

**A. Develop a hash table, without using any additional libraries or classes, that has an insertion function that takes the package ID as input and inserts each of the following data components into the hash table:**

See hashtable.py

**B. Develop a look-up function that takes the package ID as input and returns *each* of the following corresponding data components:**

See packages.py

**C. Write an original program that will deliver *all* packages and meet all requirements using the attached supporting documents “Salt Lake City Downtown Map,” “WGUPS Distance Table,” and “WGUPS Package File.”**

**1. Create an identifying comment within the first line of a file named “main.py” that includes your student ID.**

See main.py

**2. Include comments in your code to explain both the process and the flow of the program.**

See main.py

**D. Provide an intuitive interface for the user to view the delivery status (including the delivery time) 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.)**

```
==== WGUPS Package Tracker ====
1. Get status of a SINGLE package at a specific time
2. Get status of ALL packages at a specific time
3. Show total mileage of all trucks
4. Exit
Enter your choice (1-4):
```

1. Provide screenshots to show the status of *all* packages loaded onto *each* truck at a time between 8:35 a.m. and 9:25 a.m.

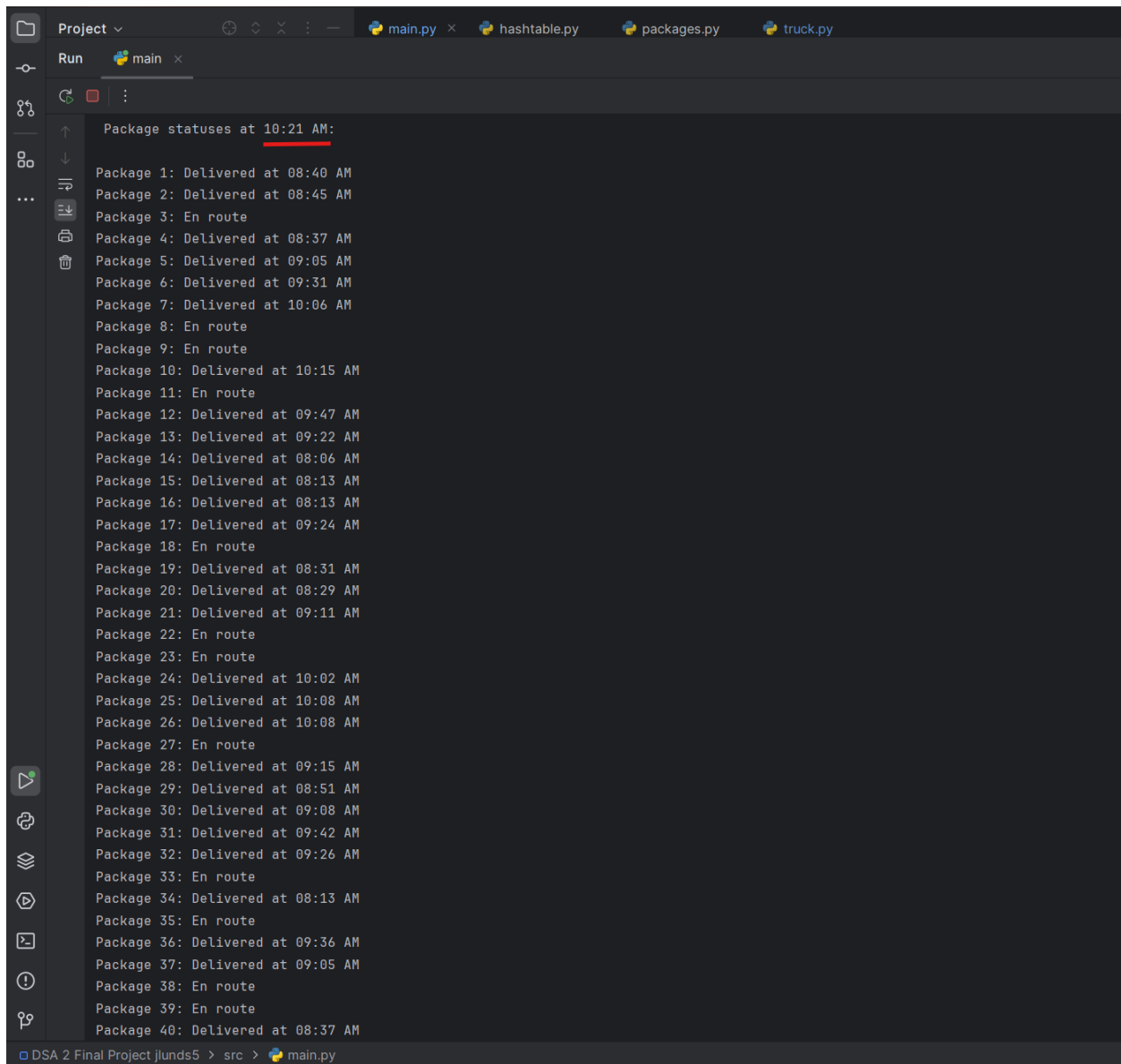
```
Truck 1 Packages: [1, 2, 4, 5, 13, 14, 15, 16, 19, 20, 29, 30, 31, 34, 37, 40]
Truck 2 Packages: [3, 6, 7, 8, 10, 11, 12, 17, 18, 21, 22, 28, 32, 36, 38]
```

```
==== WGUPS Package Tracker ====
1. Get status of a SINGLE package at a specific time
2. Get status of ALL packages at a specific time
3. Show total mileage of all trucks
4. Exit
Enter your choice (1-4): 2
Enter a time (HH:MM, 24-hour format): 08:36
```

Package statuses at 08:36 AM:

Package 1: En route  
Package 2: En route  
Package 3: At hub  
Package 4: En route  
Package 5: En route  
Package 6: At hub  
Package 7: At hub  
Package 8: At hub  
Package 9: En route  
Package 10: At hub  
Package 11: At hub  
Package 12: At hub  
Package 13: En route  
Package 14: Delivered at 08:06 AM  
Package 15: Delivered at 08:13 AM  
Package 16: Delivered at 08:13 AM  
Package 17: At hub  
Package 18: At hub  
Package 19: Delivered at 08:31 AM  
Package 20: Delivered at 08:29 AM  
Package 21: At hub  
Package 22: At hub  
Package 23: En route  
Package 24: En route  
Package 25: En route  
Package 26: En route  
Package 27: En route  
Package 28: At hub  
Package 29: En route  
Package 30: En route  
Package 31: En route  
Package 32: At hub  
Package 33: En route  
Package 34: Delivered at 08:13 AM  
Package 35: En route  
Package 36: At hub  
Package 37: En route  
Package 38: At hub  
Package 39: En route  
Package 40: En route

2. Provide screenshots to show the status of *all* packages loaded onto *each* truck at a time between 9:35 a.m. and 10:25 a.m.



The screenshot shows a Python IDE with a terminal window displaying the output of a program. The terminal output lists the status of 40 packages at 10:21 AM. The status for each package is either 'Delivered' or 'En route', along with the time of delivery or status check. The packages are numbered 1 through 40. The IDE interface includes a file explorer on the left, a run button, and a status bar at the bottom indicating the current file is 'main.py' in the 'src' directory of a project named 'DSA 2 Final Project jlungs5'.

```
Package statuses at 10:21 AM:
Package 1: Delivered at 08:40 AM
Package 2: Delivered at 08:45 AM
Package 3: En route
Package 4: Delivered at 08:37 AM
Package 5: Delivered at 09:05 AM
Package 6: Delivered at 09:31 AM
Package 7: Delivered at 10:06 AM
Package 8: En route
Package 9: En route
Package 10: Delivered at 10:15 AM
Package 11: En route
Package 12: Delivered at 09:47 AM
Package 13: Delivered at 09:22 AM
Package 14: Delivered at 08:06 AM
Package 15: Delivered at 08:13 AM
Package 16: Delivered at 08:13 AM
Package 17: Delivered at 09:24 AM
Package 18: En route
Package 19: Delivered at 08:31 AM
Package 20: Delivered at 08:29 AM
Package 21: Delivered at 09:11 AM
Package 22: En route
Package 23: En route
Package 24: Delivered at 10:02 AM
Package 25: Delivered at 10:08 AM
Package 26: Delivered at 10:08 AM
Package 27: En route
Package 28: Delivered at 09:15 AM
Package 29: Delivered at 08:51 AM
Package 30: Delivered at 09:08 AM
Package 31: Delivered at 09:42 AM
Package 32: Delivered at 09:26 AM
Package 33: En route
Package 34: Delivered at 08:13 AM
Package 35: En route
Package 36: Delivered at 09:36 AM
Package 37: Delivered at 09:05 AM
Package 38: En route
Package 39: En route
Package 40: Delivered at 08:37 AM
```

3. Provide screenshots to show the status of *all* packages loaded onto *each* truck at a time between 12:03 p.m. and 1:12 p.m.

Project

main.py × hashtable.py packages.py truck.py

Run main ×

↻

⏏

⋮

↑

↓

↔

⇅

🖨

🗑

▶

🔗

📁

🔍

📄

⚠

🔧

Package statuses at 01:00 PM:

Package 1: Delivered at 08:40 AM

Package 2: Delivered at 08:45 AM

Package 3: Delivered at 10:24 AM

Package 4: Delivered at 08:37 AM

Package 5: Delivered at 09:05 AM

Package 6: Delivered at 09:31 AM

Package 7: Delivered at 10:06 AM

Package 8: Delivered at 10:26 AM

Package 9: Delivered at 10:41 AM

Package 10: Delivered at 10:15 AM

Package 11: Delivered at 11:07 AM

Package 12: Delivered at 09:47 AM

Package 13: Delivered at 09:22 AM

Package 14: Delivered at 08:06 AM

Package 15: Delivered at 08:13 AM

Package 16: Delivered at 08:13 AM

Package 17: Delivered at 09:24 AM

Package 18: Delivered at 11:03 AM

Package 19: Delivered at 08:31 AM

Package 20: Delivered at 08:29 AM

Package 21: Delivered at 09:11 AM

Package 22: Delivered at 11:29 AM

Package 23: Delivered at 11:23 AM

Package 24: Delivered at 10:02 AM

Package 25: Delivered at 10:08 AM

Package 26: Delivered at 10:08 AM

Package 27: Delivered at 11:00 AM

Package 28: Delivered at 09:15 AM

Package 29: Delivered at 08:51 AM

Package 30: Delivered at 09:08 AM

Package 31: Delivered at 09:42 AM

Package 32: Delivered at 09:26 AM

Package 33: Delivered at 10:24 AM

Package 34: Delivered at 08:13 AM

Package 35: Delivered at 11:00 AM

Package 36: Delivered at 09:36 AM

Package 37: Delivered at 09:05 AM

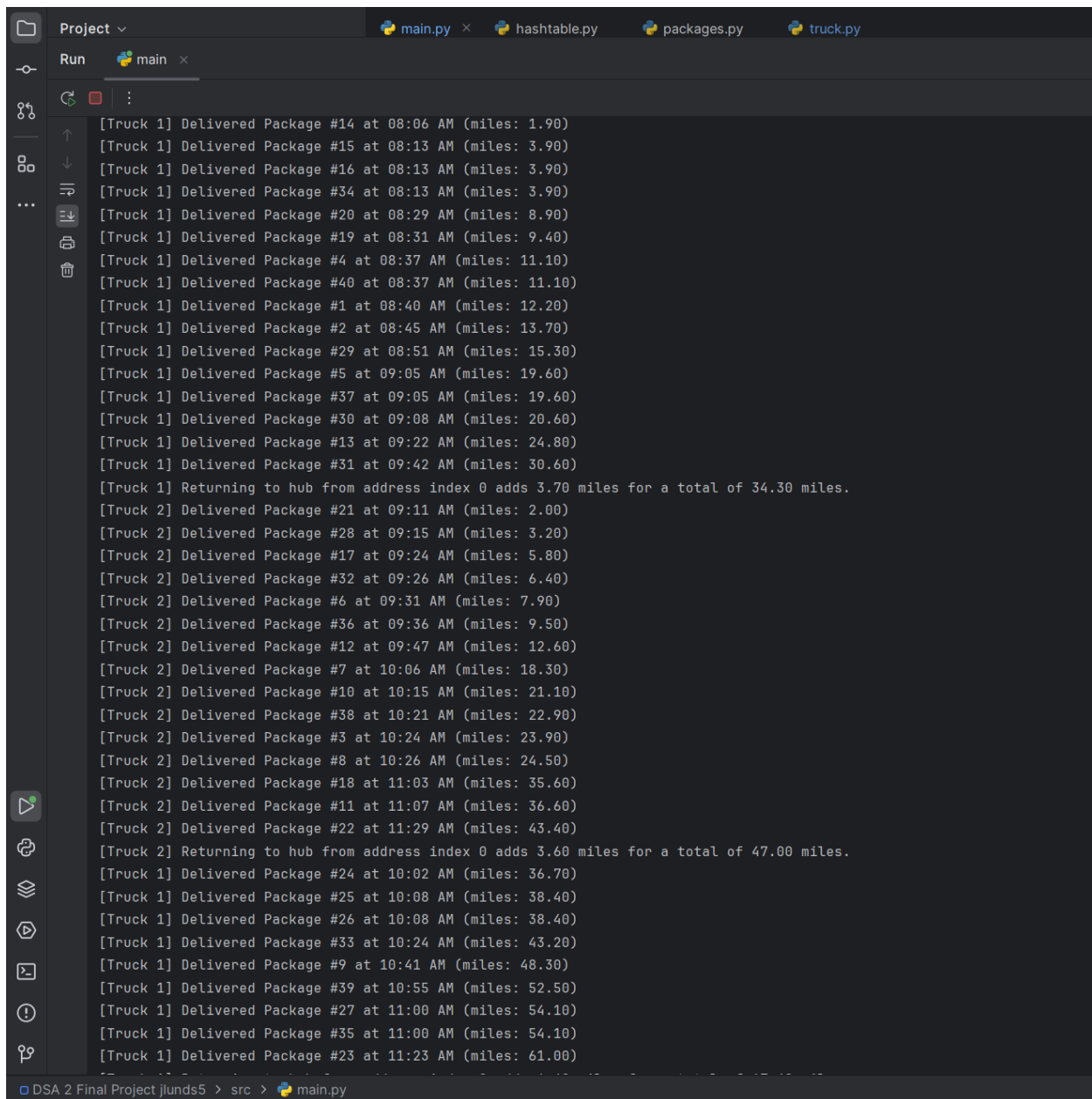
Package 38: Delivered at 10:21 AM

Package 39: Delivered at 10:55 AM

Package 40: Delivered at 08:37 AM

DSA 2 Final Project jlunds5 > src > main.py

E. Provide screenshots showing successful completion of the code that includes the total mileage traveled by *all* trucks.



The screenshot shows a Python IDE with a project named "Project" and four open files: `main.py`, `hashtable.py`, `packages.py`, and `truck.py`. The `main.py` file is active, and the "Run" button is highlighted. The output console displays the following text:

```
[Truck 1] Delivered Package #14 at 08:06 AM (miles: 1.90)
[Truck 1] Delivered Package #15 at 08:13 AM (miles: 3.90)
[Truck 1] Delivered Package #16 at 08:13 AM (miles: 3.90)
[Truck 1] Delivered Package #34 at 08:13 AM (miles: 3.90)
[Truck 1] Delivered Package #20 at 08:29 AM (miles: 8.90)
[Truck 1] Delivered Package #19 at 08:31 AM (miles: 9.40)
[Truck 1] Delivered Package #4 at 08:37 AM (miles: 11.10)
[Truck 1] Delivered Package #40 at 08:37 AM (miles: 11.10)
[Truck 1] Delivered Package #1 at 08:40 AM (miles: 12.20)
[Truck 1] Delivered Package #2 at 08:45 AM (miles: 13.70)
[Truck 1] Delivered Package #29 at 08:51 AM (miles: 15.30)
[Truck 1] Delivered Package #5 at 09:05 AM (miles: 19.60)
[Truck 1] Delivered Package #37 at 09:05 AM (miles: 19.60)
[Truck 1] Delivered Package #30 at 09:08 AM (miles: 20.60)
[Truck 1] Delivered Package #13 at 09:22 AM (miles: 24.80)
[Truck 1] Delivered Package #31 at 09:42 AM (miles: 30.60)
[Truck 1] Returning to hub from address index 0 adds 3.70 miles for a total of 34.30 miles.
[Truck 2] Delivered Package #21 at 09:11 AM (miles: 2.00)
[Truck 2] Delivered Package #28 at 09:15 AM (miles: 3.20)
[Truck 2] Delivered Package #17 at 09:24 AM (miles: 5.80)
[Truck 2] Delivered Package #32 at 09:26 AM (miles: 6.40)
[Truck 2] Delivered Package #6 at 09:31 AM (miles: 7.90)
[Truck 2] Delivered Package #36 at 09:36 AM (miles: 9.50)
[Truck 2] Delivered Package #12 at 09:47 AM (miles: 12.60)
[Truck 2] Delivered Package #7 at 10:06 AM (miles: 18.30)
[Truck 2] Delivered Package #10 at 10:15 AM (miles: 21.10)
[Truck 2] Delivered Package #38 at 10:21 AM (miles: 22.90)
[Truck 2] Delivered Package #3 at 10:24 AM (miles: 23.90)
[Truck 2] Delivered Package #8 at 10:26 AM (miles: 24.50)
[Truck 2] Delivered Package #18 at 11:03 AM (miles: 35.60)
[Truck 2] Delivered Package #11 at 11:07 AM (miles: 36.60)
[Truck 2] Delivered Package #22 at 11:29 AM (miles: 43.40)
[Truck 2] Returning to hub from address index 0 adds 3.60 miles for a total of 47.00 miles.
[Truck 1] Delivered Package #24 at 10:02 AM (miles: 36.70)
[Truck 1] Delivered Package #25 at 10:08 AM (miles: 38.40)
[Truck 1] Delivered Package #26 at 10:08 AM (miles: 38.40)
[Truck 1] Delivered Package #33 at 10:24 AM (miles: 43.20)
[Truck 1] Delivered Package #9 at 10:41 AM (miles: 48.30)
[Truck 1] Delivered Package #39 at 10:55 AM (miles: 52.50)
[Truck 1] Delivered Package #27 at 11:00 AM (miles: 54.10)
[Truck 1] Delivered Package #35 at 11:00 AM (miles: 54.10)
[Truck 1] Delivered Package #23 at 11:23 AM (miles: 61.00)
```

The status bar at the bottom indicates the file path: `DSA 2 Final Project jlunds5 > src > main.py`.

```
[Truck 1] Delivered Package #39 at 10:55 AM (miles: 52.50)
[Truck 1] Delivered Package #27 at 11:00 AM (miles: 54.10)
[Truck 1] Delivered Package #35 at 11:00 AM (miles: 54.10)
[Truck 1] Delivered Package #23 at 11:23 AM (miles: 61.00)
[Truck 1] Returning to hub from address index 0 adds 6.40 miles for a total of 67.40 miles.
```

```
==== WGUPS Package Tracker ====
```

1. Get status of a SINGLE package at a specific time
2. Get status of ALL packages at a specific time
3. Show total mileage of all trucks
4. Exit

```
Enter your choice (1-4): 3
```

```
Mileage Summary:
```

```
Truck 1 mileage: 67.40 miles
```

```
Truck 2 mileage: 47.00 miles
```

```
Total mileage for all trucks: 114.40 miles
```

**F. Justify the package delivery algorithm used in the solution as written in the original program by doing the following:**

**1. Describe two or more strengths of the algorithm used in the solution.**

I chose to implement the nearest neighbor algorithm because, as experts have described, it is “known for its simplicity and effectiveness” (Taunk et al., 2019). Its simplicity is one of its biggest strengths, as the simplicity makes it easy to code as well as update when bugs or additional requirements arise. Another strength is its speed, particularly for smaller datasets. While the algorithm’s polynomial time ( $O(n^2)$ ) does mean it will be slow for massive amounts of calculation (for example, if there were hundreds of thousands of packages), it is well suited for cases like WGUPS where we only have a maximum of 40 packages. A third strength, is its ability to self-adjust dynamically—as each package is delivered, the algorithm recalculates to determine what the next best route is in real time (Taunk et al., 2019).

**2. Verify that the algorithm used in the solution meets *all* requirements in the scenario.**

As can be seen in the screenshots in parts D and E, the algorithm satisfies all the requirements in the scenario. All 40 packages are delivered, within the 140-mile constraint, while still factoring in special cases such as package #9 (which could not be delivered until after 10:20am when the address is known) and meeting other tight deadlines. All of this was accomplished despite loading no more than 16 packages per truck and keeping the speed at 18 mph. This was accomplished by reusing Truck 1. Truck 1 is reused after its first delivery run because WGUPS only has two drivers available. This allows Truck 1 to make an additional trip while Truck 2 is out with delayed and restricted packages. This reuse ensures all packages are delivered under the 140-mile requirement without violating any delivery or driver constraints. Lastly, the program provides a nice user interface with 4 easy-to-select options that provides all the outputs required (delivery times, the status of a single package at a specific time, the statuses of all packages, etc.).

**3. Identify two other named algorithms that are different from the algorithm implemented in the solution and would meet *all* requirements in the scenario.**

The first and simplest alternative algorithm would be the brute force algorithm. This involves calculating every possible route, and selecting the one with the absolute shortest path. This would provide the optimal route in terms of mileage. It could also be adapted to satisfy the cases with tight deadlines or special notes, because you could write logic that pre-filters valid permutations that satisfy all the deadlines and special notes (such as delivering #9 after 10:20am, grouping packages that must be delivered together, and meeting 9:00am or 10:30am deadlines). One could also immediately reject any permutations that violate delivery windows or any other constraints. The second algorithm, Dijkstra's algorithm, would satisfy all the requirements by finding the shortest paths from the hub to all addresses — or between any two points (Javaid, 2013). This would help to minimize the travel between stops and meet the less than 140-mile threshold. Furthermore, Dijkstra's algorithm can be combined with constraint logic to meet special requirements such as the strict deadlines or the special notes (grouping packages, or delivering them after a certain timeframe) (Javaid, 2013).

**a. Describe how *both* algorithms identified in part F3 are different from the algorithm used in the solution.**

The brute force algorithm and Dijkstra's algorithm are significantly different from the nearest neighbor algorithm used in this solution. The brute force method works by calculating every possible route, and selecting the one with the lowest mileage. This guarantees that the optimal route will be found. However, it is very impractical as the number of computations in the worst case scenario is  $O(n!)$ , meaning that if the number of packages was higher it would take an enormous amount of computational effort to find the best route. This means that it does not scale well, and could not be used for more than a handful of packages. This is in stark contrast to the nearest neighbor's  $O(n^2)$ , which still does not scale immensely well, but is far more scalable



than  $O(n!)$  (Taunk et al., 2019). Dijkstra's algorithm is also very different, as it calculates all shortest paths from a starting node rather than building a full delivery route step by step, which is denoted by its Big O notation of  $O((V + E) \log V)$ . This means that it is better suited for optimizing individual legs of a trip, rather than chaining them together (Javaid, 2013) to find an overall best route like we do for the package deliveries. If Dijkstra's algorithm was implemented here, it would have a Big O of  $O(n \times (V + E) \log V)$  since we are dealing with routes for the 40 packages. Thus, while both algorithms have their uses, the nearest neighbor algorithm is a better choice for the WGUPS assignment.

**G. Describe what you would do differently, other than the two algorithms identified in part F3, if you did this project again, including details of the modifications that would be made.**

One thing I would do differently in the future is to group packages by their deadlines or special notes so that it would be easier to load them onto the truck. The solution I implemented involved me handpicking packages based on their deadline or special notes needs and hardcoding their package IDs into the "load\_packages" method. This works fine in this case, where we only have about 2 dozen packages with tight deadlines or special notes. But had there been hundreds or thousands of packages with such constraints, hardcoding them into the function would have been a nightmare. Thus, by writing a function that groups them according to their needs, the program would become much more scalable. For example, packages that can only be loaded onto truck 2, like packages 3 and 18, would be classified in their own group (which we would implement as a list) and then load the entire list, rather than hard coding the values in. Additionally, I would also implement a more thorough, detailed log of delivery events that would provide the user with a very detailed interface, and simplify my main.py file (which is currently a bit long) by modularizing it. For example, I'd create a "Scheduler" or "Dispatcher" class in a separate module that handles truck rotations, prioritization, and status checking, and call that class in the main.py file.

**H. Verify that the data structure used in the solution meets *all* requirements in the scenario.**

The custom-built hash table I created for this project fully meets the requirements of the WGUPS routing system. The hash table effectively stores all 40 packages, using their IDs as unique keys and ensuring a fast  $O(1)$  time for lookup, insertion and updating values—all of which is critical for having a fast retrieval of delivery status. Furthermore, by using an OOP approach each item in the Hash Table also has all of the necessary data fields (delivery address, deadline, city, zip code, weight, and status). By using a list of buckets (chaining), the structure is also easy to implement and modify. Lastly, because it is built from scratch, it does not utilize Python's built-in dictionaries—meeting one of the most important of the course requirements.

**1. Identify two other data structures that could meet the same requirements in the scenario.**

A list (or “array” in other languages) or binary search tree could have been utilized to meet the same requirements.

**a. Describe how *each* data structure identified in H1 is different from the data structure used in the solution.**

A BST maintains order among elements, which is helpful for sorted views or range queries (unlike hash tables which have no index). However, it is more complex to implement and less efficient than a hash table for simple key-based lookups (Karimov, 2020). In comparison, the lookup, insertion, and deletion time complexity in a binary search tree is of  $O(\log n)$ , compared to the  $O(1)$  for hash tables. While binary search trees are certainly efficient, in this case the hash table has the advantage. A list (or array), on the other hand, also has the advantage of preserving the order of an object’s insertion by keeping track of the index. Lists are also very simple to implement, however they can be inefficient because they would store all of the package objects in a single list and then traverse the list in order to find a package by its ID. This would make for a time complexity of  $O(n)$ , which is much slower than a hash table’s  $O(1)$ . While this would work fine for this case of only 40 packages, if there were hundreds or thousands of packages the runtime would be much slower. Thus, if there are many packages or a need to frequently lookup packages, the list data structure is not an optimal choice.

**I. Sources**

Javaid, M. A. (2013). Understanding dijkstra algorithm. *SSRN Electronic Journal*.  
<https://doi.org/10.2139/ssrn.2340905>

Karimov, E. (2020). Data Structures and algorithms in swift: Implement stacks, queues, dictionaries, and lists in your apps. Apress.

Taunk, K., De, S., Verma, S., & Swetapadma, A. (2019). A brief review of nearest neighbor algorithm for learning and classification. *2019 International Conference on Intelligent Computing and Control Systems (ICCS)*, 1255–1260.  
<https://doi.org/10.1109/iccs45141.2019.9065747>