Daniel [redacted]
[redacted]

The solution presented is for an algorithm that determines the most optimal route for delivering a set of packages in a city. Optimal route in this case, refers to having the shortest distance traveled while having all packages delivered on time.

Supplied in this document is an analysis of the program as a whole, and the algorithms in use. Each bulleted item below corresponds to the appropriately labeled item in the Task Requirements for the project submission.

A. The core algorithm is the Nearest Neighbor Search (NNS), combined with rules-based heuristics for package selection.
B. The algorithm has two phases. The first phase is package selection. This involves a detailed heuristic for selecting the best packages for a route based on several conditions, such as delivery deadline, and destination relative to other packages. The second phase is route selection based on the subset of packages which are loaded onto a truck. This follows a simple nearest neighbor algorithm, and simply finds the closest node from where the truck currently is, and navigates to that location to deliver the next package.

   Pseudocode for the route selection is as follows:

```
method search(start, to_visit):
    let visited = queue of the resultant path we are building
    let to_visited = queue of locations we need to visit

    for all packages in truck:
        add package location to queue to_visit

    call search_internal(start, to_visit, visited)
    return visited

method search_internal(start, to_visit, visited):
    let nearest = infinity

    if to_visit is empty return

    for all items in to_visit:
        if distance between item and start is less than nearest
            set nearest = item

    pop nearest from to_visit
    add nearest to visited
    call search_internal(nearest, to_visit)
```

   1. The packages to load onto each truck are selected with a self-adjusting rules-based heuristic, where subsequent packages are selected based on which packages were already selected.

   a. An additional self-adjusting heuristic is used for route selection. If there are packages with an earlier delivery deadline, those are selected for delivery first over packages with an End-of-day (EOD) requirement.
2. The worst-case complexities are as follows:
   a. Space is O(N^4), which comes from O(V^2 * P * R).
   b. Time is O(N^3), which comes from O(P^2 * R)
   c. Where:
      1. V is the number of vertices (delivery locations)
      2. P is the number of packages
      3. R is the number of packages required to be shipped with a particular package
3. The solution provided is scalable and adaptable in the following ways:
   a. Everything is dynamic. The number of drivers, trucks, packages, locations, and package update events. The entire solution is able to adapt to any number of variables for the given situation. If a future HUB requires 5 drivers, or 4 trucks, the solution will still adapt to give you an approximate best delivery route.
   b. Space usage is more manageable, as the space required is proportional to the square of the number of delivery locations.
4. The code for the provided solution has been organized into logical units, based on what that particular section of code does. Things that exist, like a driver, truck, or package, are 'entities' and are organized together. Abstract things, like a HashMap or Graph, are organized in a 'structures' module. This organization of related items into modules help with maintainability, as it aids the developer to locate a particular piece of code, or see how it is related to other code.
   Given that the algorithm is based on nearest neighbor, the efficiency of the code is on average about 1.5x worse than the true best case, which would typically be found with a brute force algorithm.
5. There are two self-adjusting mechanisms at work in this solution:
   a. The selection of packages to be placed on each truck is based on which packages are already placed, subject to the package notes. For more information on the particular selection algorithm at work, see the comment at the top of the **RuntimeState.assign_packages()** method in **RuntimeState.py**.
      1. A strength of this approach is that the best packages for the route will be chosen, regardless of truck availability or package notes.
      2. A weakness of this approach, however, is that the rules as implemented are designed around the specific package notes in the provided scenario. If additional restrictions are placed on packages, the code will need to be adjusted for that, which could affect the efficiency of package selection.
   b. The path selection algorithm allows for changing paths during route delivery.
      1. A strength of this is that it provides efficiency in situations where a delivery exception can occur (such as a wrong address), such that the package can be re-routed and delivered without the truck returning

back to the HUB first. This is done by regenerating the route path based on newly generated location information.

2. A weakness of the current method, however, is that it only takes affect after the current package on route is delivered. This means if the re-routed package is closer at that moment, time is wasted as it travels further away to deliver the current packages first, before returning to that one. Similar to that, the trucks cannot be rerouted in the middle of a transition from one location to another. The truck must arrive at a location, before it can reroute to another. In other words, redirecting while the truck is on a "edge" of the graph is now allowed, while it would be in real life.
   Additionally, the route does not include the return path to the HUB. When all the packages have been delivered, the truck immediately returns to the HUB, no matter the path weight.

C. See the code provided, the **main()** method is in **PackageRouting.py**.

D. The data structure chosen to work with the algorithm is a Graph using an adjacency matrix for representing the vertex/edge relationships.
   1. The adjacency matrix is of size V × V, where V is the number of vertices. In this case, a vertex is a delivery location.
      Each element of the matrix contains the weight of the edge. In this scenario, the weight is the distance between two locations.
      The distance between locations u and v is denoted by **matrix[u][v]**. This lookup is a constant time O(1) operation.

E. The HashSet used can be found in structures/HashSet.py

F. See above

G. This interface is the terminal UI display

H. See screenshots in attachments.
   Please note that all packages are delivered by 11:54am, so the screenshot for the time period 12:03pm-1:12pm is the screenshot from 11:54am, since the nothing will change after that time.

I. The algorithm of choice is the Nearest Neighbor search with temporal and locality sensitive hashing
   1. The algorithm chosen has the following strengths:
      a. The algorithm is fast. It's simple and only needs to find the smallest weight for the next node. Since the algorithm is simple, it's easy to maintain and understand.
      b. The algorithm groups packages into two buckets: those with delivery deadlines, and those without. This allows prioritization of packages which require earlier delivery.
      c. It also selects packages which are going to the same destination, so they are delivered together.
   2. The search algorithm helps ensure lowest mileage due to how it looks for the next closest location from the last delivery location.
      To ensure packages are delivered on time, the packages added to the truck are split into two groups: packages with delivery deadlines, and packages that only need to be

delivered by end-of-day. If there are any packages with deadlines on board, the route selection searches for the nearest location among those packages first.

3. Two other algorithms that would satisfy the requirements are Dijkstra's Shortest Path and Floyd-Warshall's Algorithm.

    a. Dijkstra's algorithm works by iterating over each node and keeping track of the path as it builds it. If a path between two nodes is found that is shorter than the previous it finds, it discards the old one and keeps going. This algorithm can work in this situation, but because every node is connected to every other node, Dijkstra's algorithm has a massive performance penalty computationally. It is better suited to situations where not every node is directly accessible by every other node, such as one-way streets or airports.
    The Floyd-Warshall algorithm is similar to Dijkstra's in that it keeps track of the previous shortest path, but this one supports negative weighs. The provided scenario doesn't indicate or imply a negative weight edge is possible, so the extra code this algorithm uses to support that is not useful here.

J. I made a few mistakes both in the design and implementation phases of this project. If I were to do this again, I would make several changes:

1. The HashSet implementation wasn't flexible enough to support the lookup methods I ended up using. I should have instead implemented a HashMap or Dictionary to better support O(1) lookups based on various key values.

2. I split the program into two pieces with different algorithms: package selection, then route selection based on already selected packages. Instead what I would do is select a different graph traversal algorithm altogether, such as Dijkstra's with a locality sensitive heading, and applied the algorithm to all unique package delivery locations at once. This would provide me with all possible shortest paths between all the locations. Then I could select the packages based on which packages have the shortest paths between them. I feel that would achieve a better solution.

K. The data structure I used to represent delivery locations is an adjacency matrix.

1. The adjacency matrix allows for fast lookups for location distances. Because the graph representing the data in this scenario is dense, meaning it has significantly high edge to vertex ratio, the adjacency matrix is the best choice. With 27 delivery locations, all connected to each other, is 702 edges. Worst case storage space is 729 entries.

    a. The time complexity for looking up the distance between any two locations is a constant time operation O(1). However, the storage complexity is O(V^2), where V is the number of nodes, or delivery locations.

    b. The overhead required to lookup the next linked item is also constant O(1). This is the strength of the adjacency matrix.

    c. As more delivery locations are added, the storage space growth rate is polynomial.

2. Two other data structures that would also meet the requirements are an adjacency list and edge list.
    a. The adjacency list only stores connected edges, unlike the matrix which also includes storage space for unconnected edges as well, which can't be used for path calculation. The storage space for the adjacency list is O(2E), where E is the number of connected edges. Lookup times for one vertex is O(1), however lookup time for an edge is O(d), where d is the degree of the vertex. Given the above 27 location example, that is 702 total edges for an undirected graph. Since the storage space is similar for worst case, it makes sense to use the matrix, and gain O(1) time for operations without losing more than O(V) space. Similar to the adjacency list, the edge list also only stores the list of connected edges. However, while the storage space is improved at only O(E), the time complexity for any operation has also increased to O(E), and as such is not suitable for use in an efficient algorithm.