Package Delivery Program

David Krupar

A. Algorithm Identification

The self-adjusting algorithm used to create the program is known as Dijkstra's algorithm. It is historically known as being one of the first algorithms to solve the “traveling salesman” problem with some degree of accuracy. This algorithm performs the task of delivering all packages.

B1. Logic Comments

The algorithm logic is described in great detail in the source code. Please refer to function “nearest_neighbor” in the Data.py file. Throughout the program, extensive detail to the logic used in the various functions are noted and explained with no inaccuracies. Pseudo code…. An array is created for temporary storage of loading data. The algorithm checks for pending packages that have not been delivered and adds them to the temporary array. The temporary array is wiped on each package delivery. While the pending deliveries are >0 the algorithm pulls data from the csv reader files. The distances are compared between the addresses, and the least distance is chosen. The package is then put into the load order and then delivered. The algorithm finishes up with a update of the vehicles distance and timestamps.

```
# Method for parcel sorting per vehicle using nearest neighbor algorithm
# Also used to calculate distance per vehicle
def nearest_neighbor(vehicle):
    # Array created for parcels not delivered
    pending_d = []
    for parcel in vehicle.first_order:
        p = Data.p_ht.get_val(parcel)
        pending_d.append(p)
    # Wipes individual vehicle parcel ordering
    vehicle.first_order.clear()

    # Applies algorithm until pending deliveries is 0
    while len(pending_d) > 0:
        n_p = None
        n_a = 2000
        for p in pending_d:
            # Uses CSV files to compute distances and create most efficient ordering
            if xy_differential(pull_data(vehicle.start), pull_data(p.address)) <= n_a:
                # n_a creates an instance of the next address to be used
                n_a = xy_differential(pull_data(vehicle.start), pull_data(p.address))
                # n_p creates an instance of the next parcel to be put in order
                n_p = p

        # Functions to add/remove parcel
        vehicle.first_order.append(n_p.ID)
        pending_d.remove(n_p)
```

```
# Function to compute mileage
vehicle.distance += n_a
# Function to update vehicle address
vehicle.start = n_p.address
# Functions to update timestamps
vehicle.time += datetime.timedelta(hours=n_a / 18)
n_p.del_time = vehicle.time
n_p.dep_time = vehicle.depart_time
```

B2. Development Environment

The development environment was a variety of tools using modern hardware and current software tooling. The program was written using the following software in detail...

PyCharm 2022.2.3 (Professional Edition)

```
Build #PY-222.4345.23, built on October 10, 2022
Licensed to DAVID KRUPAR
Runtime version: 17.0.4.1+7-b469.62 amd64
VM: OpenJDK 64-Bit Server VM by JetBrains s.r.o.
Windows 10 10.0
GC: G1 Young Generation, G1 Old Generation
Memory: 2040M
Cores: 8
```

Modern use of python usually requires a 3rd party set of tools in order to get the specialized result that is required in today's business world. PyCharm is an IDE that was created and operated by a company by the name of JetBrains. WGU has illustrated the power of using JetBrains products, and they have surely helped along the learning journey in software development. The debugging toolsets offered are some of the best in the industry.

From a hardware perspective, the program was created using an Intel i7 4790, 16gb of DDR3 ram, and dual monitors. The extra ram comes in handy when using modern IDE's.

B3. Space-Time and Big-O

A variety of notations have been made in the program comments answering the time and space complexity of the application. However, to simplify, we will discuss them here also. Given our particular delivery driver situation the space and time complexity of the entire program is O(n^2). This particular algorithm and data set are limited by our “truck” count. These are known as vertices, and can be loaded with a large or small amount of “packages”. The fact that there are 3 trucks that we must use to complete our task, this restricts our ability to "3 vertices" from an algorithmic standpoint.

Hashing is a storage technique which mostly concerns itself making searching faster and more efficient. The best case for a hash table search in our program will be the element being found at key location. Therefore, the key must be obtained and then submitted. This is achieved in constant time. So, best case complexity is *O(1)*. The insertion time complexity of the hashtable used is big-O of O(1), the function is located in the "Hash.py" file, listed as "set_val". There are two other functions for the hash table, "delete_val" and "get_val". These are commonly refereed to as delete and lookup. Those two functions both have a time complexity of big-O of O(1). The Space complexity of using a hashtable in this scenario will be big-O of O(n).

For the algorithm of the program, Dijkstras is used, and the Time complexity using big-O notation is O(n^2). The space complexity is also the same, sitting at big-O notation of O(n^2). In the particular exercise, the n accounts for the three vertices, which is this program are the three vehicles. This will be a limiting factor for space and time in future growth of the program. However, there are worse space time complexities, so it will do fine for this scenario.

B4. Scalability and Adaptability

Adaptability and scalability are becoming more important in the modern software working environment. It is best to start a program that is accurate and simple, then mold it into a larger program slowly, which helps maintain code quality and promotes team building. Scalability and adaptability are absolutely critical for this. The algorithm chosen for this program scales well as larger and larger datasets are thrown at it. In this particular case, a larger dataset would include a route delivery with 5,000 packages. Our software could handle such a task with good efficiency. Again, our space and time complexity are based on our "truck count" which then uses a greedy algorithm to assign packages to the correct trucks in the correct order. It was interesting to learn in this project that this algorithm was a very early adoption of this particular real world scenario. There are many other logarithms being discovered currently that are sure to create an interesting field of computer science in the near future.

B5. Software Efficiency and Maintainability

For our program, we chose to stick to the object oriented principles taught to us in past WGU courses. This is critical to the efficiency and maintainability of the codebase. Writing a "good" algorithm into a program can only accomplish so much. We have been taught in various WGU courses that the flexibility and simplicity of the code structure is highly valued. It was interesting to apply those past lessons taught by WGU into a more modern data driven application. Many times these two fields of study do not intersect, and it was fun and exciting to apply those skills and learn how to apply the data field into the software design field.

B6. Self-Adjusting Data Structures

The data structure chosen for this particular project is known as a Hash Table. HashTables are known for being used to provide a quick and efficient solution, and are very relevant in today's software development world. The HashTable is self adjusting as data can be added, removed, and searched from the HashTable. HashTables are great algorithmic tools as they offer a linear time complexity of BigO of O(n), which makes it at the minimum a great starting point when discussing which data structures to use for a particular data set. Potential disadvantages of using hashtables for the exercise is the potential for collisions. This is a common fault of using the hash table data structure. They can also be complex to implement. The biggest disadvantage of using a hash table is that they do not allow null values.

C. Original Code

The code is a result of 80 hours of study and application. It is original, written using the Python language, and free from errors due to extensive testing using the PyCharm debugger toolset.

C1. Identification Information

Per the instruction set and rubric for c950. Main.py has a header which includes the required information.

C2. Process and Flow Comments

The program uses extensive commenting to assist in the grading and also the future use of the codebase. It is critical to offer clean structure, as well as good commenting practices, which will assure that both you and your team will be able to shape the program easily for future use. In the software industry, many times situations and events happen in an instant, there is no time when that occurs to work with a poor quality codebase. The codebase must be built to a standard that allows for immediate and quick responses, which occur all too often in this industry.

D. Data Structure

A variety of Data structures are used in today's software development environments. For this particular scenario, the data structure known as a hash table was selected. The advantage of this type of data structure is that is allows the storage of parcels in key-value pairs. The insertion time is O(1), the function is located in the "Hash.py" file, listed as "set_val". There are two other functions for the hash table, "delete_val" and "get_val". These are commonly refereed to as delete and lookup. Those two functions both have a run time complexity of O(n). This solution is optimized for the scenario given the input size in the task presented.

D1. Explanation of Data Structure

Insertion is performed in contestant time O(1). The other functions "delete_val" and "get_val" are both performed in linear runtime complexity of O(n). Each parcelID is unique, proving an environment that is free from collisions. Using key value pairs, the hash table hashes each parcelID into an array in memory. Prior to the decision, this information was used to consider the relationship between the data points to be stored.

E. Hash Table

Please refer to the "Hash.py" file for the insertion function listed as "set_val". For simplicity I will include it now....

```
def set_val(self, key, item):
    hashed_key = hash(key) % len(self.list)
    bucket_list = self.list[hashed_key]
    for kv in bucket_list:
        if kv[0] == key:
            kv[1] = item
            return True
    key_value = [key, item]
    bucket_list.append(key_value)
    return True
```

F. Look-Up Function

The look-up function is listed in the "Hash.py" file as "get_val". It is most importantly used within the "Main.py" file, below the class "Main", within the user interface code, as it is vital to the purpose of the program. Listed below is the look-up function, which can be found in the "Hash.py" file.

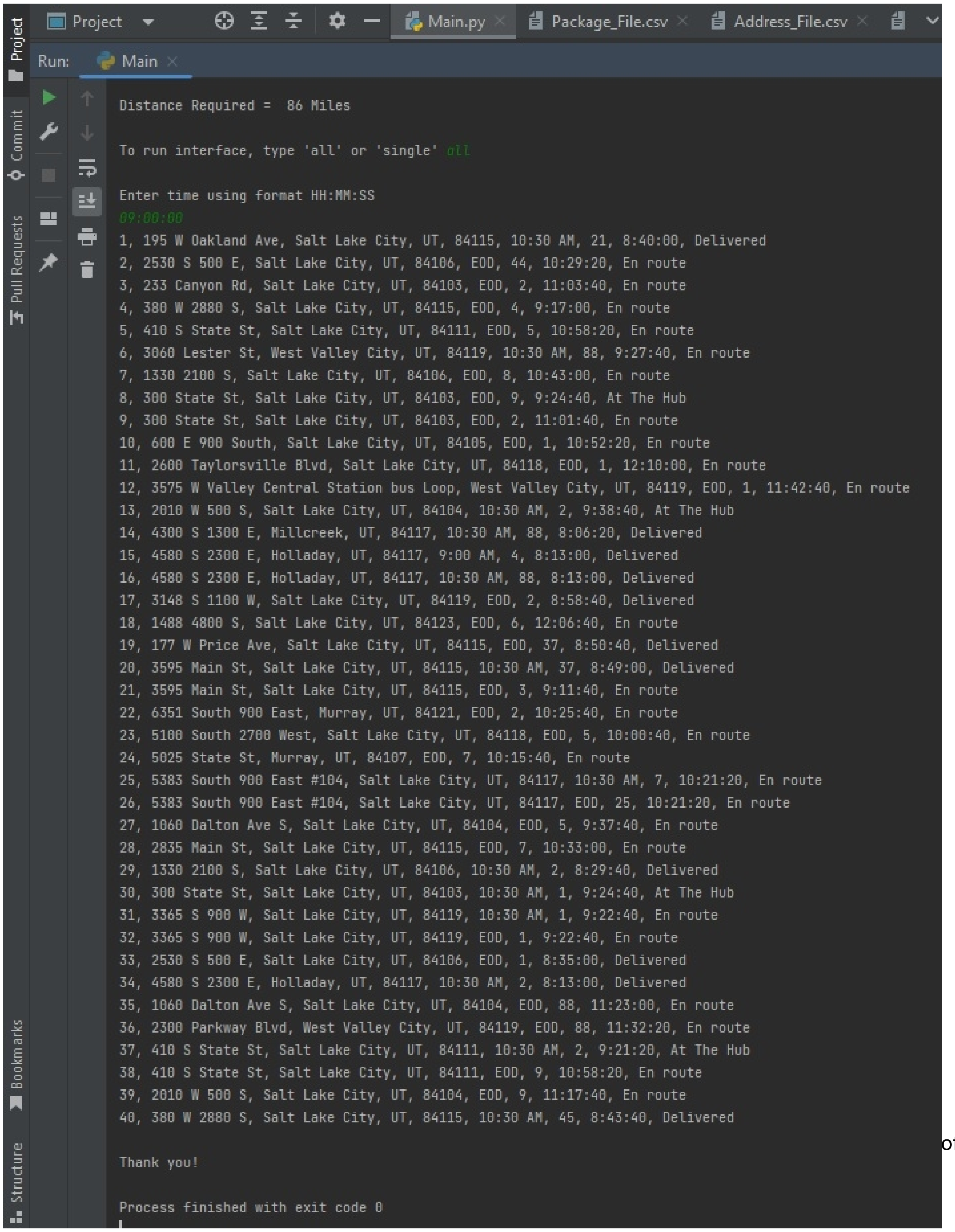
```
def get_val(self, key):
    hashed_key = hash(key) % len(self.list)
    bucket_list = self.list[hashed_key]
    for pair in bucket_list:
        if key == pair[0]:
            return pair[1]
    return None
```

G. Interface

The concept for the interface was to provide a simple and clean source of information for the user. It was decided not to clutter the program with menus and options. The program always gives the “miles needed” as an opening remark, this helps save time and makes for quick adjustments in the parcel assignments if needed. The program requests the user to decide on “all” or “single”, referring to parcel data inquiry. Once selected, the user must enter a time in the format clearly listed within the interface. After inserting the time, the data is drawn and presented to the user.

G1. First Status Check

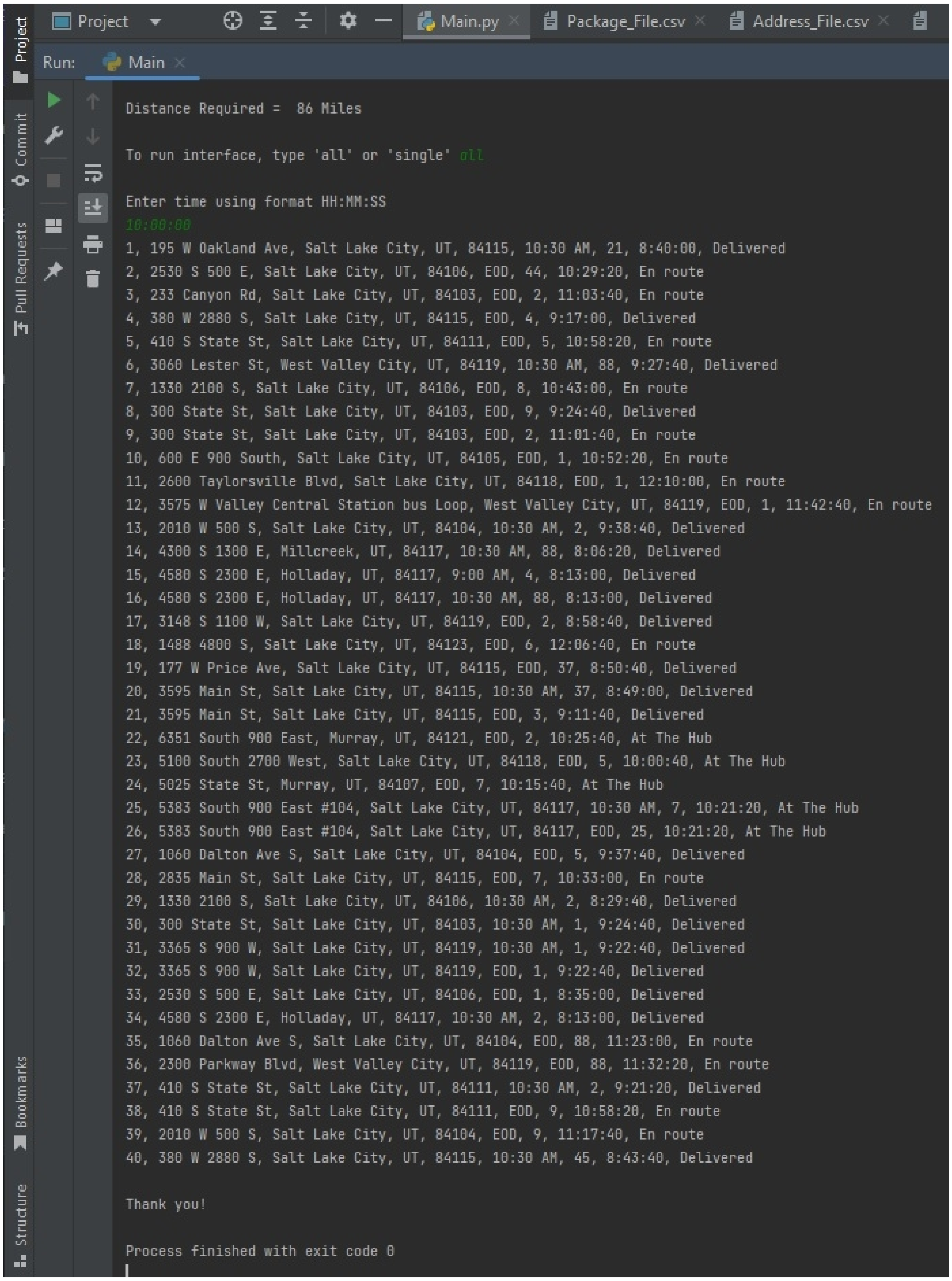
Screenshot of packages at 0900 (per requirements of 0835 thru 0925)



```
Distance Required =  86 Miles

To run interface, type 'all' or 'single' all

Enter time using format HH:MM:SS
09:00:00
1, 195 W Oakland Ave, Salt Lake City, UT, 84115, 10:30 AM, 21, 8:40:00, Delivered
2, 2530 S 500 E, Salt Lake City, UT, 84106, EOD, 44, 10:29:20, En route
3, 233 Canyon Rd, Salt Lake City, UT, 84103, EOD, 2, 11:03:40, En route
4, 380 W 2880 S, Salt Lake City, UT, 84115, EOD, 4, 9:17:00, En route
5, 410 S State St, Salt Lake City, UT, 84111, EOD, 5, 10:58:20, En route
6, 3060 Lester St, West Valley City, UT, 84119, 10:30 AM, 88, 9:27:40, En route
7, 1330 2100 S, Salt Lake City, UT, 84106, EOD, 8, 10:43:00, En route
8, 300 State St, Salt Lake City, UT, 84103, EOD, 9, 9:24:40, At The Hub
9, 300 State St, Salt Lake City, UT, 84103, EOD, 2, 11:01:40, En route
10, 600 E 900 South, Salt Lake City, UT, 84105, EOD, 1, 10:52:20, En route
11, 2600 Taylorsville Blvd, Salt Lake City, UT, 84118, EOD, 1, 12:10:00, En route
12, 3575 W Valley Central Station bus Loop, West Valley City, UT, 84119, EOD, 1, 11:42:40, En route
13, 2010 W 500 S, Salt Lake City, UT, 84104, 10:30 AM, 2, 9:38:40, At The Hub
14, 4300 S 1300 E, Millcreek, UT, 84117, 10:30 AM, 88, 8:06:20, Delivered
15, 4580 S 2300 E, Holladay, UT, 84117, 9:00 AM, 4, 8:13:00, Delivered
16, 4580 S 2300 E, Holladay, UT, 84117, 10:30 AM, 88, 8:13:00, Delivered
17, 3148 S 1100 W, Salt Lake City, UT, 84119, EOD, 2, 8:58:40, Delivered
18, 1488 4800 S, Salt Lake City, UT, 84123, EOD, 6, 12:06:40, En route
19, 177 W Price Ave, Salt Lake City, UT, 84115, EOD, 37, 8:50:40, Delivered
20, 3595 Main St, Salt Lake City, UT, 84115, 10:30 AM, 37, 8:49:00, Delivered
21, 3595 Main St, Salt Lake City, UT, 84115, EOD, 3, 9:11:40, En route
22, 6351 South 900 East, Murray, UT, 84121, EOD, 2, 10:25:40, En route
23, 5100 South 2700 West, Salt Lake City, UT, 84118, EOD, 5, 10:00:40, En route
24, 5025 State St, Murray, UT, 84107, EOD, 7, 10:15:40, En route
25, 5383 South 900 East #104, Salt Lake City, UT, 84117, 10:30 AM, 7, 10:21:20, En route
26, 5383 South 900 East #104, Salt Lake City, UT, 84117, EOD, 25, 10:21:20, En route
27, 1060 Dalton Ave S, Salt Lake City, UT, 84104, EOD, 5, 9:37:40, En route
28, 2835 Main St, Salt Lake City, UT, 84115, EOD, 7, 10:33:00, En route
29, 1330 2100 S, Salt Lake City, UT, 84106, 10:30 AM, 2, 8:29:40, Delivered
30, 300 State St, Salt Lake City, UT, 84103, 10:30 AM, 1, 9:24:40, At The Hub
31, 3365 S 900 W, Salt Lake City, UT, 84119, 10:30 AM, 1, 9:22:40, En route
32, 3365 S 900 W, Salt Lake City, UT, 84119, EOD, 1, 9:22:40, En route
33, 2530 S 500 E, Salt Lake City, UT, 84106, EOD, 1, 8:35:00, Delivered
34, 4580 S 2300 E, Holladay, UT, 84117, 10:30 AM, 2, 8:13:00, Delivered
35, 1060 Dalton Ave S, Salt Lake City, UT, 84104, EOD, 88, 11:23:00, En route
36, 2300 Parkway Blvd, West Valley City, UT, 84119, EOD, 88, 11:32:20, En route
37, 410 S State St, Salt Lake City, UT, 84111, 10:30 AM, 2, 9:21:20, At The Hub
38, 410 S State St, Salt Lake City, UT, 84111, EOD, 9, 10:58:20, En route
39, 2010 W 500 S, Salt Lake City, UT, 84104, EOD, 9, 11:17:40, En route
40, 380 W 2880 S, Salt Lake City, UT, 84115, 10:30 AM, 45, 8:43:40, Delivered

Thank you!

Process finished with exit code 0
```

G2. Second Status Check

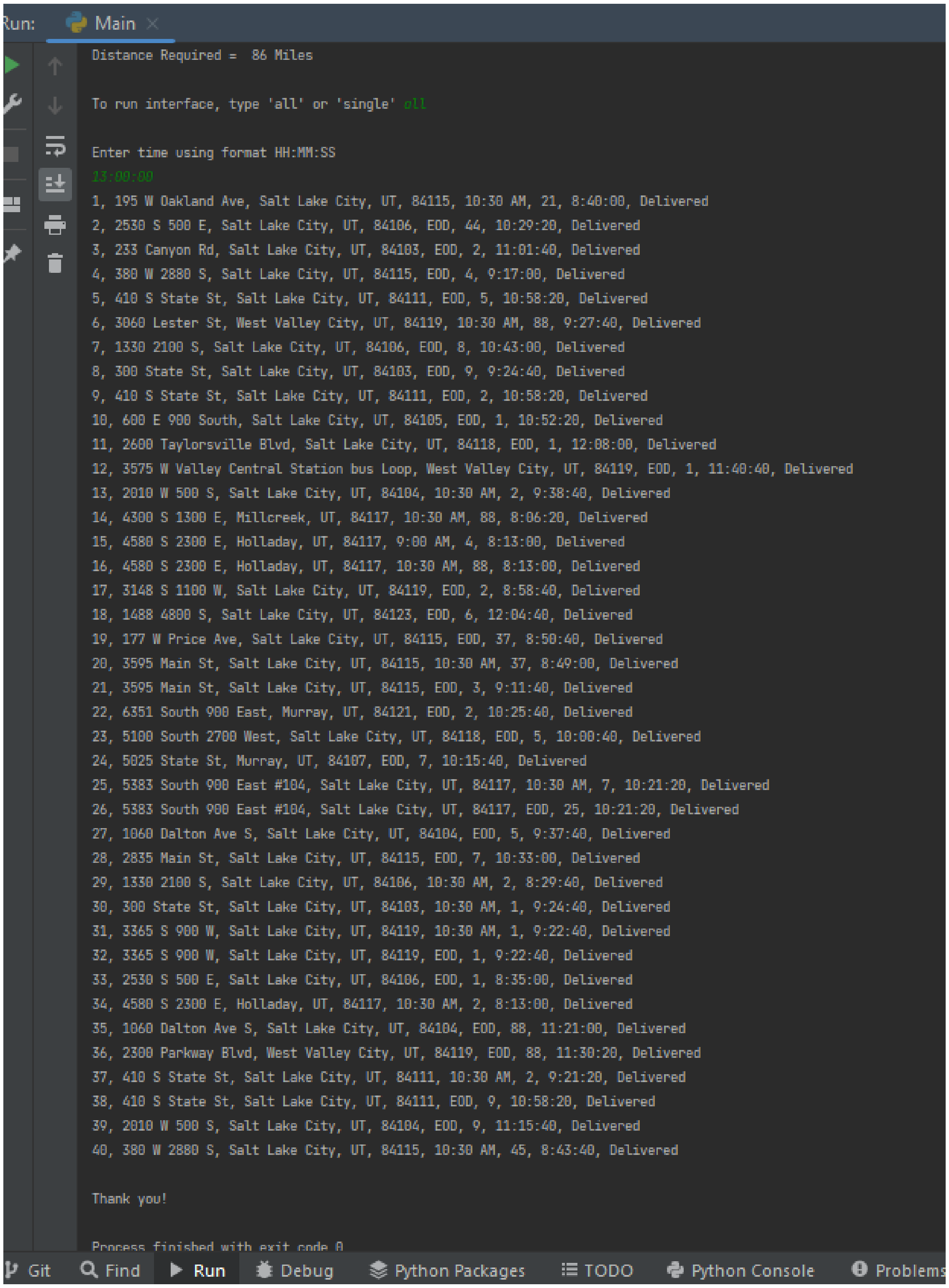
Screenshot of packages at 1000 (per requirements of 0935 thru 1025)



```
Distance Required =  86 Miles

To run interface, type 'all' or 'single' all

Enter time using format HH:MM:SS
10:00:00
1, 195 W Oakland Ave, Salt Lake City, UT, 84115, 10:30 AM, 21, 8:40:00, Delivered
2, 2530 S 500 E, Salt Lake City, UT, 84106, EOD, 44, 10:29:20, En route
3, 233 Canyon Rd, Salt Lake City, UT, 84103, EOD, 2, 11:03:40, En route
4, 380 W 2880 S, Salt Lake City, UT, 84115, EOD, 4, 9:17:00, Delivered
5, 410 S State St, Salt Lake City, UT, 84111, EOD, 5, 10:58:20, En route
6, 3060 Lester St, West Valley City, UT, 84119, 10:30 AM, 88, 9:27:40, Delivered
7, 1330 2100 S, Salt Lake City, UT, 84106, EOD, 8, 10:43:00, En route
8, 300 State St, Salt Lake City, UT, 84103, EOD, 9, 9:24:40, Delivered
9, 300 State St, Salt Lake City, UT, 84103, EOD, 2, 11:01:40, En route
10, 600 E 900 South, Salt Lake City, UT, 84105, EOD, 1, 10:52:20, En route
11, 2600 Taylorsville Blvd, Salt Lake City, UT, 84118, EOD, 1, 12:10:00, En route
12, 3575 W Valley Central Station bus Loop, West Valley City, UT, 84119, EOD, 1, 11:42:40, En route
13, 2010 W 500 S, Salt Lake City, UT, 84104, 10:30 AM, 2, 9:38:40, Delivered
14, 4300 S 1300 E, Millcreek, UT, 84117, 10:30 AM, 88, 8:06:20, Delivered
15, 4580 S 2300 E, Holladay, UT, 84117, 9:00 AM, 4, 8:13:00, Delivered
16, 4580 S 2300 E, Holladay, UT, 84117, 10:30 AM, 88, 8:13:00, Delivered
17, 3148 S 1100 W, Salt Lake City, UT, 84119, EOD, 2, 8:58:40, Delivered
18, 1488 4800 S, Salt Lake City, UT, 84123, EOD, 6, 12:06:40, En route
19, 177 W Price Ave, Salt Lake City, UT, 84115, EOD, 37, 8:50:40, Delivered
20, 3595 Main St, Salt Lake City, UT, 84115, 10:30 AM, 37, 8:49:00, Delivered
21, 3595 Main St, Salt Lake City, UT, 84115, EOD, 3, 9:11:40, Delivered
22, 6351 South 900 East, Murray, UT, 84121, EOD, 2, 10:25:40, At The Hub
23, 5100 South 2700 West, Salt Lake City, UT, 84118, EOD, 5, 10:00:40, At The Hub
24, 5025 State St, Murray, UT, 84107, EOD, 7, 10:15:40, At The Hub
25, 5383 South 900 East #104, Salt Lake City, UT, 84117, 10:30 AM, 7, 10:21:20, At The Hub
26, 5383 South 900 East #104, Salt Lake City, UT, 84117, EOD, 25, 10:21:20, At The Hub
27, 1060 Dalton Ave S, Salt Lake City, UT, 84104, EOD, 5, 9:37:40, Delivered
28, 2835 Main St, Salt Lake City, UT, 84115, EOD, 7, 10:33:00, En route
29, 1330 2100 S, Salt Lake City, UT, 84106, 10:30 AM, 2, 8:29:40, Delivered
30, 300 State St, Salt Lake City, UT, 84103, 10:30 AM, 1, 9:24:40, Delivered
31, 3365 S 900 W, Salt Lake City, UT, 84119, 10:30 AM, 1, 9:22:40, Delivered
32, 3365 S 900 W, Salt Lake City, UT, 84119, EOD, 1, 9:22:40, Delivered
33, 2530 S 500 E, Salt Lake City, UT, 84106, EOD, 1, 8:35:00, Delivered
34, 4580 S 2300 E, Holladay, UT, 84117, 10:30 AM, 2, 8:13:00, Delivered
35, 1060 Dalton Ave S, Salt Lake City, UT, 84104, EOD, 88, 11:23:00, En route
36, 2300 Parkway Blvd, West Valley City, UT, 84119, EOD, 88, 11:32:20, En route
37, 410 S State St, Salt Lake City, UT, 84111, 10:30 AM, 2, 9:21:20, Delivered
38, 410 S State St, Salt Lake City, UT, 84111, EOD, 9, 10:58:20, En route
39, 2010 W 500 S, Salt Lake City, UT, 84104, EOD, 9, 11:17:40, En route
40, 380 W 2880 S, Salt Lake City, UT, 84115, 10:30 AM, 45, 8:43:40, Delivered

Thank you!

Process finished with exit code 0
```

G3. Third Status Check

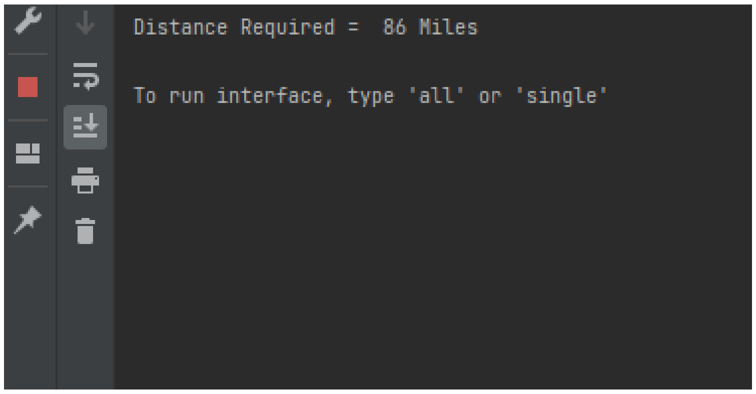
Screenshot of packages at 1300 (per requirements of 1203 thru 1312)



```
Distance Required =  86 Miles

To run interface, type 'all' or 'single' all

Enter time using format HH:MM:SS
13:00:00
1, 195 W Oakland Ave, Salt Lake City, UT, 84115, 10:30 AM, 21, 8:40:00, Delivered
2, 2530 S 500 E, Salt Lake City, UT, 84106, EOD, 44, 10:29:20, Delivered
3, 233 Canyon Rd, Salt Lake City, UT, 84103, EOD, 2, 11:01:40, Delivered
4, 380 W 2880 S, Salt Lake City, UT, 84115, EOD, 4, 9:17:00, Delivered
5, 410 S State St, Salt Lake City, UT, 84111, EOD, 5, 10:58:20, Delivered
6, 3060 Lester St, West Valley City, UT, 84119, 10:30 AM, 88, 9:27:40, Delivered
7, 1330 2100 S, Salt Lake City, UT, 84106, EOD, 8, 10:43:00, Delivered
8, 300 State St, Salt Lake City, UT, 84103, EOD, 9, 9:24:40, Delivered
9, 410 S State St, Salt Lake City, UT, 84111, EOD, 2, 10:58:20, Delivered
10, 600 E 900 South, Salt Lake City, UT, 84105, EOD, 1, 10:52:20, Delivered
11, 2600 Taylorsville Blvd, Salt Lake City, UT, 84118, EOD, 1, 12:08:00, Delivered
12, 3575 W Valley Central Station bus Loop, West Valley City, UT, 84119, EOD, 1, 11:40:40, Delivered
13, 2010 W 500 S, Salt Lake City, UT, 84104, 10:30 AM, 2, 9:38:40, Delivered
14, 4300 S 1300 E, Millcreek, UT, 84117, 10:30 AM, 88, 8:06:20, Delivered
15, 4580 S 2300 E, Holladay, UT, 84117, 9:00 AM, 4, 8:13:00, Delivered
16, 4580 S 2300 E, Holladay, UT, 84117, 10:30 AM, 88, 8:13:00, Delivered
17, 3148 S 1100 W, Salt Lake City, UT, 84119, EOD, 2, 8:58:40, Delivered
18, 1488 4800 S, Salt Lake City, UT, 84123, EOD, 6, 12:04:40, Delivered
19, 177 W Price Ave, Salt Lake City, UT, 84115, EOD, 37, 8:50:40, Delivered
20, 3595 Main St, Salt Lake City, UT, 84115, 10:30 AM, 37, 8:49:00, Delivered
21, 3595 Main St, Salt Lake City, UT, 84115, EOD, 3, 9:11:40, Delivered
22, 6351 South 900 East, Murray, UT, 84121, EOD, 2, 10:25:40, Delivered
23, 5100 South 2700 West, Salt Lake City, UT, 84118, EOD, 5, 10:00:40, Delivered
24, 5025 State St, Murray, UT, 84107, EOD, 7, 10:15:40, Delivered
25, 5383 South 900 East #104, Salt Lake City, UT, 84117, 10:30 AM, 7, 10:21:20, Delivered
26, 5383 South 900 East #104, Salt Lake City, UT, 84117, EOD, 25, 10:21:20, Delivered
27, 1060 Dalton Ave S, Salt Lake City, UT, 84104, EOD, 5, 9:37:40, Delivered
28, 2835 Main St, Salt Lake City, UT, 84115, EOD, 7, 10:33:00, Delivered
29, 1330 2100 S, Salt Lake City, UT, 84106, 10:30 AM, 2, 8:29:40, Delivered
30, 300 State St, Salt Lake City, UT, 84103, 10:30 AM, 1, 9:24:40, Delivered
31, 3365 S 900 W, Salt Lake City, UT, 84119, 10:30 AM, 1, 9:22:40, Delivered
32, 3365 S 900 W, Salt Lake City, UT, 84119, EOD, 1, 9:22:40, Delivered
33, 2530 S 500 E, Salt Lake City, UT, 84106, EOD, 1, 8:35:00, Delivered
34, 4580 S 2300 E, Holladay, UT, 84117, 10:30 AM, 2, 8:13:00, Delivered
35, 1060 Dalton Ave S, Salt Lake City, UT, 84104, EOD, 88, 11:21:00, Delivered
36, 2300 Parkway Blvd, West Valley City, UT, 84119, EOD, 88, 11:30:20, Delivered
37, 410 S State St, Salt Lake City, UT, 84111, 10:30 AM, 2, 9:21:20, Delivered
38, 410 S State St, Salt Lake City, UT, 84111, EOD, 9, 10:58:20, Delivered
39, 2010 W 500 S, Salt Lake City, UT, 84104, EOD, 9, 11:15:40, Delivered
40, 380 W 2880 S, Salt Lake City, UT, 84115, 10:30 AM, 45, 8:43:40, Delivered

Thank you!

Process finished with exit code 0
```

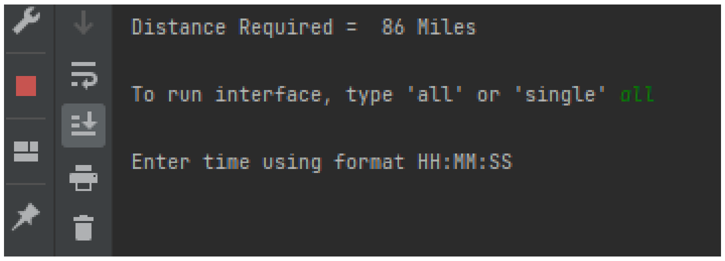
H. Screenshots of Code Execution

Once the program is started, this information is given.



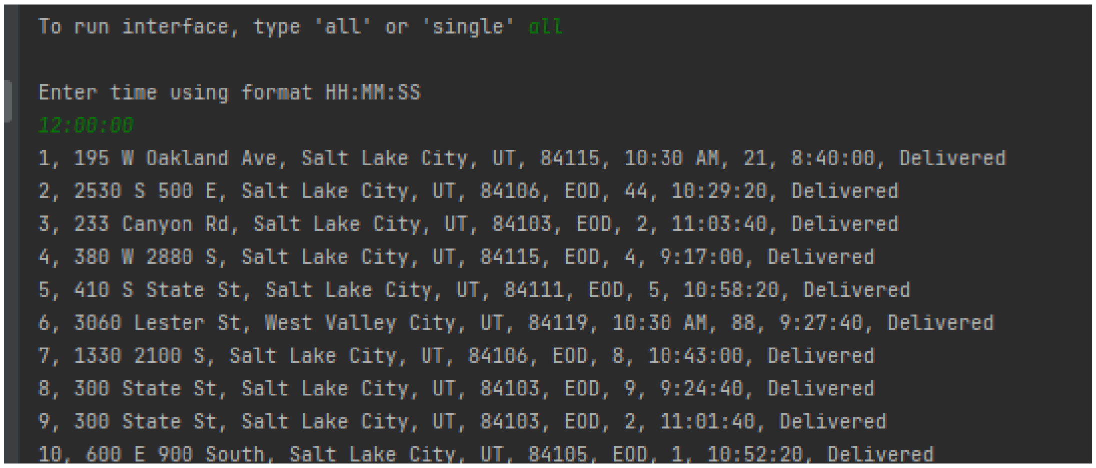
```
Distance Required =  86 Miles

To run interface, type 'all' or 'single'
```

If “all” is entered we are prompted for a time in the requested format.



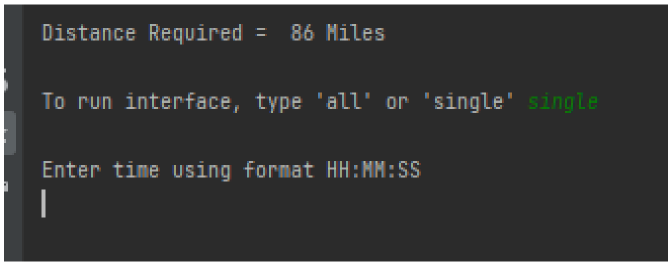
```
Distance Required =  86 Miles

To run interface, type 'all' or 'single' all

Enter time using format HH:MM:SS
```

Once a time is entered, all of the packages are queried with a delivery time and updated status.



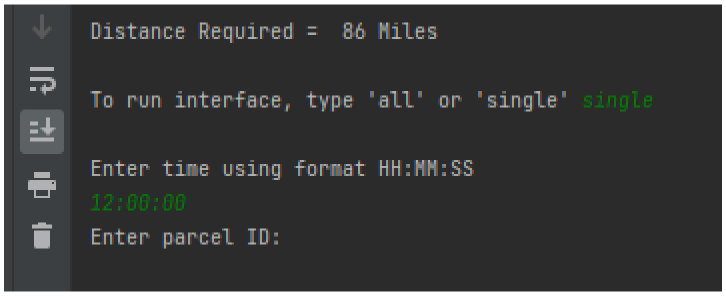
```
To run interface, type 'all' or 'single' all

Enter time using format HH:MM:SS
12:00:00
1, 195 W Oakland Ave, Salt Lake City, UT, 84115, 10:30 AM, 21, 8:40:00, Delivered
2, 2530 S 500 E, Salt Lake City, UT, 84106, EOD, 44, 10:29:20, Delivered
3, 233 Canyon Rd, Salt Lake City, UT, 84103, EOD, 2, 11:03:40, Delivered
4, 380 W 2880 S, Salt Lake City, UT, 84115, EOD, 4, 9:17:00, Delivered
5, 410 S State St, Salt Lake City, UT, 84111, EOD, 5, 10:58:20, Delivered
6, 3060 Lester St, West Valley City, UT, 84119, 10:30 AM, 88, 9:27:40, Delivered
7, 1330 2100 S, Salt Lake City, UT, 84106, EOD, 8, 10:43:00, Delivered
8, 300 State St, Salt Lake City, UT, 84103, EOD, 9, 9:24:40, Delivered
9, 300 State St, Salt Lake City, UT, 84103, EOD, 2, 11:01:40, Delivered
10, 600 E 900 South, Salt Lake City, UT, 84105, EOD, 1, 10:52:20, Delivered
```

Now the program will be used for single package query. The program is started and the user types in the requested “single” text input.



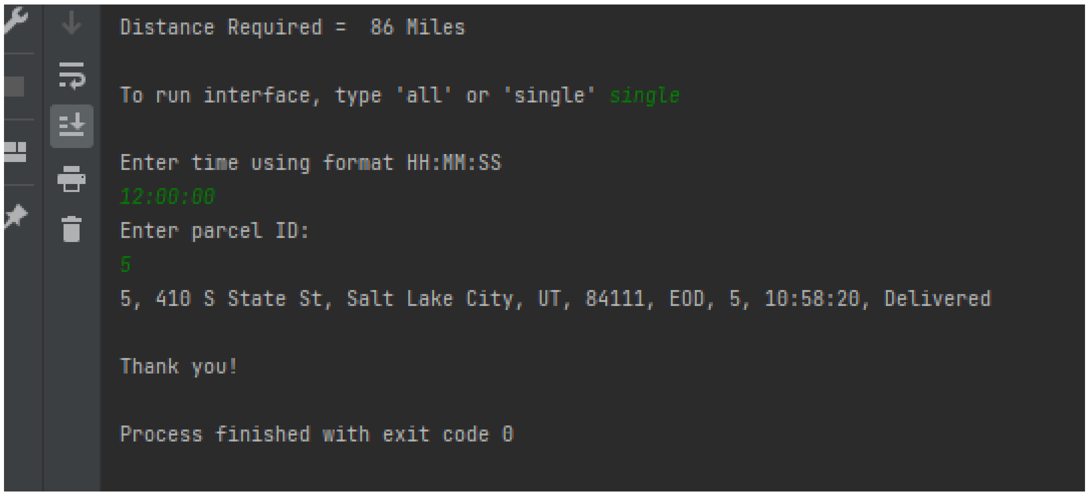
```
Distance Required =  86 Miles

To run interface, type 'all' or 'single' single

Enter time using format HH:MM:SS
|
```

The user is then prompted to enter a time in the requested format, then prompted for an individual parcel ID



```
Distance Required =  86 Miles

To run interface, type 'all' or 'single' single

Enter time using format HH:MM:SS
12:00:00
Enter parcel ID:
```

In this example, the user entered parcel id “5”. The following data is returned as requested, and the program politely thanks the user for its interaction.



```
Distance Required =  86 Miles

To run interface, type 'all' or 'single' single

Enter time using format HH:MM:SS
12:00:00
Enter parcel ID:
5
5, 410 S State St, Salt Lake City, UT, 84111, EOD, 5, 10:58:20, Delivered

Thank you!

Process finished with exit code 0
```

I1. Strengths of Chosen Algorithm

Dijkstra's algorithm provides two very important strengths that apply to the given scenario.

1. The task at hand is to find the shortest path given a variety of parcels. This is a great use of Dijkstra's algorithm as it is historically used for finding the optimal path between any two vertices on a graph.
2. The future scaling of this programming should work well. Dijkstra's algorithmic run time complexity is O(E log V). Given the "finite" scale of a trucks holding capacity, this algorithm will work well for future growth and flexibility.

I2. Verification of Algorithm

The algorithm meets all requirements verified in the screenshots listed above. Please refer to G1 thru G3

1. Total miles are less than 140
2. All packages were delivered on time.

I3. Other possible Algorithms

Listed below are two other algorithms that would work well for the given scenario.

1. Bellman-Ford algorithm

Meets the requirements of delivering parcels below 140 miles within the time constraints

2. Prim Algorithm

Also very capable of delivering parcels below 140 miles within the time requirements.

I3A. Algorithm Differences

1. Bellman-Ford algorithm allows graphs with negative values to be used. This may not seem practical or useful, however it would add potential capabilities from the creation of the program, which is always something to consider while planning the design of a program. Prim and Dijkstra do not offer this capability, that is the comparison.
2. Prim differs from Dijkstra in its graph traversal. Prim chooses the "cheapest" path from the start, while Dijkstra considers "future paths" and traverses backwards to come to a optimal solution. Both are useful for the scenario, however, it must be mentioned how they differ.

J. Different Approach

One aspect that would be done differently if the project was done again would be to apply a variety of algorithms within the python file structure. Perhaps not too many, however, it would be nice to have algorithms written in separate python files ready to apply quickly and easily in the "Main.py" class. I actually will be doing this in the future to continue my learning and understanding of the python language and a deeper education into algorithms. This class was very fun and engaging, a very different take on software development from past WGU courses.

K1. Verification of Data Structure

The data structure known as a "hash table" is used effectively and meets all the requirements. Please refer to G1 thru G3, which includes screenshots that illustrate that the requirements are met.

1. Total miles to all trucks
2. Combined delivery distance below 140 miles
3. All packages delivered on time.

The hash table with look-up function is present in the "Hash.py" file. This data structure works without errors and provides accurate and efficient reporting.

K1A. Efficiency

Given the scenario of this project, there are a total of 40 parcels. The data structure time complexity for this program is O(n). As more parcels are added to the scenario, the time to complete the look-up function will scale in a linear fashion.

K1B. Overhead

As the hash table size increases, due to added parcels, additional memory would be required. Thanks to modern computing power, and also the fact that this program was designed to run in a local environment, the program should scale well in regards to space complexity.

K1C. Implications

Given the current number of vehicles in the given scenario, the runtime complexity sits at O(n^3). Changes to the number of trucks or the number of cities will compound and could lead to slower performance. As taught in past WGU courses, avoiding nested loops in the codebase is a way to help optimize the program from an algorithmic sense. The implications of growth in the program is important to consider and is why researching data structures and algorithm combinations are critical to software development.

K2. Other Data Structures

Two data structures that could be used that differ from part D which could meet the requirements in the scenario could include.

1. Binary Search Tree
2. Stack

K2a. Data Structure Differences

1. Binary Search Trees have a runtime complexity of O(log n). The traversal starts at a "root" node and continues in a binary fashion into further "subtrees". The efficiency of this data structure is partially due to the elements in a "left" subtree are smaller then the elements in the "right" subtree. This is a completely different approach from a hashmap structure, and should be considered for future efficiency.
2. Stacks are data structures which allow for LIFO or FILO. Last In First Out and First In Last Out. The terms used for added and removing data is commonly referred to as "push" and "pop". A stack is also different from a hashmap as it is a single "vertical" or "horizontal" stack of data compared to a hashmap which is in a table format.

L. Sources - Works Cited

Binary search tree: Set 1 (search and insertion). (2023, January 02).

Retrieved January 18, 2023, from

https://www.geeksforgeeks.org/binary-search-tree-set-1-search-and-insertion/

Stack Data Structure. (n.d.).

Retrieved January 18, 2023, from

https://www.geeksforgeeks.org/stack-data-structure/

Hash map in python. (2022, May 09).

Retrieved January 18, 2023, from

https://www.geeksforgeeks.org/hash-map-in-python/

M. Professional Communication

Total time to complete the project was 80 hours. Attention to detail, organization, and focus on the main requirements were always monitored. The terminology is researched and verified with a variety of sources listed above. This document was scanned for spelling and grammar to assist in accurate interpretations of the topic at hand.

Thank you very much for your time, it is greatly appreciated.