## A: ALGORITHM SELECTION

This program uses a Nearest Neighbor algorithm to schedule the package deliveries. This algorithm is applied to selected subsets of the input data to ensure that the scenario requirements are met.

## B1: LOGIC COMMENTS

The algorithm is applied during the loading of the trucks. The trucks themselves run on a queue to visit package destinations in the order chosen at loading.

Basic overview:

Packages are sorted by requested truck (imported from CSV file).

Each truck’s packages are sorted into early delivery and everything else.

Early delivery packages are grouped by delivery time.

Nearest neighbor algorithm is used on each subgroup of early delivery packages to get them into the truck’s delivery queue.

The pool of “EOD” packages is checked to see if any packages share addresses with early delivery packages, those packages are queued up with the matched early packages.

Once early packages are done, nearest neighbor algorithm is used on the remaining packages until truck is full.

The algorithm changes to try and drive the truck farther away from the Hub with each package. Once it reaches farthest distance it works its way back in, taking care of outlying packages so the next truck does not have to drive out as far.

Algorithm:

Bucket Sort (One bucket for each truck plus an extra)

For t = 1 to number\_of\_trucks:

Create bucket

Append to truck\_bucket

Create package\_bucket to hold non-truck specific packages

For each package:

If package.requested\_truck > 0:

Add package to requested truck\_bucket

Else:

Add package to package\_bucket

For each truck t:

Create list of early delivery packages (early\_packages)

For each package in truck\_bucket[t]:

If package has early delivery:

Add to early\_packages

For each package in package\_bucket:

If package has early delivery:

Add to early packages

For each package in early\_packages:

Remove package from truck\_bucket/package\_bucket

While there are early packages to deliver:

Create current\_earliest variable and initialize to datetime.max

Create current\_list and initialize to an empty list []

For each package in early\_packages:

If package deadline is earlier than current\_earliest:

Store deadline as new current\_earliest

Clear current\_list

Add package to current\_list

If package deadline is the same as current\_earliest:

Add package to current\_list

Run **load\_onto\_truck** method with current\_list.

Depth\_first\_searching is False and optional packages include the trucks truck\_bucket and the package\_bucket.

For each package in the truck’s delivery queue:

Remove package from early\_packages, package\_bucket, and truck\_bucket[t].

Loop until no more early packages to load for this truck.

While there are packages remaining in truck\_bucket[t]:

Run **load\_onto\_truck** method with truck\_bucket[t].

Depth\_first\_searching is True and optional packages include the package\_bucket.

For each package in the truck’s delivery queue:

Remove package from package\_bucket and truck\_bucket[t].

While there are packages remaining in package\_bucket:

Run **load\_onto\_truck** method with package\_bucket[t].

Depth\_first\_searching is True.

For each package in the truck’s delivery queue:

Remove package from package\_bucket.

Once For loop is complete trucks are loaded and ready to be dispatched.

Method: **Load\_onto\_truck:**

Selected arguments:

Current\_location: Address of last package added to truck

Package\_list: Packages to be loaded

Optional\_packages: Packages that can be loaded if space allows

Depth\_first\_search: We are done with this truck’s early packages and want to try to drive away from the Hub until we reach the furthest package possible.

While len(package\_list) > 0 and truck is not at its load limit (specified by scenario):

If (load\_limit – truck.delivery\_queue) == 0:

No room on truck for optional packages, need to finish package\_list

Result = **get\_closest\_package**(package\_list)

Else if depth\_first == True:

Result = **get\_closest\_outward\_package**(package\_list +

optional\_packages)

If result[0 (found package)] == 0:

No outward package found, do nearest neighbor instead.

Result = **get\_closest\_package**(package\_list +

optional\_packages)

Else:

Check if there are any zero distance packages we can drop off while taking care of package\_list.

Result = **get\_closest\_package**(package\_list +

optional\_packages)

If result[1 (distance)] > 0:

Result = **get\_closest\_package**(package\_list)

Next\_package = result[0]

New\_location = next\_package.get\_address()

Load next\_package onto truck (append to truck’s delivery queue)

If truck’s earliest departure is earlier than next\_package’s

earliest departure:

Update truck’s earliest departure to be next\_package’s

earliest departure.

Return location of truck’s most recently added package.

Method: **Get\_closest\_package:**

**This method implements a Nearest Neighbor greedy algorithm.**

Selected arguments:

Location: Source address

Package\_list: Packages to be searched for nearest neighbor

For each package in package\_list:

If distance to package is closest found so far:

Store distance and package

Return [closest package, distance]

Method: **Get\_closest\_outward\_package:**

**This method implements a modified Nearest Neighbor greedy algorithm. It finds the nearest neighbor that is farther from the Hub than the source address.**

Selected arguments:

Location: Source address

Package\_list: Packages to be searched for nearest neighbor

For each package in package\_list:

Measure distance from location to Hub

Measure distance from package to Hub

If distance to package is closest found so far AND

package is farther from hub than current location:

Store distance and package

Return [closest package, distance]

## B2: Development Environment

This program was developed on PyCharm Community 2020.3.2, running on a Windows 10 machine. No external modules were imported beyond the provided data tables.

## B3: SPACE-TIME AND BIG-O

1. Decode input tables (runs in O(*n2*) where *n* is the number of addresses in data table)
   1. Importing the table of packages runs in O(*n*) time.
   2. Importing the table of distances runs in O(*n2*) time and will run slowly for a sufficiently large number of addresses. For the constraints of this project (40 addresses) this is acceptable.
2. Load trucks (runs in O(*n2*) time for large *n* where *n* is the number of packages)
   1. Each package may be compared to all remaining packages to find its nearest neighbor.
3. Dispatch trucks (runs in O(*n*) time for large *n* where *n* is the number of packages)
   1. Each package is touched once as it is delivered.

Overall, the program runs in O(*n2*) for large *n* where *n* is the number of packages. An input could be constructed to improve this down to O(*n*) but this would be an unrealistic input with each package having manually tuned deadlines and specifically requested trucks for loading.

## B4: ADAPTABILITY

This application will scale well within the given scenario. While it runs in O(*n2*) time based on the number of packages provided, this number would not realistically grow into the thousands for a given hub of this small company.

For larger companies, this application will not scale well. As the number of addresses and packages grow this application will not be able to break the problem down into manageable pieces. The application would certainly be able to function and produce a result, but the time required could grow to be excessive. The largest shortcoming would be in the handling of the distance table. Unless the table is pruned to only include addresses pertaining to packages it would quickly become too large to store and process.

## B5: SOFTWARE EFFICIENCY AND MAINTAINABILITY

This application is both efficient and maintainable. It runs in polynomial time and, for this scenario, would be able to run most of its functions overnight before trucks were loaded. The hash table used for storing packages scales in size to match the number of packages, conserving memory space by only using the minimum required.

For maintainability, the program’s functions are split into separate modules that can be replaced or upgraded as needed. Data import, package loading, and truck dispatching are all separated. The underlying algorithm can also be replaced without rewriting the entire application.

## B6: SELF-ADJUSTING DATA STRUCTURES

The PackageHash class found in packagehash.py is a self-adjusting data structure. It initializes to a list of size 20 and resizes in O(1) time if a key exceeds its size. The table uses direct addressing, mapping the package id to the table index. The strengths of this data structure are that it prevents collisions (assuming package ids are unique) and resizes as needed. The weaknesses of this data structure are that it inefficiently handles non-consecutive package ids by creating empty table space in the missing ids and it does not scale itself back down if packages are removed from the table.

## C: ORIGINAL CODE

The application delivers all packages on time with a total mileage of 121.4 and a finish time of 12:10 PM.

## D: DATA STRUCTURE

The PackageHash class in packagehash.py stores and retrieves package data. The data structure uses a list to store Package objects.

## D1: EXPLANATION OF DATA STRUCTURE

The PackageHash data structure implements a directly addressed hash table. The package’s unique id values are used as their hash keys. The package is stored and retrieved from this index (key – 1) in the list. This structure works well for storing this data because the packages are already assigned unique id numbers.

## E: HASH TABLE

The provided hash table must include an insertion function which can insert all the packages info (see part D) into the hash table.

PackageHash.add(package) provides this functionality. The Package object contains all of a package’s information.

## F: LOOK-UP FUNCTION

PackageHash.get\_package(key) provides this functionality. It returns a Package object that contains all the required information.

## G: INTERFACE

Provided in program.

## G1-G3: 1st, 2nd, and 3rd status checks.

Screenshots located in project directory.

## H: SCREENSHOTS OF CODE EXECUTION

Screenshots located in project directory.

## I1: STRENGTHS OF THE CHOSEN ALGORITHM

Two strengths of the chosen algorithm:

1. The algorithm runs efficiently for a small number of packages, generating a good enough solution in a reasonable amount of time.
2. The algorithm provides consistent results and does not rely on randomization to produce a solution.

## I2: VERIFICATION OF ALGORITHM

The algorithm delivers all packages on time and returns the trucks to the Hub by 12:10 PM with a total of 121.4 miles traveled.

## I3: OTHER POSSIBLE ALGORITHMS

Two other possible algorithms:

1. Randomize package order and then test solution
   1. Separate packages by delivery deadline
   2. Randomize package order within each subset
   3. Test solution and compare to previous best solution
   4. Iterate until a good enough solution is reached
2. Run simple nearest neighbor algorithm and then use a swapping technique to improve
   1. Separate packages by delivery deadline
   2. For each subset, use nearest neighbor algorithm to get simple solution
   3. For each subset, choose two different package pairs and swap two of the packages
   4. Check if solution improves
   5. Iterate until all pairs are stable and no swaps improve the solution

## I3A: ALGORITHM DIFFERENCES

Algorithm I3.1 would be slightly easier to program than my chosen algorithm but would require vastly greater computational resources to run. It replaces the nearest neighbor algorithm with pure randomization and iteration. The chosen algorithm does not iterate repeatedly until a solution is found, it runs once and is finished. Algorithm I3.1 could require many iterations to find an acceptable solution.

Algorithm I3.2 begins only slightly differently from my chosen algorithm, replacing the modified nearest neighbor with a simple nearest neighbor. It relies instead on iterating through the initial solution and attempting to improve it by swapping pairs of packages in the delivery order. This would likely produce a better result in the end but would take longer to run and more work to develop.

## J: DIFFERENT APPROACH

If I were to attempt this project again, I would start by designing an optimization program for the distance table. Currently the program uses the table as provided but there are cases where the table is not optimal. There are shortcuts that are faster than the direct connections between certain addresses. This would improve the solutions found by even the most basic algorithm and could potentially save time trying to hit the 140-mile goal.

## K1: VERIFICATION OF DATA STRUCTURE

The PackageHash class in packagehash.py stores and retrieves package data. The data structure uses a list to store Package objects and does not use the built-in dictionary data structure.

## K1A: EFFICIENCY

The lookup function of the hash table returns results in O(1) time regardless of the number of packages stored in the table. The storage function is also running in O(1) time and table resizing is done in a single operation. It is efficient timewise for any number of packages.

## K1B: OVERHEAD

The space usage of the hash table is very efficient if packages have a continuous ordering of package ids. If there are gaps between the ids the table will grow to have empty space where those gaps are. These gaps could potentially be large. The table will also remain at its largest size without sizing back down if packages are later removed.

## K1C: IMPLICATIONS

The data structure is unaffected by the number of trucks and the number of cities. That information is stored in different lists elsewhere in memory. The number of requests to the hash table will grow in proportion to the number of packages and not to the number of trucks or cities.

## K2: OTHER DATA STRUCTURES

Two other possible data structures:

1. Use a hashing function to create a hash key from the package id. Store the package in a list using the hash key as the list index. Use chaining to handle collisions.
2. Use package id as the hash key to directly address data into a list (same as chosen data structure). Instead of storing items as a Package object, store them as a tuple and return this tuple when the hash table is queried.

## K2A: DATA STRUCTURES DIFFERENCES

1. This structure uses a hash function to address items to the table instead of direct addressing. This structure would have more efficient space usage for a random set of package ids. It uses chaining to handle collisions where the chosen data structure will not have collisions if ids are unique. Any collisions in this structure will reduce its time efficiency as the chained objects will need to be searched through one at a time. Overall, it could save space at the cost of time and complexity.
2. This structure is like the chosen structure but would store data in a tuple instead of a Package object. This would make little difference to the program itself but would reduce readability and useability for the programmer. It would also have an impact on maintainability. The programmer would need to remember which index in the tuple contained desired information instead of calling methods of the Package class.

## L: SOURCES

No sources were used or quoted in the writing of this document. No code samples were copied into the program.