily accessible. I would also limit CSV extraction to just one class. The separation of distances and packages felt right when I began this project, but ultimately caused more redundancy.

1. *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.*

The algorithm meets all requirements. There is a lookup function built into the program that retrieves data from the hash table based on the package ID. Search results provide all reporting on the package. Each truck is an object that tracks its total miles. The mileage varies each time due to the program’s flexibility, but the mileage is always under 140 miles. This is largely due to the grouping of packages going to different cities, which occurs in the optimize\_routes method in the Dispatch class. All packages are delivered on time as well. After the loading process, all delivery deadlines are checked. If they are not met, optimize\_routes ensures that they will be moving forward.

* + - 1. *Explain how the time needed to complete the look-up function is affected by changes in the number of packages to be delivered.*

Since the program has a Time complexity of O(N^2), the time required for look-up will grow considerably as the number of packages grow. However, I did break up the code into compartmentalized methods to ensure that only parts of the program would need to be updated if processing times become too long.

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

The space usage will grow as the number of packages grow. The space-time complexity is O(N^2).

* + - 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.*

The look-up time and the space usage will not grow as quickly if trucks or cities are increased, as opposed to increasing the number of packages. Trucks are treated as objects, which adds flexibility to the program and makes the storing of truck data efficient. The number of cities would not affect the data structure, but a greater number of addresses, regardless of city, would cause the data structure to grow.

* 1. *Identify****two****other data structures that could meet the same requirements in the scenario.*
  2. *Describe how*each*data structure identified in part K2 is different from the data structure used in the solution.*
     1. Linked List: A linked list is a linear data structure. The elements, or nodes, are linked together using pointers. Searching through the list can potentially result in traversing it in its entirety, so it would likely not be more efficient than a hash table. However, inserting or deleting elements can be more efficient in large data sets.
     2. Balanced Tree: A balanced tree is a self-balancing binary tree that ensures efficient operations. They provide a more predictable time complexity for operations performed on the stored data. While hash table use hash functions to map keys to indices, balanced trees organize elements hierarchically which maintains balance throughout the tree. This would likely be more time-consuming to implement, but it would be more time efficient than a hash table.