**DATA STRUCTURES AND ALGORITHMS II**

**Task 1: Program Planning**

**James Revello (SID: 010649181)**

# Section A:

I will be using the Nearest Neighbor algorithm for package delivery in this assignment. This algorithm is simple to implement and will create a functioning prototype that we can use now and scale later as business needs evolve. It will be decoupled and modular so we can implement another algorithm if business needs change. For now, it is more than sufficient to handle business requirements at this juncture and the key benefit is how easy it will be to explain to other delivery hubs when we port our program to other cities. Other alternatives, such as Dijkstra’s Algorithm or 2-Opt, would be complicated to explain to other hubs as we release to other locations. The former requires knowledge of graphs and vertices and the latter requires swapping edges from tour to tour. It is far more intuitive to simply find the next nearest neighbor and travel there, even if it may not be the most efficient option for tackling the delivery problem.

# Section B:

For our nearest neighbor solution, we will implement a hash-table to store package data. Again, in the interest of simplicity and ease-of-explanation, the hash table will be tailored to this specific algorithm; we will not be using a ‘stock’ dictionary or HashMap from the Python Library.

To account for the relationship between our stored components, each package will have a Package ID number, a delivery address, and a timestamped delivery status (“At Hub”, “En Route”, “Delivered”). Since our business handles small scale deliveries at 40 packages on average, the hash table will merely append packages with an insert function.

Our packages are all unique, so we need to take great care to ensure there are no collisions in which two packages occupy the same space on the table. This is why we will use chaining in our Hash Table and only use the unique Package ID when entering packages.

The aforementioned insert function will take the Package ID as an argument. It will also have a remove function for when deliveries occur using the same parameter. By using the Package ID as the only parameter, we can modify the package details without breaking our algorithm.

As an aside, we may later want to include sender information, Tracking ID’s, or a “fragile” status. The object-oriented nature of the code will allow us to change package details without breaking the rest of the code.

# Section C:

The following is a program overview detailing my plan for implementation:

1. Pseudocode for Nearest Neighbor Implementation

* While **Packages\_on\_truck:** 
  + **Find\_nearest\_neighbor():**
    - Create **nearest\_neighbor** variable to store shortest adjacent address distance
    - For each **package:** 
      * if distance between **current\_truck\_location** and **package\_address** is SMALLER than **nearest\_neighbor**
        + Update nearest neighbor with **package\_distance**
  + **Set\_next\_location()** to **nearest\_neighbor** from last step
  + Increment **truck\_distance\_traveled** with distance traveled to **nearest\_neighbor**
  + Update **delivered\_packages[]** by adding delivered package
  + Update **packages\_on\_truck[]** by removing delivered package
  + Update **truck\_timer** and **package\_delivery\_time**
  + Return to start
* When **packages\_on\_truck** is empty:
  + Return to HUB
  + Update packages, truck timer, etc.

1. For this project, we will use Python 3 in the Pycharm IDE. Using a fully-fledged, integrated development environment rather than a basic code editor allows us to implement useful features, auto complete code, refactor safely, and test/debug our code, all without having to hand-select plugins to achieve the same functionality. The best part is: Pycharm is free, so our business will not have an upfront cost. We will use Gitlab for source control so we can roll back any problems in the code if we break something. As we scale and disseminate more code, we might consider creating Docker Files to ensure teams from different locations can develop without worrying about dependencies. For hardware, we will be using a standard ASUS laptop on Windows 11.
2. Our program will have the following space-time complexities:
   1. The Hash Table class has insert, remove, and search functions. All three have an average time complexity of O(1) but the worst case time complexity is O(n) in case of a collision. Space complexity for the class is O(n). (n in this case is the number of key-value pairs).
   2. In the Populate Package Table and Address List functions, the time complexity is O(n) because they iterate over each package/address once. The space complexity is O(n) where n is the number of packages/addresses.
   3. In the Load Truck method of my Truck class, the time complexity is O(n) because it iterates over each package only one time without any other nested loops. The space complexity is O(n) where n is the number of packages.
   4. In the Deliver Packages method of my Truck class, the time complexity will be O(n^2). This is because it contains a while loop which iterates over the packages, with a nested Find Min Distance function which also iterates over packages. The space complexity for this function remains O(n) for n number of packages.
   5. The Distance Between Addresses has a simple time complexity of O(1) because it has a constant number of operations. The space complexity is also O(1) because the space requirements for the function is constant.
   6. The User Interface portion of my Main class has a time complexity of O(n) because, depending on the user selection, the number of packages printed to screen will grow with the number of packages in general. The space complexity is O(1) however, because it uses a constant amount of space.
   7. Overall, the time complexity of the entire program is O(N^2) because of our nearest neighbor algorithm growing disproportionately compared to the O(n) and O(1) functions in the program. The overall space complexity for the entire program is O(n) for n packages.
3. Our program will be able to scale nicely as package sizes increase because growing package numbers will not outpace the overall runtime of O(n^2). We can grow our Hash Table to accommodate more packages as the business scales and have no concerns since the Hash Table’s implementation has an O(n) runtime.
4. Our program is implemented with maintainability in mind. We will use modular, cohesive building blocks so each portion of the programmatical whole can be scaled and improved. Each portion is loosely coupled so the object-oriented implementation can be changed but the object’s output remains the same. For example, package features might be added as new business logic needs tracked. Maybe new trucks will be bought with faster mileage or bigger capacity, so truck objects should have set variables rather than magic numbers. The nearest neighbor algorithm will be self-adjusting, and will deliver results in polynomial time as a worst-case scenario. And if someone somehow finally solves the NP-Hard Traveling Salesman Problem, we can plug it in without breaking the rest of our code, thanks to object-oriented methodology.
5. The Hash Table data structure was used for this program because it can accommodate massive input sizes with a complexity of O(n) as a worst-case scenario (for insertion and removal of packages). Meanwhile, our ability to search for packages or update our packages takes a runtime of O(1), which is extremely responsive. This is extremely valuable, since we will likely be searching for packages much more often than we will be adding them to the given Hash Table. For example, if we decide to roll out a package tracking program for end customers, we can take search requests with a constant runtime rather than a linear or exponential one. There is an inherent weakness to Hash Tables, however: with any sort of Hash Table structure there is risk of collision if there are duplicative Package ID’s being inserted. While we tried to ameliorate this risk by implementing chaining to our Hash Table, the Package ID key will need to be adjusted for searching packages across different days without duplicating the key.
6. For this program in particular, the simplest package key implementation is to use the Package ID provided on the CSV file. It is a number starting from 1-N, where N is the number of total packages for that day. It is easy to understand, and is the only package variable which is entirely unique and not duplicatable (unlike package weights or addresses, for example).

# Section D:

Tepe, C., Dr (2020, November 17). C950 - Data Structures and Algorithms 2 Course Webinar [Video Webinar]. <https://wgu.hosted.panopto.com/Panopto/Pages/Viewer.aspx?id=f08d7871-d57a-496e-a6a1-ac7601308c71>

Western Governor's University (2021, July 28). C950 WGUPS Project Implementation Steps - Example - Nearest Neighbor. WGU Course Resources. <https://srm--c.vf.force.com/apex/CourseArticle?id=kA03x000001DbBGCA0>