|  |
| --- |
| Western Governors University |
| TSP Optimization Algorithm Analysis |
| Package Delivery Branch and Bound Algorithm Implementation |

|  |
| --- |
| Misty Hurley  5-25-2021 |

Contents

[A. Algorithm Selection 3](#_Toc72876781)

[B. Algorithm Overview 3](#_Toc72876782)

[1. Algorithm Logic 4](#_Toc72876783)

[2. Programming Environment 6](#_Toc72876784)

[3. Complexity Analysis 6](#_Toc72876785)

[4. Scalability 6](#_Toc72876786)

[5. Efficiency and maintainability 7](#_Toc72876787)

[6. Strengths and Weaknesses of the Self-Adjusting Hash Table 7](#_Toc72876788)

[C. Write an original program 7](#_Toc72876789)

[D. Self-adjusting Data Structure Explanation 7](#_Toc72876790)

[G. Screenshots of User Interface with Status Lookups 8](#_Toc72876791)

[Show the status of *all* packages at a time between 8:35 a.m. and 9:25 a.m. (8:45 AM) 9](#_Toc72876792)

[Show the status of *all* packages at a time between 9:35 a.m. and 10:25 a.m. (9:50 AM) 10](#_Toc72876793)

[Show the status of *all* packages at a time between 12:03 p.m. and 1:12 p.m. (1:11 PM) 11](#_Toc72876794)

[H. Screenshot successful completion of the code 12](#_Toc72876795)

[I. Core Algorithm Justification 13](#_Toc72876796)

[1. Describe at least two strengths of the algorithm used in the solution. 13](#_Toc72876797)

[2. Verify that the algorithm used in the solution meets *all* requirements in the scenario. 13](#_Toc72876798)

[3. Other possible algorithms that may be used to solve this problem: 13](#_Toc72876799)

[J.  Describe what you would do differently, other than the two algorithms identified in I3, if you did this project again. 14](#_Toc72876800)

[K.  Justify the data structure identified in part D 14](#_Toc72876801)

[1.  Verify that the data structure used in the solution meets *all* requirements in the scenario. 14](#_Toc72876802)

[a.  Explain how the time needed to complete the look-up function is affected by changes in the number of packages to be delivered. 14](#_Toc72876803)

[b.  Explain how the data structure space usage is affected by changes in the number of packages to be delivered. 14](#_Toc72876804)

[c.  Describe how changes to the number of trucks or the number of cities would affect the look-up time and the space usage of the data structure. 15](#_Toc72876805)

[2.  Identify two other data structures that could meet the same requirements in the scenario. 15](#_Toc72876806)

[Describe how *each* data structure identified in part K2 is different from the data structure used in the solution. 15](#_Toc72876807)

[Misc. Notes 15](#_Toc72876808)

[References 16](#_Toc72876809)

# A. Algorithm Selection

This project aims to solve the following traveling salesman problem (TSP): Given a table of packages to be delivered, including addresses and various restrictions, and a table of distances to other addresses, find a set of routes optimized for distance and time that delivers each package by the assigned deadlines, while meeting all other restrictions.

The program utilizes a branch and bound algorithm to attempt to find the shortest path between addresses. Branch and bound is an algorithm design paradigm used for efficient tree traversal that can be used to solve combinatorial optimization problems such as the TSP. The *branch* portion of the algorithm successively partitions the set of all possible solutions into disjoint subsets, while the *bound* portion refers to discarding proposed solutions that do not fall within the established upper and lower limits (Stephens, 2013).

This implementation reorders the provided package list to optimize total distance without violating any of the required constraints (deadlines, truck requirements, and delays). The primary benefit of this algorithm is that it finds a good solution fast. Theoretically, the complexity of this solution is O(n2). However, the initial set being optimized by nearest neighbor and the heuristic culling should reject enough nodes to result in an average case closer to O(n\*logn).

# B. Algorithm Overview

**General Program Logic**

HashTable package\_table = key(package\_id):value(package)

packages\_left = clone(package\_table)

while packages\_left > 0

* pick truck with earliest availability time
* create route with picked truck\_id and start\_time
* if fewer than a max load of packages is left and any of them require a specific truck, wait to use required truck
  + else use earliest available truck
* attempt to intelligently fill route with packages with regards to constraints and ease; roughly nearest neighbor
  + prefer meeting deadlines first
  + prefer packages that belong to groups
  + try to use all package in a group if adding a single package from a group
  + prefer the closest package to the last package loaded (or the hub if there are none)
  + try to include packages that match an address with any current package
* Call branch and bound to find distance/time optimized route for the packages selected, trunk to leaf (depth-wise)
  + prune any branch out of bounds or violating rules:
    - deadline must be met
    - has correct truck\_id or no truck\_id
    - delivery group must fit together in solution
    - delay\_time <= route start\_time
    - package count <= MAX\_TRUCK\_PACKAGES
  + heuristically determine to be likely better than best solution so far
  + track best solution, reject branches that are impossible or not within 95% of expected target at current depth

## Algorithm Logic

**Branch and Bound Algorithm:**

*See route.py methods:* find\_optimal\_path, tsp\_traverse *for additional comments*

**def find\_optimal\_path():**

**global best\_solution, best\_cost, current\_solution, current\_cost  
# for building and checking a solution while traversing the tree  
current\_solution = []  
current\_solution\_table = HashTable(2 \* Util.MAX\_TRUCK\_PACKAGES)  
current\_cost = 0  
# for tracking the best solution so far  
best\_solution = []  
best\_cost = float("inf")**

**def tsp\_traverse():**

**global best\_solution, best\_cost, current\_solution, current\_cost**

**packages\_to\_route = length of package\_ids**

**array package\_candidates = []**

**for package\_id in package\_ids:**

**if current\_solution\_table is not None:**

**continue**

**# get package object**

**# package\_table = HashTable built from package.csv reader**

**package = package\_table.get(package\_id)**

**if current\_solution is empty:**

**last\_address = HUB\_ADDRESS**

**else:**

**last\_address = address of last package in solution so far**

**# distance\_table = HashTable built from distance.csv reader**

**package\_cost = distance\_table.get(last\_address, package\_address)**

**potential\_cost = current\_cost + package\_cost**

**if potential\_cost > best\_cost \* ((current length + 1)/ packages \* .95)**

**continue # not a good candidate for significant gains**

**if start\_time + (potential\_current\_cost / TRUCK\_MPH) > deadline:**

**continue # can’t meet deadline**

**package\_candidate\_cost\_table(package\_id, package\_cost)**

**if length of package\_candidates == 0:**

**return # reject this branch if no valid candidates**

**# grab top performing candidates**

**sort package\_candidates by package\_candidate\_cost\_table**

**for index, package\_candidate in enumerate(package\_candidates):**

**append to top\_package\_candidates**

**if index > length(package\_candidates) / 4:**

**break**

**for package\_id in top\_package\_candidates**

**append current\_solution**

**add package\_id to current\_solution\_table**

**previous\_cost = current\_cost**

**current\_cost += package\_id\_current\_cost**

**Curr\_length = length(current\_solution)**

**if curr\_length == packages\_to\_route:**

**# leaf; is this a new best solution?**

**current\_cost += current\_solution**

**if current\_cost < best\_cost**

**best\_solution = current\_solutiion**

**best\_cost = current\_cost**

**# pop address back of the stack to traverse parents/siblings**

**pop current\_solution**

**current\_cost = previous\_cost**

**else:**

**# branch; continue traversing through candidate**

**tsp\_traverse() # traverse deeper into tree**

**# pop address back of the stack to traverse parents/siblings**

**pop current\_solution**

**current\_cost = previous\_cost**

**tsp\_traverse() # traverse from root**

**package\_ids = best\_solution**

## Programming Environment

This program was written and developed using:

* Python 3.9.5
* PyCharm Community Edition 2021.1.1
* Microsoft Windows Version 10.0.19042.985
* Microsoft Surface Pro 6

## Complexity Analysis

*Note: See code for expanded in-text discussion of Big O analysis*

Branch and bound algorithm results in a theoretical complexity O(n) = n \* n. Though realistically, heuristic culling should reject enough nodes to be closer to roughly O(n) = n \* log(n). Additionally, the starting order is nearest neighbor, which serves to eliminate dead-end branches sooner than starting with an arbitrary order, thus avoiding unnecessary traversal.

## Scalability

Scalability has been considered throughout program design. The algorithm can adapt to a growing number of packages, provided the distance table is updated with any new address information. Nested hash tables are used to provide quick lookups, suitable for a growing dataset.

Variables such as the number of drivers and trucks can be easily updated. However, the current test data results in a solution that finishes at 12:54 PM, which is approximately 42% of given workday hours. Thus, it is reasonable to assume that the list of packages can approximately double in size before more drivers or trucks are necessary.

Exponential growth in the number of packages may result in significant slowdown in the program, at which point a more optimized solution involving a database management solution would be a worthwhile investment. Additionally, a substantial number of new addresses would result in an unavoidable exponential growth in the distance lookup table but would not affect lookup times.

## Efficiency and maintainability

As the TSP is classified as an NP-hard problem, a solution in polynomial time meets the criteria for efficiency. The dataset (including trucks, packages, and addresses) can continue to grow without significant slowdown. Nearly all lookups are done using hash tables, which are dynamically scaled to handle input data.

## Strengths and Weaknesses of the Self-Adjusting Hash Table

Most lookups are implemented via hash tables, which results in O(1) constant time for searching, creating, and deleting stored data, so long as the number of slots assigned is reasonable for the expected dataset (it is recommended to use length \* 2 to increase entropy and avoid collisions). Hash tables can use a variety of data types for values.

Though the primary benefit of a hash table is fast lookups, this benefit is only unidirectional, as searching matching keys from values would require looping through the whole dataset. Additionally, hash maps can become memory intensive at extremely large scale. The program can become inefficient if there are many collisions that would push the search complexity more toward O(n) linear time due to the use of singly-linked lists to tie-break. Fortunately, the chance of a significant number of collisions should be minimized by having a robust hashing algorithm that results in uniform slot distribution and by implementing more slots than the minimum theoretically required to store the full dataset (more than one slot per entry).

# C. Write an original program

To run this program, execute main.py and follow instructions in the command-line interface.

# D. Self-adjusting Data Structure Explanation

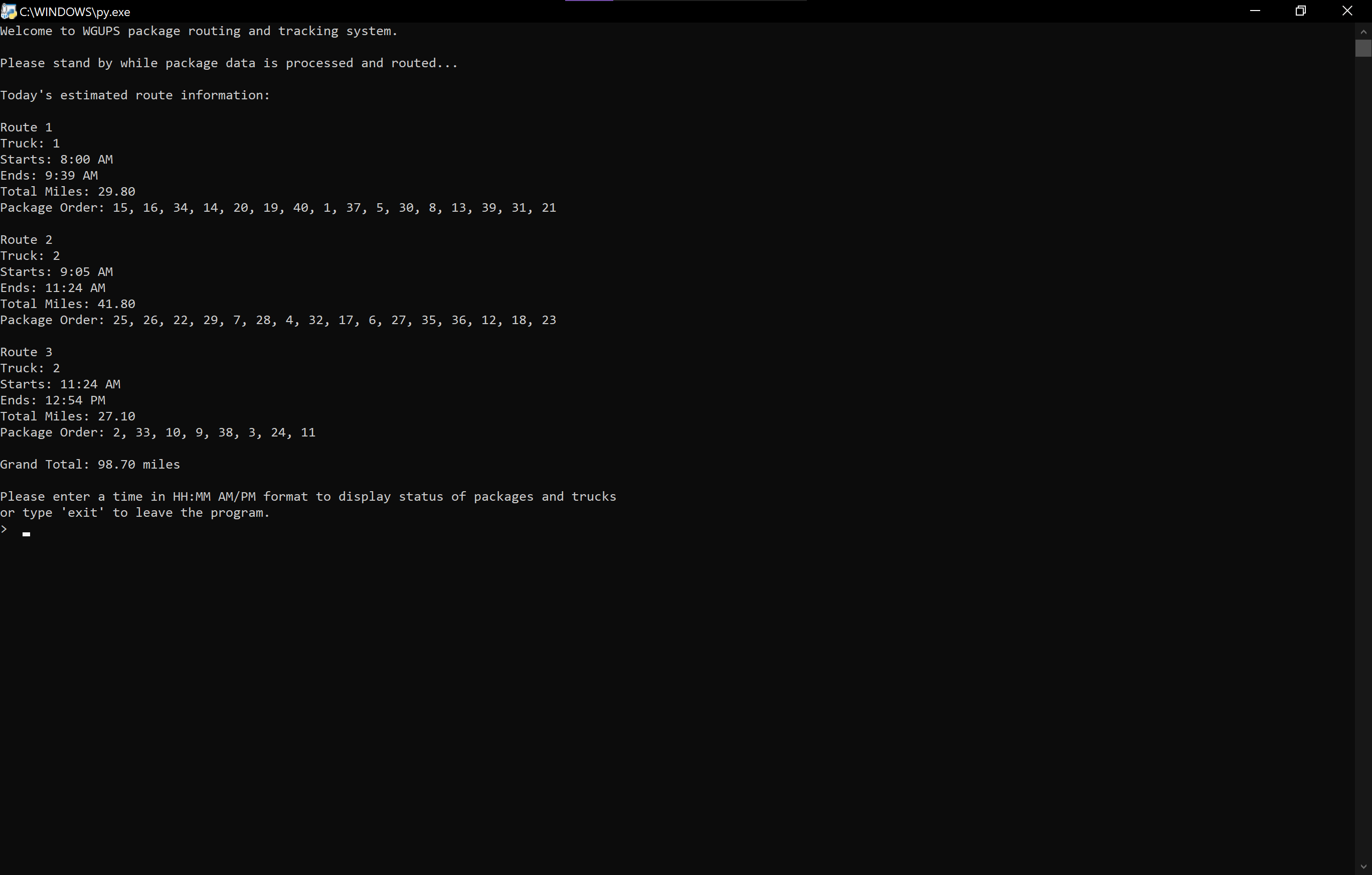
The hash table implementation used for this program is expandable as it features a user definable number of slots. Collision resolution is implemented via singly linked lists, and the number of slots is deterministically calculated from a fixed-size dataset at runtime, resulting in a self-adjusting data structure. However, the self-adjusting nature can result in significant memory usage and slowdown if an appropriate capacity is not chosen upon creation.

The hash table implementation was selected due to the given assumptions that each provided table includes unique identifiers, i.e. package id for the package table and address for the distance table. Additionally, a nested hash table solution by address pair is used to quickly look up distances between addresses.

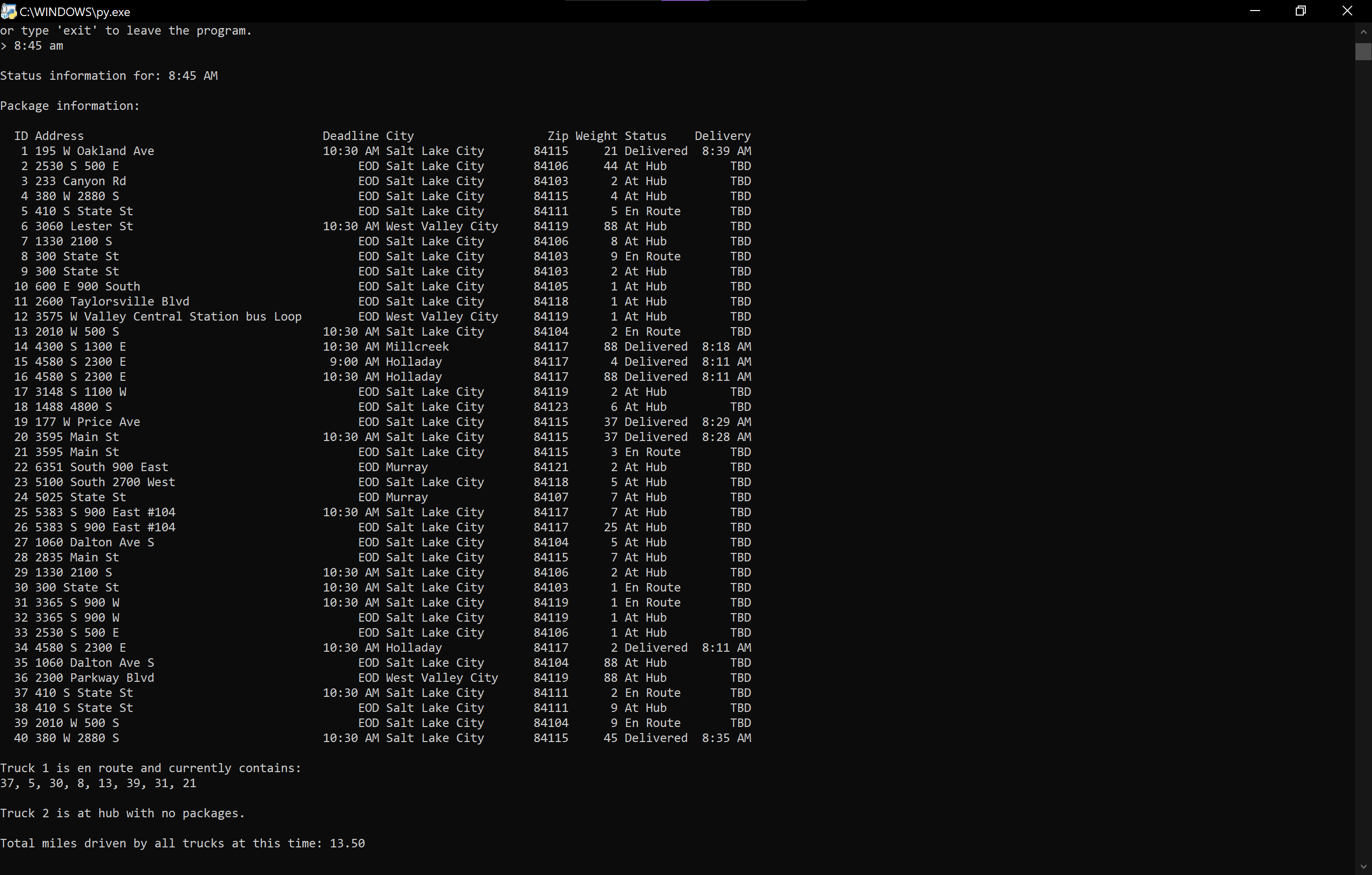
*Note: See hashtable.py in code files for more information on the HashTable class.*

# G. Screenshots of User Interface with Status Lookups

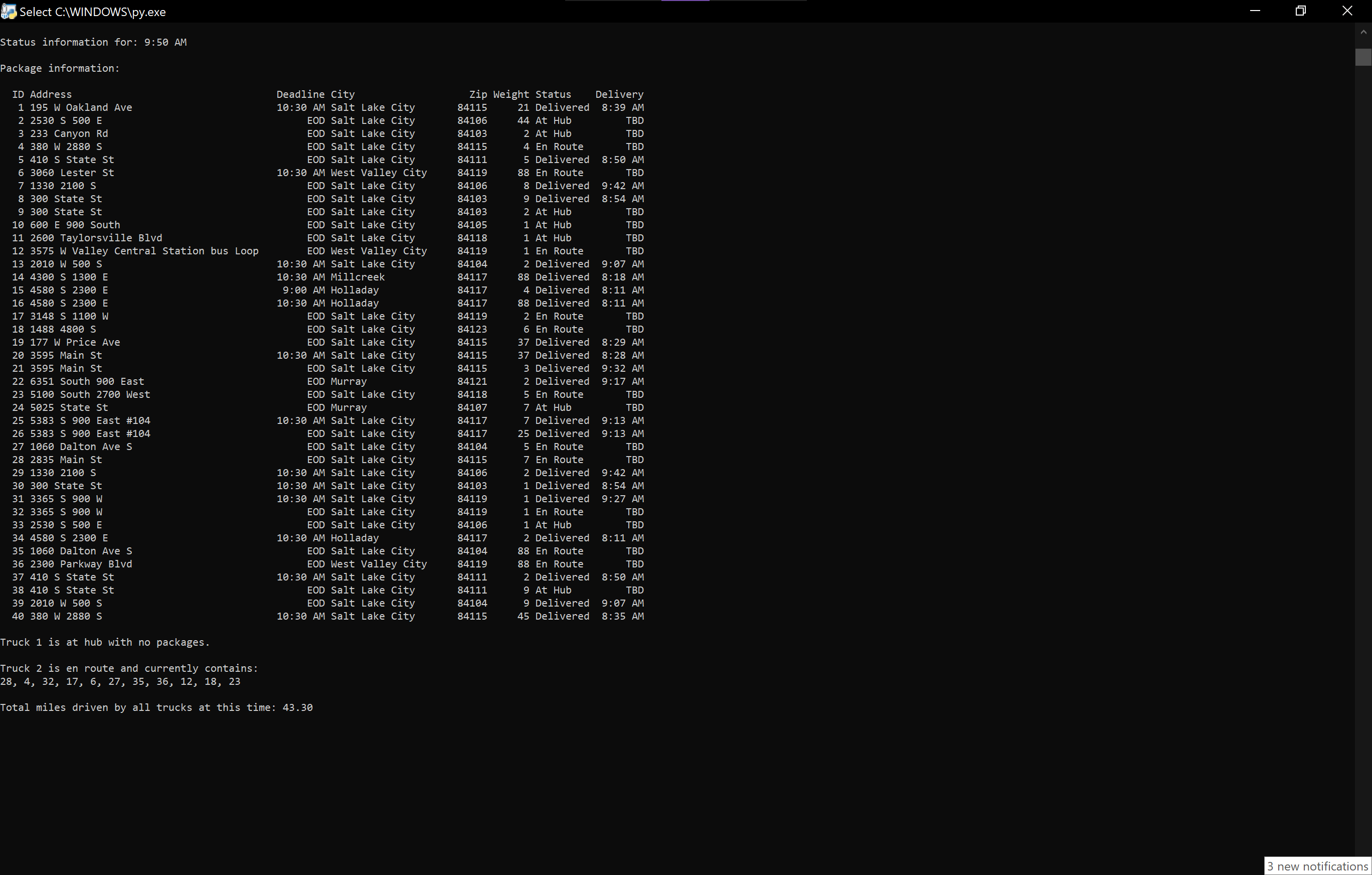
*Note: Screenshot files may also be viewed from the Screenshots folder*



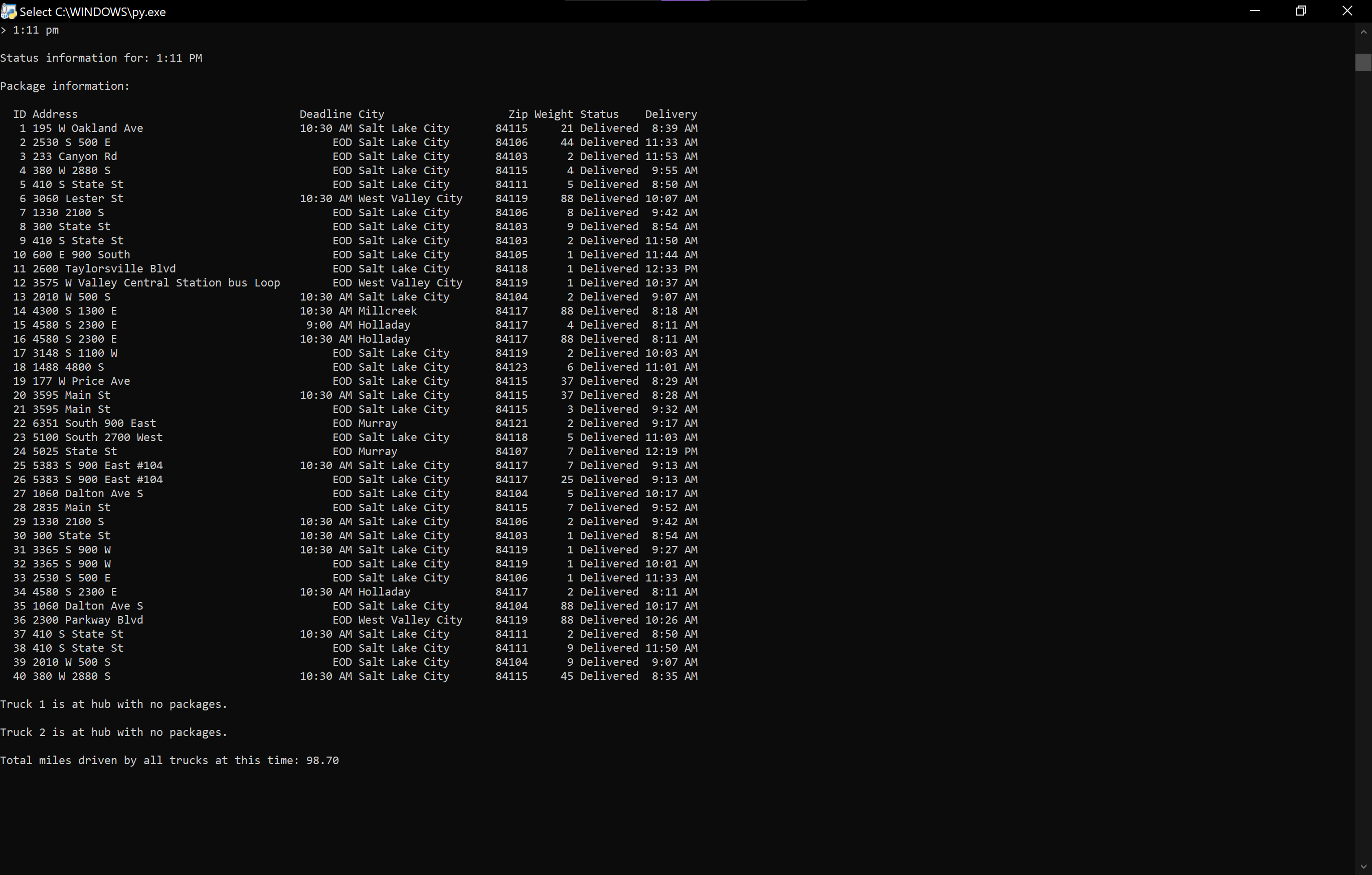
## Show the status of *all* packages at a time between 8:35 a.m. and 9:25 a.m. (8:45 AM)



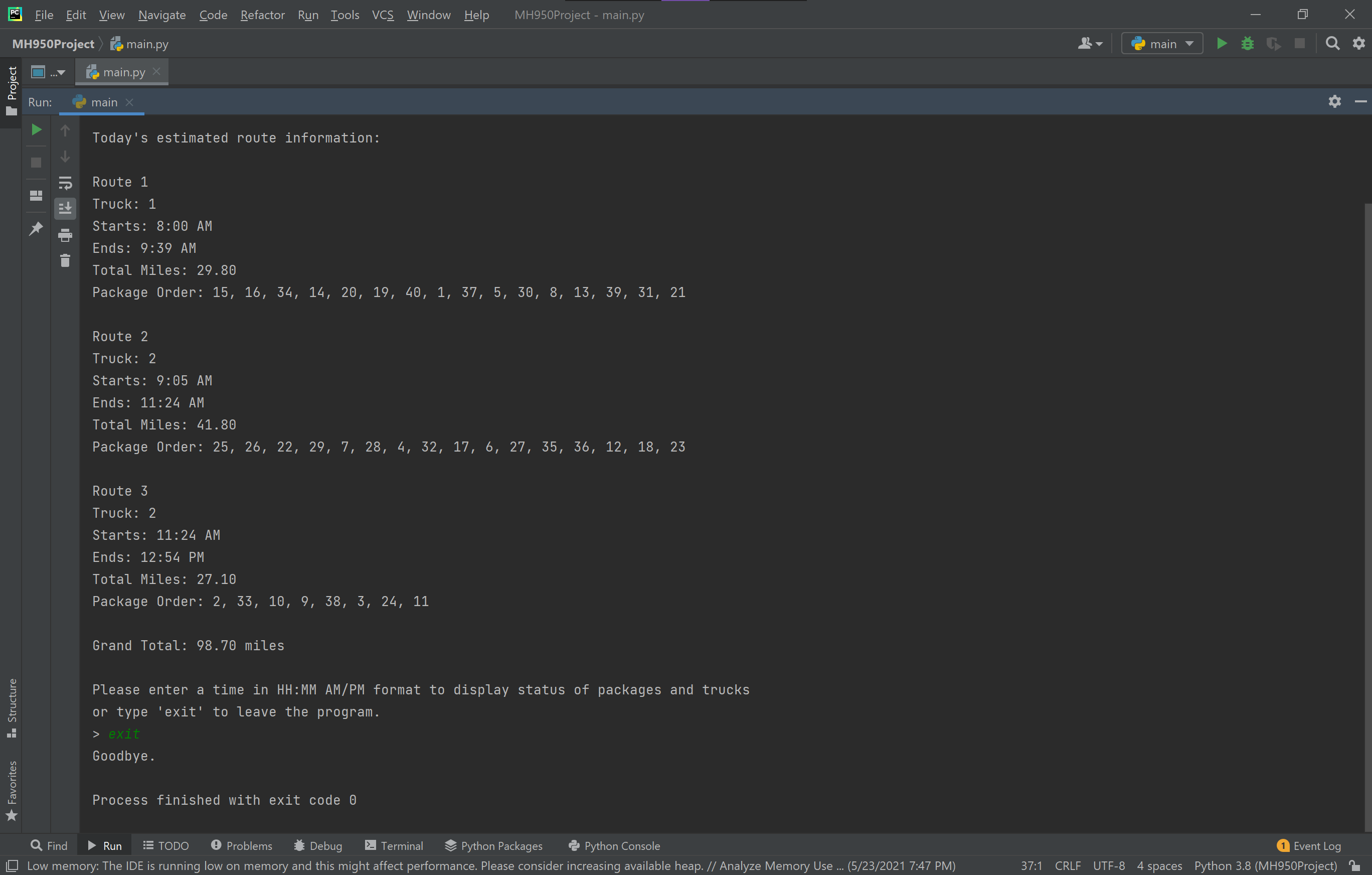
## Show the status of *all* packages at a time between 9:35 a.m. and 10:25 a.m. (9:50 AM)



## Show the status of *all* packages at a time between 12:03 p.m. and 1:12 p.m. (1:11 PM)



# H. Screenshot successful completion of the code



# I. Core Algorithm Justification

1. Describe at least two strengths of the algorithm used in the solution.

This implementation reorders the provided package list to optimize total distance without violating any of the required constraints (deadlines, truck requirements, and delays) and ultimately chooses the naively shortest path from each node (nearest neighbor). This is followed by optimizing the order of the packages picked for the route via branch and bound. The primary benefit of this algorithm is that it finds a good solution relatively quickly given the quality of the result it achieves. Theoretically, the complexity of this solution is O(n2). However, the heuristic culling should reject enough nodes to result in an average case of O(n\*logn). The process is also deterministic and provides a consistent result on every execution. The solution is consistently close to optimal as a result.

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

The solution meets all requirements outlined in the scenario, including:

* Total distance traveled including returning to the hub at the end of the day is under 140 miles (total: 98.70 miles)
* All packages are delivered before the associated delivery deadlines
* Delayed packages do not exit the hub until after the associated delays
  + Truck 2, Route 2 & 3
* Packages with the special note type “Must be delivered with…” are associated with a group ID that ensures these packages are delivered in the same route
  + 13, 14, 15, 16, 19, 20 are delivered on Truck 1, Route 1
* Packages that must be delivered on Truck 2 are delivered on Truck 2
  + 36, Route 2
  + 3 & 38, Route 3
* Incorrect address is displayed in the status report until it is fixed at 10:20 AM

## Other possible algorithms that may be used to solve this problem:

* Nearest Neighbor
* Dijkstra's shortest path

The nearest neighbor algorithm is a solution in which all vertices are initialized as unvisited, an arbitrary vertex is selected (or in this use case, the chosen vertex would be guided to meet constraints), then the shortest edge from the current to the next vertex is selected, and the process repeats until all vertices are visited (Selkow et al, 2008). Advantages include that the algorithm is easy to implement and performs at O(N2) complexity. A disadvantage is that an arbitrarily selected vertex will sometimes result in a distance that is much longer than optimal. Although the implemented branch and bound algorithm utilizes a shortest path approach that is similar to the nearest neighbor algorithm, additional elimination methods are used to narrow down potential candidates in order to encourage a more optimal path.

Dijkstra’s shortest path algorithm marks all nodes unvisited and creates a set of unvisited nodes, assigns an initial node to 0 and infinity to other nodes, calculates the distance of all neighbor nodes, selects the node with the shortest path as current, and repeats the visiting process with all unvisited neighbor nodes until the all vertices have been visited. This algorithm is a breadth-first search in which no node is traversed more than once, ultimately ending at the shortest unterminated path it traversed. The branch and bound algorithm implemented would traverse vertex orders that Dijkstra might not, thus possibly reaching a more optimal path, albeit with an additional time trade-off.

# J.  Describe what you would do differently, other than the two algorithms identified in I3, if you did this project again.

Zip codes were removed from the distance table for convenience and brevity, as the test data primarily included Salt Lake City addresses with only few exceptions. Nested hash tables were used to connect addresses in package IDs to the addresses in the distance table. If the company were to expand beyond the Salt Lake City area, this would not be viable, as this could result in duplicate addresses, which would not work with the hash table, which requires unique key identifiers to be effective. This could be prevented by adding zip codes into the package table addresses or adding an address ID column to the distance table.

Improvements could be made to the optimal path algorithm so that no package preload is required by picking core packages based on route information instead of massaging prioritized packages into the algorithm. Additionally, truck information is hardcoded into the program. I would have also liked to expand the truck object with additional driver information. Ideally, the truck and driver information would have been derived from a file as it was for the distances and packages.

# K.  Justify the data structure identified in part D

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

### a.  Explain how the time needed to complete the look-up function is affected by changes in the number of packages to be delivered.

Due to the nature of a hash table, lookups are expected to remain at O(1) constant time. *See sections B6, D.*

### b.  Explain how the data structure space usage is affected by changes in the number of packages to be delivered.

Space complexity is expected to scale linearly with a growing dataset.

*See sections B6, D.*

### c.  Describe how changes to the number of trucks or the number of cities would affect the look-up time and the space usage of the data structure.

Changes to the number of trucks will not affect the current implementation of the program, other than increasing the number of elements in an array. Performance may be affected when looking up the earliest truck departure, as it would affect the size of n in an O(n) search.

A growing number of cities would probably necessitate increasing the complexity of the truck table, as it would be ideal to assign certain trucks to different hubs. Time complexity is not expected to be affected. City-zip would ideally be strung together as discussed in Section J. Length of key string is not expected to affect the lookup time.

## 2.  Identify two other data structures that could meet the same requirements in the scenario.

* Array
* Linked list

### Describe how *each* data structure identified in part K2 is different from the data structure used in the solution.

An array solution would be smaller, though less expandable. Performance losses would be expected to occur upon expanding datasets. The order would have to be maintained so that a binary search would be the most efficient method for lookups at O(logN) time, which is still much worse than the O(1) constant time of the hash table.

A linked list solution would be only slightly larger than an array. Expansion would be practically free. However linear traversal is required for lookups, which would result in a worst case of O(N).

# Misc. Notes

The solution implemented returns trucks to the hub after the last delivery. If an inputted time occurs between the last package delivered and the return to hub, the status lookup will intentionally identify the truck as en route, as the truck is en route to the hub.

Mileage is a function of time, not delivery completion. If an inputted time occurs between deliveries, partial mileage will be added to the cumulative total.

# References

Selkow, S., Heineman, G.T., & Pollice, G. (2008). *Algorithms in a Nutshell* [eBook edition]. O’Reilly Media. <https://learning.oreilly.com/library/view/algorithms-in-a/9780596516246/>

Stephens, R. (2013). *Essential Algorithms: A Practical Approach to Computer Algorithms* [eBook edition]. Wiley. <https://learning.oreilly.com/library/view/essential-algorithms-a/9781118612101/>

zyBooks. *Data Structures and Algorithms II* (2018). zyBooks.com