Shawn Neville

Student ID: 001158357

**Algorithm Overview**

**Section 1: Programming/Coding**

**A:** **Algorithm Selection:**

I chose to a greedy algorithm called Dijkstra’s as my core algorithm to solve the traveling salesman problem. See algorithm.py for more details.

**B1:** **Logic Comments:**

Comments in source code align with industry standards. Each of the functions in source code have been commented on to explain the logic applied to the solution. See python files in project folder for further detail.

**B2: Application of Programming Models:**

This Python program runs on local machines only. The project folder contains two CSV files, WGUPS Distance Table.csv and WGUPS Package File.csv that contain the package and location data required for the program to run. Because these files are stored locally on the machine that will host the program. No network communication protocols are required. The project was developed in the PyCharm IDE using the Python programming language.

**B3: Space-Time and Big O:**

Each block of code in project source code files include comments that detail the Space-time complexity using Big O notation.

Big O notation for entire program is O(n^3)

**B4: Adaptability:**

The algorithm I chose has an asymptotic behavior of O(E log V) This means the algorithm scales well as larger and larger data lists are inputted. There may be a noticeable increase in run time as input data gets very large. However, it runs on one truck delivery list at a time so it could still be an optimal solution for real world application.

**B5: Software Efficiency and Maintainability:**

Even though the program run time complexity is O(n^3) I think the efficiency is good enough. Especially given that there are only forty packages in this scenario. It is also unlikely that a truck would be able to carry enough packages to notice the exponential asymptotic behavior of the algorithm. Almost all my code is structured into class objects and functions. This makes for changes in fewer places in the code allowing easy modification and maintenance.

**B6: Self-Adjusting Data Structures:**

The self-adjusting data structure in the program is a hash table. It can adapt to data change. It can adjust to new data being added, removed, or searched. When this data structure is accessed it affects running time with a linear run time complexity because it’s Big O is: O(n)

**C1: Identification Information:**

Name, and student ID are located within first line of code in each file.

**C2: Process and Flow Comments:**

Each block of code in files includes comments to satisfy this requirement.

**D: Data Structure:**

A Hash Table data structure performs well to the usage described in the project scenario because it allows user to store unordered items (packages) in key-value pairs. My hash table allows for very quick insertion time O(1). The deletion and look up functions have a run time complexity of O(n). Due to the size of input data in the scenario I still think it’s an optimal solution.

**D1: Explanation of Data Structure:**

The Hash Table data structure I built is used to store package data in key value pairs. This is done by hashing each package ID to a location in an array. Because each package ID is unique, there are no collisions. This means there are no duplicate hashes produced. Insertion function is performed in constant time O(1). Deletion and look up are performed in linear run time complexity O(n).

**E: Hash Table:**

The hash table I created includes an insert function that uses the package ID to hash it to a location/bucket in a list. See file “hashmap.py”

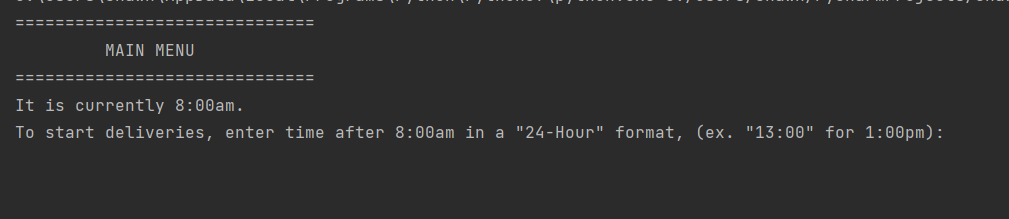
**F: Look-Up Function:**

The lookup function takes in the package ID as a parameter and allows the user to view all data elements for a package. See line 50 in file “main.py”

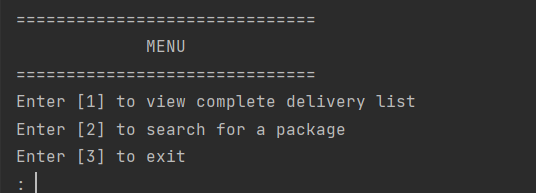
**G: Interface:**

The console interface I created provides comprehensive menus for the user to easily use the program. See “main.py” for further detail.

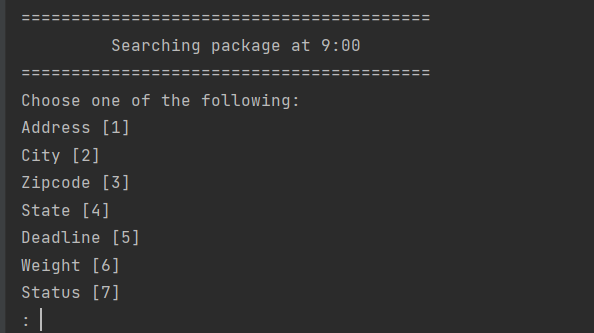
This first menu prompts user to enter a time in 24-hour format. This is used to display a list of a packages and their status for the inputted time.



The sub menu allows user to view complete delivery list of packages, option to search for package information, or exit the program.

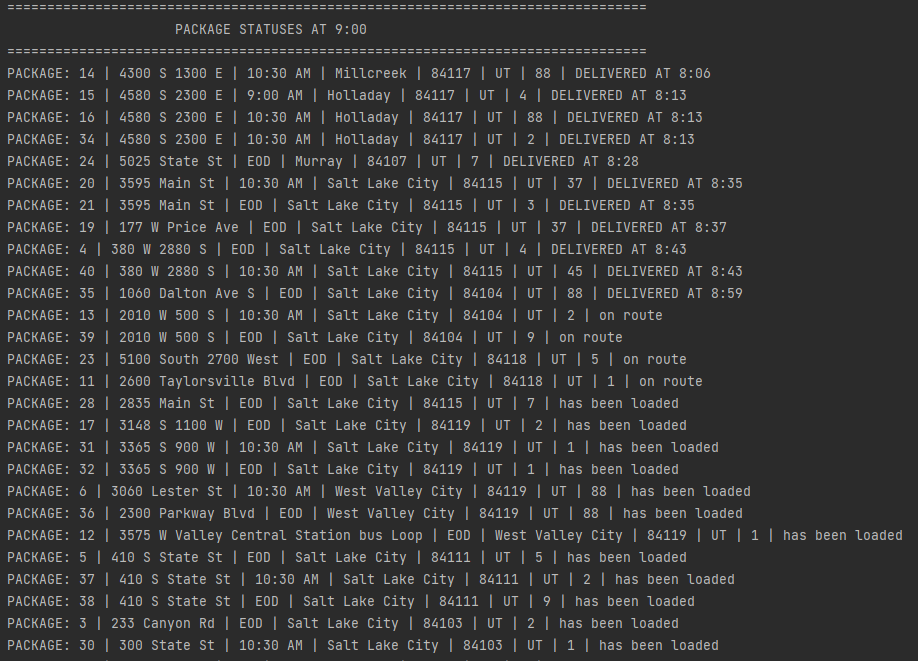


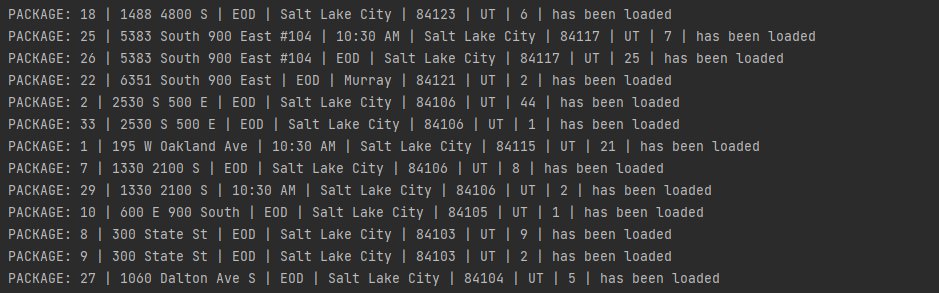
If the user chooses to search for a package, they are giving additional options after entering package ID to search for specific package information. After search is complete. User is brought back to sub menu



**G1: First Status Check:**

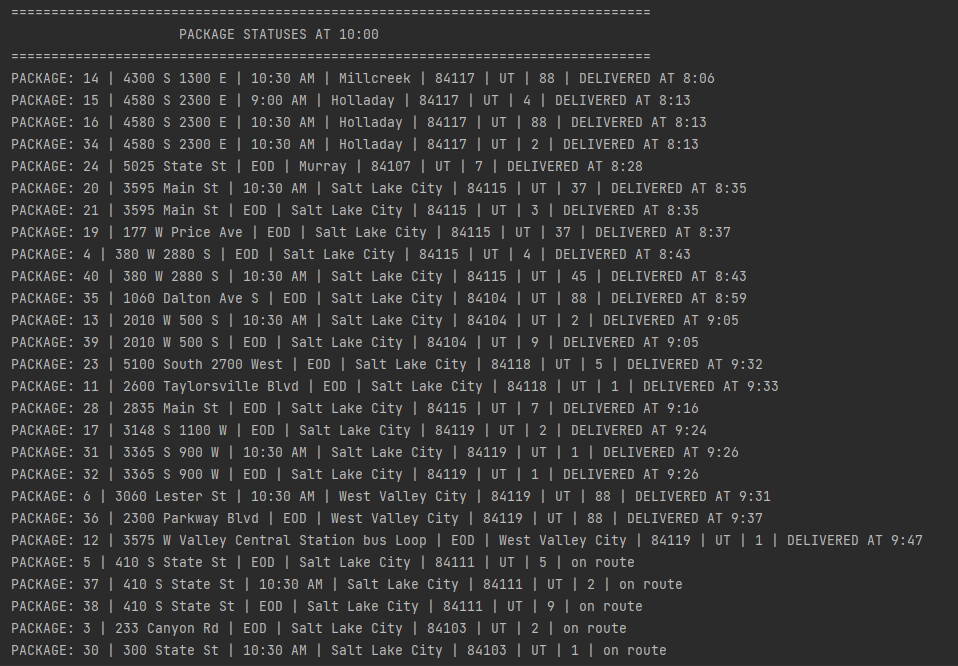
Listing of all packages at 9:00a.m. (between 8:35 a.m. and 9:25 a.m.)

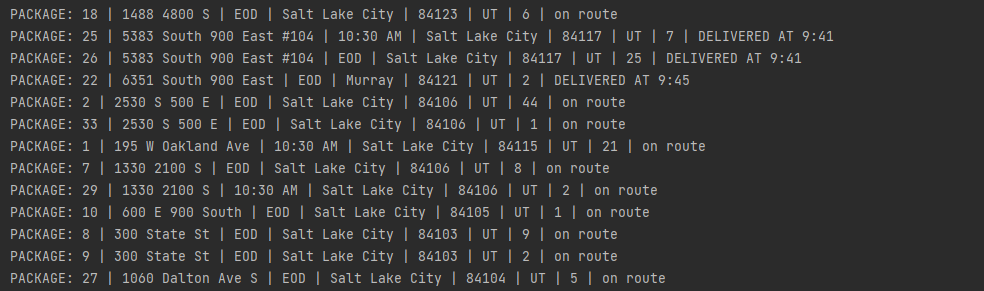




**G2: Second Status Check:**

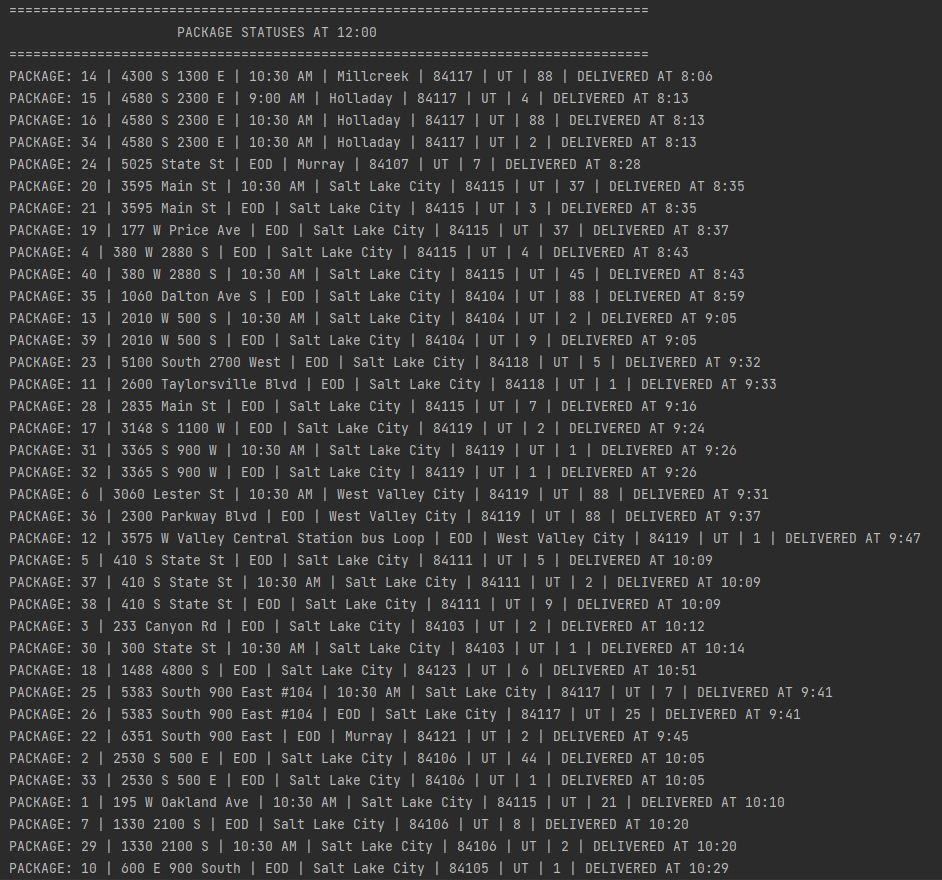
Listing of all packages at 10:00a.m. (between 9:35 a.m. and 10:25 a.m.)

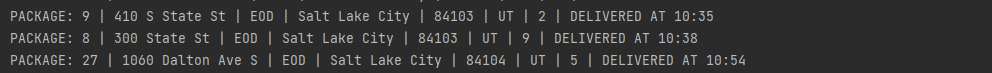




**G3:Third Status Check:**

Listing of all packages at 12:00p.m. (between 12:03p.m. and 1:12p.m.)

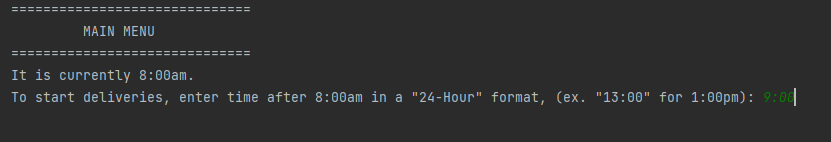




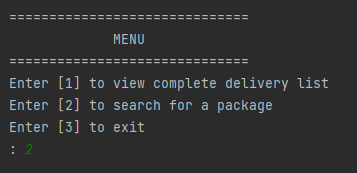
**H: Screenshots of Code Execution:**

Theses screenshots capture a complete execution of the code.

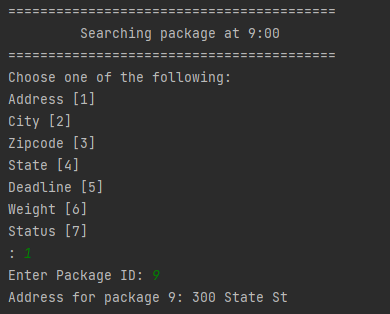
First menu prompting for time. I enter 9:00a.m.



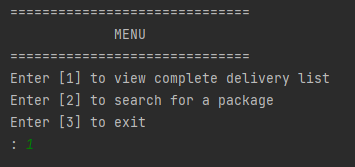
After listing of all packages at 9:00a.m. are displayed. Sub menu prompts for menu choice. I enter 2 to search for a package.



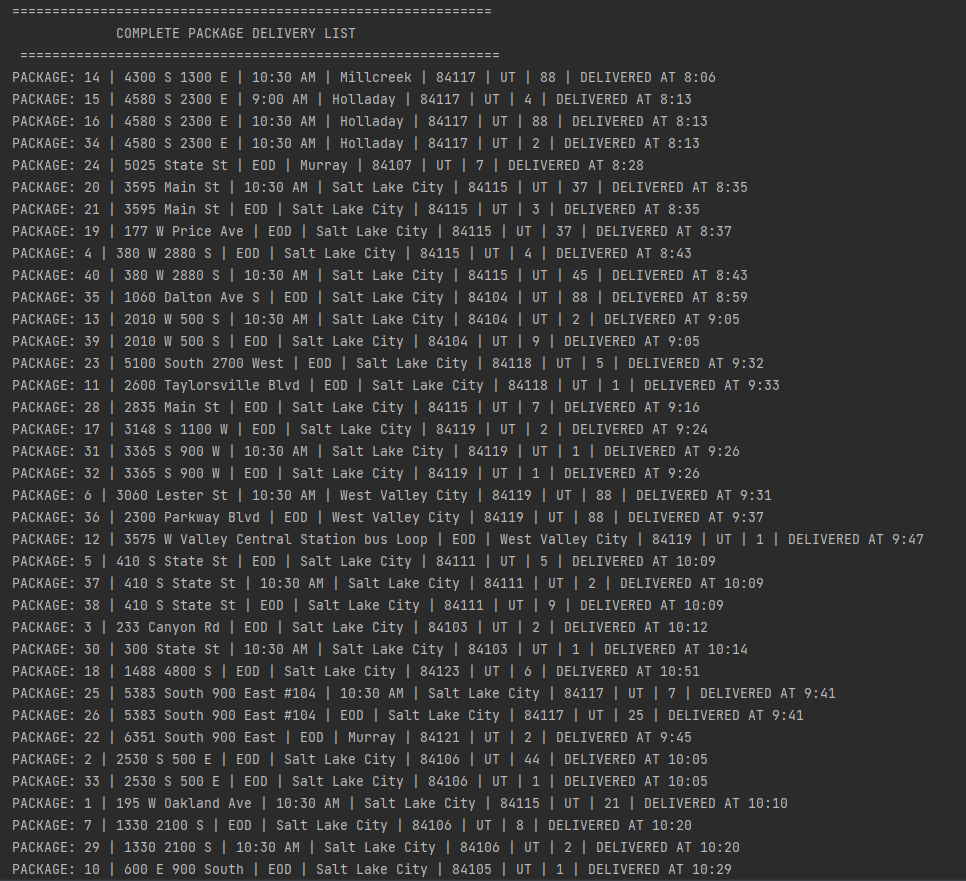
Search menu prompts for search parameter. I enter 1 for address followed by package ID 9. Address for package 9 is displayed.

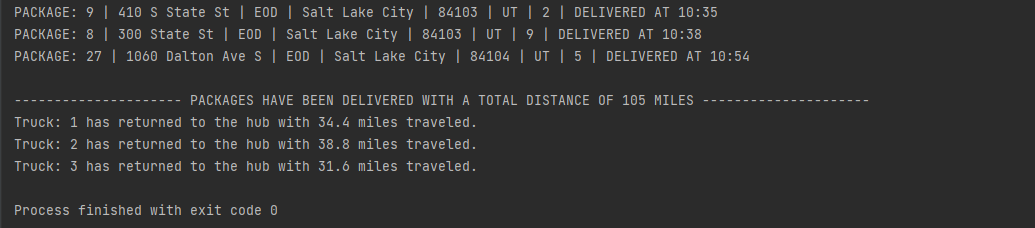


I am returned to sub menu. I enter 1 to view listing of all packages once delivered.



Complete package delivery list is displayed with total milage and each truck’s traveled miles. Program finishes and exits.





**Section 2: Annotations**

**I1: Strengths of the Chosen Algorithm:**

The two strengths of Dijkstra’s greedy algorithm are:

It works well in this project scenario because it’s able to find the optimal/shortest path between any two vertices in the graph.

Another strength is its run time complexity of O(E log V). This means the algorithm should scale relatively well as larger data sets are inputted. In the real-world trucks are unlikely to hold enough packages that would cause notice of increased runtime.

**I2: Verification of Algorithm:**

Because the algorithm selects locations with the shortest distance, it meets the criteria and aligns with assumptions in the project scenario. This is demonstrated in screenshots provided in G-H where delivery timestamps, total miles, and truck miles are displayed.

**I3: Other Possible Algorithms:**

Two other algorithms that could also meet the requirements of the scenario are:

* **Prim’s Algorithm**: Prim’s finds the minimal spanning tree whereas Dijkstra’s finds the shortest path tree.
* **Bellman-Ford’s algorithm**: Finds the shortest path to every vertex in a graph from a single vertex.

**I3A: Algorithm Differences:**

* **Prim’s Algorithm** differs from Dijkstra’s because rather than finding the shortest path tree it finds the minimal spanning tree. It’s also different because It builds a tree by choosing the cheapest weighted edge as it traverses the graph. Dijkstra’s accounts for future paths and works backwards to build the shortest path.
* **Bellman-Ford’s algorithm** differs from Prim’s and Dijkstra’s because it supports graphs with negative edge weights. It also differs because it has a run time of O(EV)

**J: Different Approach:**

If I were to do this project again, I would probably use Prim’s algorithm because I think it would have been easier to implement. I would also find a way to dynamically sort packages and load them on trucks based on the restrictions the packages have in the CSV files.

**K1: Verification of Data Structure:**

The total miles are added to trucks, all packages are delivered on time by the program. The Hash Table I created has look-up function and reporting is accurate. Screenshots from G-H also validate this.

**K1A: Efficiency:**

A Hash Table data structure performs well enough in this project scenario because there are only 3 trucks and a total of forty packages. The time complexity for my Hash Table search and delete function is O(n). This means that time will scale linearly as more packages are added to the program.

**K1B: Overhead:**

Computational time will increase linearly with the size of input data due to the hash table look-up function having run time complexity of O(n). Memory usage would also increase as the amount of input data added to the hash table increases. Bandwidth is not a concern as data files are stored locally and network communication is not required for this program.

**K1C: Implications:**

Because my program has an overall run time complexity of O(n^3), upscaling the program with vast amounts of input data such as cities, trucks, and packages could be a concern for performance. Further optimization would be desirable. Finding ways to avoid nested loops would help. Data structures such as binary search trees would be ideal because their functions have run time complexities of O(log n)

**K2: Other Data Structures:**

Two other data structures that could be used in the solution to meet all the requirements are binary search trees and python dictionaries

Queues

stacks

**K2A: Data Structure Differences:**

**A Stack:** is a LIFO (last in first out) data structure. This means the last item to go in the stack is the first to go out. This data structure could possible be used to prioritize packages. Such as putting packages that have the latest deadline in the stack first.

**Binary Search Trees:** reference other nodes through parent and child relationships. Each node in a Binary search Tree has at most two child nodes references. This makes it take on a tree like structure. Because Binary Search Trees are balanced and pre-ordered, operations on this structure are faster than lists having an average run time of O(log n).

**L: Sources:**

Learn.zybooks.com. (n.d.). *zyBooks*. [online] Available at: <https://learn.zybooks.com/zybook/WGUC950AY20182019>

James, J. [Joe James] (2015, Jan. 14). *Bellman-Ford Single-Source Shortest-Path algorithm*. [Video]. YouTube. <https://www.youtube.com/watch?v=dp-Ortfx1f4&list=PLj8W7XIvO93oxLOZTi8JFghuRcKieIZU-&index=7>

James, J. [Joe James] (2014, May. 22). *Prims Algorithm for Minimum Spanning Trees*. [Video]. YouTube. <https://www.youtube.com/watch?v=MaaSoZUEoos&list=PLj8W7XIvO93oxLOZTi8JFghuRcKieIZU-&index=5>

James, J. [Joe James] (2015, Jan. 22). *Binary Search Trees (BST) Explained in Animated Demo*. [Video]. YouTube. <https://www.youtube.com/watch?v=mtvbVLK5xDQ>