Sample Core Algorithm Overview

### Stated Problem:

The purpose of this project is to create an algorithm using Python to develop an effective and efficient Package Delivery algorithm. Given a set of packages and a set of nodes on a map, the algorithm must deliver all packages to the specified locations according to the package’s special notes and delivery deadlines. An interface must be provided to view the package statuses at a specified time, and to run a work day, recording the delivery times and package statuses at the end of it.

### Algorithm Overview (A):

The proposed core algorithm is a “Nearest Neighbor” heuristic used to provide an efficient solution to the traveling salesman problem. This heuristic is a greedy algorithm, determining the “Nearest Neighbor” according to a set of priorities determined by the same location, deadline, earliest load time (delayed status), group, truck, and distance, in that order.

The route-building algorithm shall run as follows:

1. Add package to truck route if same location as last package added exists
2. Add deadline package to truck route if a) earliest load time < clock, b) sooner deadline, or c) distance is shortest to the last package.
3. If no more deadline packages: Determine which truck grouped\_packages belong on. Add packages by shortest distance to the last package.
4. If no more grouped\_packages: Add remaining package or truck specific package to truck route if a) earliest load time < clock, and b) distance is shortest to the last package.

### Operation time (B5):

Operation time for the package delivery algorithm depends on the complexity of finding a candidate package. That is, for each package, all packages are looped through to determine the best subsequent package resulting in a complexity of O(N^2). One way to reduce this complexity would be to implement a heap structure that heapifies based on the conditions for the best candidate. This would reduce the inner loop to O(logN), therefore reducing the total complexity of the sortPackages method to O(NLogN).

### Implemented Methods (B1, B2, B3):

#### work(time=-1) ::

Executes work day, delivering packages according to special notes and delivery deadlines.

Algorithm Complexity: Best Case: O(N) Average Case: O(N) Worst Case: O(N^2)

----------------------------------------------------------------------------

# Reverse routes for easy removal/traversal

''' O(N) AC: O(N) WC: O(N) '''

Do reverse routes

''' O(N) AC: O(N) WC: O(N) '''

Do distribute packages

# work day is complete if all packages are delivered.

# number of trips back to hub is a multiplier and does not affect

# delivering number of package's complexity.

While work day not complete:

# x number of trucks is multiplier, not packages,

# therefore have no impact on complexity.

For truck in trucks:

''' BC: O(N) AC: O(N) WC: O(N) '''

While truck not empty:

If time != -1 and truck clock has reached or will reach time in 1 turn: break

Do truck travel

Do truck deliver

''' BC: O(1) AC: O(1) WC: O(N) '''

Do update package

If truck clock has reached time: work day complete; break

Elif packages still exist: return to hub.

If work day completed: mark truck completed

If all trucks have completed work day: workday is completed

''' BC: O(N) AC: O(N) WC: O(N) '''

If work day not completed: distributePackages()

return

#### distributePackages(packages) ::

Distributes sorted/received packages into trucks available for receiving.

Algorithm Complexity - Best Case: O(N) Average Case: O(N) Worst Case: O(N)

----------------------------------------------------------------------------

Get next truck routes from end of self.routes

For each truck:

''' BC: O(N) AC: O(N) WC: O(N) '''

Load packages onto truck

return

#### receivePackages(packages) ::

Take packages, call package sorting, and prepare them in hub.

Algorithm Complexity: Best Case: O(N^2) Average Case: O(N^2) Worst Case: O(N^2)

----------------------------------------------------------------------------

''' BC: O(N^2) AC: O(N^2) WC: O(N^2) '''

routes = sortPackages()

For trip in routes:

for truck in trip:

Remove placeholder package

''' BC: O(N) AC: O(N) WC: O(N) '''

for package in truck:

If not first trip: Set package status to 'Awaiting pickup'

If first trip: Set package status to 'Out for Delivery'

If package.earliestLoadTime: Set package status to 'Delayed'

Return

#### sortPackages(packages) :: returns sortedPackages

This method parses package notes, and sorts them into truck routes and trips.

Algorithm Complexity - Best Case: O(N^2) Average Case: O(N^2) Worst Case: O(N^2)

----------------------------------------------------------------------------

''' BC: O(N) AC: O(N) WC: O(N) '''

For package in packages

package.parseNotes()

Do sort into list queues

# while merge not complete is multiplier, therefore does not affect complexity.

While merge not completed:

''' BC: O(1) AC: (logN) WC: O(N) '''

''' if packages(N)=50: at most, group packages can be 25 with 2 elements each. (25 \* 2 = 50 = N) '''

''' or group packages can be 2 with 25 elements each. (2 \* 25 = 50 = N) '''

''' the two loops are fundamentally tied to eachother because as one increases, the other decreases. '''

For groupedpackage in grouped\_packages:

For packagenumber in groupedpackage:

If packagenumber in other groupedpackage: Merge groupedpackage

# Replace grouped packageIDs with actual package

''' BC: O(1) AC: (logN) WC: O(N) '''

''' if packages(N)=50: at most, group packages can be 25 with 2 elements each. (25 \* 2 = 50 = N) '''

''' or group packages can be 2 with 25 elements each. (2 \* 25 = 50 = N) '''

''' the two loops are fundamentally tied to eachother because as one increases, the other decreases. '''

For groupedpackage in grouped\_packages:

For packagenumber in groupedpackage:

''' O(1) AC: O(1) WC: O(N) '''

packagenumber = lookupPackage(packagenumber)

''' BC: O(N) AC: O(N) WC: O(N) '''

''' First two while loops ensure each package is addressed. '''

While packages exist not in routes:

Generate object to hold trip

while trip's route not completed:

if truck capacity allows:

''' BC: O(N) AC: O(N) WC: O(N) '''

for package in deadline, remaining, and truck packages:

If package location same as last package added's location: add package; move on to next truck

''' BC: O(N) AC: O(N) WC: O(N) '''

If deadline\_packages exist: determine most eligible candidate to add

Else if grouped\_packages exist:

''' BC: O(1) AC: (logN) WC: O(N) '''

If group's assigned truck is current truck: Add all group packages

''' BC: O(N) AC: O(N) WC: O(N) '''

Else if remaining\_packages or truck\_packages exist:

Determine best candidate package between remaining\_packages and truck\_packages

Increment truck

Increment trip

return routes

The pseudo code above shows all of the O(\*) terms necessary to determine the time complexity for the Sorting and Receiving of all packages. These methods are then used in conjunction with the method work() that executes the work day. Distribution of packages onto trucks is handled in the work() method.

### Market Adaptability, Scalability, and Maintainability (B4, B5)

This algorithm is scalable in many ways and not so scalable in other ways. The biggest breakdown of the solution comes in an abnormally large amount of delayed packages. This could cause trucks to be sitting in place, wasting delivery time.

One adaptable feature of this algorithm is its ability to deal with both directed and undirected graphs. In fact, this algorithm will become more efficient than other algorithms when given a more directed edge graph.

One adaptability failure may be a decrease in efficiency when given a highly clustered graph with a large amount of clusters, due to its method of organizing packages according to its average distance to all other nodes.

##### (K1c)

This algorithm is able to be applied to other cities due to the fact that it uses an adaptable graph to model the delivery locations. Expanding the service to multiple cities will be easy to accomplish. The only negative to using a graph is that it takes O(N^2) to create the graph, however after it is created, the process of retrieving information is very quick.

This algorithm is also easily able to scale to handle more trucks. The number of trucks shouldn’t have a noticeable impact on the performance of the algorithm because they function as a computational multiplier on N packages, therefore not affecting the complexity. The trucks are also built to be highly configurable to accommodate multiple types of trucks with varying speeds and capacities. This should allow the algorithm to adapt to any scenario the operators throw at it.

This algorithm can handle large numbers of package increases, so long as the computational hardware scales with it at O(N^2). If a heap data structure is used to determine the most eligible candidate, giving the algorithm a complexity of O(NLogN), then the hardware would need to scale with that. This would make the algorithm a computationally efficient solution that would not require hardware to scale quite as extremely with the number of packages.

### Self Adjusting Data Structures (B6)

#### Hash Table

I chose a hash table to store, search on, and update my packages. This was chosen for the following strengths and weaknesses.

Strengths:

* Simple.
* O(1) best case complexity, O(1) average case complexity, O(N) worst case complexity for inserting and searching. Efficient accessing of data will result in a more efficient core algorithm.

Weaknesses:

* Efficiency of the hash table greatly decreases when you have irregular key data. In my hash table’s case, the packages are being stored with a key function of packageID % N (N = number of packages hash table is initially created with). If the data adjusts in a non-random pattern to have many of the same mod values, search and update times can increase to a complexity of O(N).

#### Graph

I chose a graph data structure to house my delivery locations and relations to other delivery locations. My graph implemented an adjacency matrix to store the relations. I chose this with the following strengths and weaknesses:

Strengths:

* Highly configurable and adaptable - allows the potential for directed edges.
* Efficient for route planning and shortest paths.
* Their adaptability allows for easy plug and play of new and different algorithms.

Weaknesses:

* Slightly more coding