# C950 WGUPS Algorithm Overview

Justin A. Langley
ID #001036634
WGU Email: jlang69@wgu.edu
2022-04-16

C950 Data Structures and Algorithms II

## Introduction

In the scenario provided, we explore a parcel service with a growing need to determine efficient routes, or vehicle routing for providing deliveries to their customers.

This can be classified as an NP-hard category problem known as the traveling salesperson problem.. According to Cormen et al., the problem statement is "a salesman must visit *n* cities" (p.1096).

The solutions proposed here cannot be verified to perfectly solve this problem (P vs. NP). We define optimal here as "looks optimal", but recognize that we can't certify it's accuracy when compared to the perfect solution that must exist, since it cannot be computed in polynomial time.

The problem is characterized by finding the optimal, or best-fit solution, to the order of cities visited. The problem is additionally modeled by a complete graph $K_n$, containing all connections between cities. (Cormen et al., p. 1096) In our case, we have additional parameters to solve for, such as distributing *n* packages to *m* trucks, and then solving for the optimal path using a solution to the traveling salesperson problem. Our model still consists of a complete graph $K_n$, but requires dividing $K_n$ into smaller subsets by vehicle. Thus, the problem will be referred to as the **vehical routing problem**. Additionally, the complete code, will be referred to as the solution.

## A. Algorithm Identification

The self-adjusting algorithm proposed for resolving the problem this parcel service is facing is: 'Convex Hull with Heuristic Improvement of Path Construction'.

The chosen algorithm requires several core components, some of which we will take for granted as existing elsewhere, but readily accessible for the sake of brevity in the pseudocode:

Data objects -- models like package, address, coordinate, truck
Utilities -- any utilities that are not part of the core algorithms

Firstly, we have a Ruler object definition. This contains logic for computing distance between two points (not shown), and orientation of ordered point triplets. The function for computing orientation of points will be important to the driving logic behind our path construction algorithm.

Next, we have a convex-hull routine, and it is the main logic for route optimizations once a truck is loaded. The main procedure calls a subroutine, the Jarvis march algorithm. The Jarvis march algorithm "simulates wrapping a taut piece of paper" around a set of points to form a convex "hull", or Q. (Cormen et al., p. 1037). The Jarvis march algorithm comes from the field of computational geometry, and was developed by R. A. Jarvis in 1973.

Using Q, from Jarvis march, the main procedure performs an insertion sort using a path cost heuristic to build the optimal path.

The caller for performing route optimization is a truck dispatching procedure, that additionally calls two other methods, one load method for generating loaded trucks, and one delivery method for generating trip data (i.e. package ids delivered by a truck on a given trip, the distance traveled, and updating the next departure time).

The load method performs truck loading based on a priority queue, by package delivery deadline, and by using additional checks to determine if a package can be loaded, or not (i.e. has a truck requirement, delayed, invalid address, or it has dependencies but the truck does not have enough remaining space). It continues until trucks are at capacity, or no packages remain to be loaded.

Lastly, delivery method accepts the route from the convex-hull path construction routine, and performs time calculations, and calculates trip distances by performing edge lookups in the graph containing address to address distance data.

This concludes our synopsis of core algorithms contained in the solution.

## B1. Logic Comments

The pseudocode below contains all of the routines used for load, path optimization, and delivery. For the sake of brevity, we assume necessary data structures are already in place. All other code containing data models or utility functions not described here is deemed trivial and is excluded.

Ruler is a helper class found in the `ruler` module of the `core` package. It is used for computing distances between points for building the graph, and also includes the `orientation` method for determining ‘clockwiseness’ of a point in relation to two others through line slope calculations. (Jarvis, R., 1973)

```
# O(1), constant time complexity to compute point orientations
CLASS Ruler:
  FUNCTION orientation
    # three points passed in to check, as Coordinate objects
    Initialize P, Q, R as Coordinate

    # difference in slope between the lines P → Q and Q → R
    value = (Q.latitude - P.latitude) * (R.longitude – Q.longitude)  # theta
    value -= (Q.longitude - P.longitude) * (R.latitude – Q.latitude) # phi
    IF value == 0:
      RETURN 0,           P → Q  is collinear to Q → R
    ELSE IF value > 0:
```

```
      RETURN 1,          P → Q is clockwise to Q → R
    ELSE:
      RETURN 2,          P → Q is counter-clockwise to Q→R
    ENDIF
  END orientation
END Ruler
```

The path construction algorithm is found in the `tsp` module of the `core` package.
It contains a class with a wrapper for choosing construction method for computing paths, but only the following algorithm is included.

```
# O(n^2), quadratic time complexity. There is a certain component for building the hull,
# but this is negligible compared to the portion to build the full path using an insertion sort.
FUNCTION convex_hull
  SUBROUTINE jarvis
    Initialize ruler # a Ruler object for measuring point orientation
    Initialize leftmost,# the point with the smallest longitude
    Initialize index, # the index of leftmost in the list of points
    Initialize L, # set equal to index
    Initialize hull, # an array holding the point with smallest longitude
    WHILE True:
      Initialize Q, (L + 1) % number of points
      FOR index in range(number of points):
        # starting point is already added, skip it
        IF index == L:
          CONTINUE
        ENDIF
        # measure the orientation of the point coordinate triplets
        D = CALL ruler.orientation on (points[L], points[index], points[Q])

        IF D == 2:          Only consider counter-clockwise points
          Q = index
        ENDIF

      L = Q
      IF L == index:# stop when we reach the starting index, so the path is not closed
        BREAK
      ENDIF
      APPEND points[Q] to hull
      END LOOP
    END LOOP
    RETURN hull
  END jarvis
```

This section below the subroutine is the driving code calling the above subroutine, and then performs the insertion sort for adding path points by cost heuristic to the convex hull to form the path.

```
# Note: points should have access to either x,y coordinates
# or in our case, latitude, longitude GPS data
Initialize points,          ← location data to optimize
Initialize graph,           ← graph object passed in
Initialize hub,             ← hub location data

IF points < 3:
   RETURN None              ← Finding point orientations requires at least 3 points
ENDIF
# build the hull, with the jarvis march gift wrapping algorithm
hull = CALL jarvis with points

# insert each remaining point into the hull
# by using insertion sort with a cost heuristic
WHILE points:
   Initialize best_ratio,           set to maximum floating point value
   Initialize best_point_index,     set to none
   Initialize insert_index,         set to 0
   # look through remaining points
   # i is to show points should be indexable
   FOR i, point in points:
      Initialize best_hull_cost,    set to maximum floating point value
      Initialize best_hull_index,   set to 0

      # look through points in the hull
      # j is to show hull points should be indexable
      FOR j, hull_point in hull:
         # the next point beyond our current point on the hull
         next_hull_point = hull[(j + 1) % len(hull)]

         # evaluate the cost by accessing the graph passed in
         # the edge containing the points from the inner and outer loops

         # the cost of the current connection explored, minus the existing
         # connection between the two points on the hull
         evalualuted_cost = graph[hull_point][point]
         evalualuted_cost -= graph[hull_point][next_hull_point]
```

```
            # update with the current best candidate for the
            # hull point to insert after, so far
            IF evaluated_cost < best_hull_cost:
               best_hull_cost = evaluated cost
               best_hull_index = j
            ENDIF
         END LOOP

         # retrieve the next point on the hull
         next_point = hull[(best_hull_index) % len(hull)]

         # the cost before "splicing" in our new point
         previous_cost = graph[hull[best_hull_index]][next_point]

         # the combined cost of the new edges made if we "splice" our point
         # between the two nearest points on the hull
         new_cost = graph[hull[best_hull_index]][point]
         new_cost += graph[point][next_point]
         ratio = new_cost / previous_cost

         # update our cost heuristics if this point is the best candidate
         IF ratio < best_ratio:
            best_ratio = ratio
            best_point_index = i
            insert_index = best_hull_index + 1
         ENDIF
      END LOOP
   END LOOP

   # insert the point in the correct position on the hull
   # and remove it from the remaining points to be added
   REMOVE next_point from points
   INSERT next_point into hull at insert_index

   ROTATE hull, # until hub is at index 0
   APPEND hub to hull, # to form a loop
   RETURN hull
END convex_hull
```

The Hub class is found in the `hub` module of the `core` package.
It handles all of the load/delivery computations, but delegates path optimizations to the above pseudocode for performing path construction.
The pseudocode below describes the functions it contains for performing data transformation.

```
# O(m•n^2logn), the path construction algorithm influences it to at least quadratic time.
# Additionally, the sort to build the priority queue adds a logarithmic component.
# Finally, these components are performed by the number of trucks being computed.
# This forces the upper bound to be somewhere between quadratic and cubic time complexity.
FUNCTION dispatch_trucks
   Intialize trip_counts,     an array for storing the number of trips a truck at a given index takes
   WHILE packages remain:
      CALL load_remaining
      FOR each index, truck:
         IF truck.packages is None:

            increment truck departure time (by 1s in the implementation)
            CONTINUE

         ELSE:

            path = CALL convex_hull with Graph, and Hub address
            IF path length is > 2:,       path has data between (hub → hub)
               CALL deliver_packages with truck, trip_counts[index], path
            ENDIF

         ENDIF
      END LOOP
   END LOOP
END dispatch_trucks

# Combined time complexity of O(m•nlogn), between quadratic and linearihmic
FUNCTION load_remaining

   # O(1)
   SUBROUTINE is_loadable
      IF package has truck requirement:
         IF package has truck requirement not equal to current truck:
            RETURN False
         ENDIF
      ENDIF
      IF package has delay:
         IF package.arrival > truck.departure_time:
            RETURN False
         ENDIF
      ENDIF
      IF package has dependencies or is depended on:
```

```
        IF truck has space < len(dependencies) + 1
          RETURN False
        ENDIF
      ENDIF
      IF package has invalid flag set:
        IF update_time > truck.departure time:
          RETURN False
        ENDIF
      ENDIF
      RETURN True,          ← all checks passed
  END is_loadable


# O(n), could require an implementation for updating the master data structure, which may force a search
  SUBROUTINE place_on_truck
      IF package has invalid flag set:
        CALL package update
        MOVE package to truck.packages
      ENDIF
      IF package has no dependencies
        MOVE package to truck.packages
      ENDIF
      IF package has dependencies or is depended on:
        MOVE package with all dependences to truck.packages
      ENDIF
  END place_on_truck

  # O(n), packages and trucks must be looped to build the list of remaining packages
  SUBROUTINE update_remaining
      Initialize loaded,       empty list of ids
      FOR truck in trucks
        extend loaded with package ids onboard truck
      END LOOP

      FOR package in packages:
        IF package.id not in loaded:
          IF package.status is not delivered
            APPEND package to remaining
          ENDIF
        ENDIF
      END LOOP
  END update_remaining
```

```
    # O(m•nlogn), the remaining package list is looped for each truck, and a sort is required
    #                   the upper bound is above linearithmic, but not quite O(n^2)
    Initialize update_time, ← the time package updates are processed
    IF remaining packages is None:
      RETURN
    ENDIF
    queue = [],      ← initialize dummy queue
    FOR i, truck in trucks:
      CLEAR queue
      CLEAR remaining packages
      CALL update_remaining

      IF remaining packages is None:
        RETURN
      ENDIF
      IF len(remaining) packages is None:
        queue = remaining
      ELSE:
        queue = sorted(remaining),          ← in reverse order by deadline
      ENDIF
      WHILE truck is not full:
        IF queue:
          package_to_load = queue.pop()
          IF CALL SUBROUTINE is_loadable with package_to_load:
            CALL load with truck and package_to_load
            HANDLE truck capacity errors IF thrown
          ENDIF
        ELSE:
          BREAK, ← Stop loading, when the queue is empty
        ENDIF
      END LOOP

    CLEAR remaining for next iteration
    CALL update_remaining for next iteration
    END LOOP
END load_remaining

# O(n), package order of each truck must be looped
FUNCTION deliver_packages
    SUBROUTINE calc_distances:
      seen = [],          ← keep track of counted delivery locations
      total = 0.0,        ← running total of trip distance
      Initialize path,    ← list containing address delivery order, passed in
```

```
    prev_point,              ← contains the previous location, initialized to path index 0
    WHILE path:
      current = path[0]
      ADD distance of prev_point → current, to total
      ADD current to seen
      UPDATE prev_point with current
      REMOVE current point from path
    END LOOP
    STORE trip distance in data structure containing list of trip distances indexed by truck id
  END calc_distances

  clock = truck.departure_time
  pids = copy(truck.package_ids)
  trips[clock] = (truck.id, pids),     ← keep a master lookup table of trips at a given time
  total_distance = CALL calc_distance with path, passed in
  FOR i, edges in path:
    calculate time elapsed per visited point and add to clock
    truck.packages[i].delivered = clock
    truck.packages[i].set_status(delivered)
    UPDATE master_table[truck.packages[i]] with current package in truck
  END LOOP
  IF pids:
    t = travel time from last delivery back to hub
    SET next departure time of this truck as the current clock + t
  ENDIF
END deliver_packages
```

This concludes the pseudocode for the core, ‘driving’ logic of the solution.

## B2. Development Environment

The programming environment to build this solution consists of a machine with the following specifications:

Operating System:

Manjaro 5.15.28-1 LTS

Cross-Compatibility Testing:

Windows 10 (Virtual Machine)

Development Environment:

IDE: PyCharm Professional 2021.3.3

Interpreter Configuration: (venv)

Python: 3.10

Hardware:

CPU: AMD Ryzen 9 3900X @ 3.8 GHz

RAM: 32 Gb DDR4 @ 3600 MHz

## B3. Space-Time and Big-O

Load Optimization:

Time Complexity:

O(m•nlogn), time complexity, bounded between linearithmic and quadratic.

There are 3 components forcing the above described time complexity. The first, of which, is the number of trucks operated on. The algorthim is performed across the number of trucks. Secondly, an O(n) operation is required to build a list of packages that require loading. Thirdly, there is a sort by package deadline that runs in O(nlogn), time complexity, the time complexity of Python's very own 'timsort'. (Auger et al., 2018) Combining these into a worst-case analysis yields the time complexity described above.

Space Complexity:

O(n) additional space, an additional array (list) is required for keeping track of remaining packages, and moving them into truck objects.

Convex Hull with Heuristic Improvement of Path Construction:

Time Complexity:

$O(n^2)$, quadratic time complexity.

The Jarvis march algorithm in the subroutine accounts for O(nm) time complexity, where m is the subset of points in n that form the convex hull of n.

The heuristic for path insertion is just an insertion sort algorithm that performs an insertion at the index with the best cost improvement when compared to all other possible candidate points. Because we have to look at each point in the updated path and compare it to all remaining points, this leaves us with $O(n^2)$ for this portion of the algorithm.

This algorithm is bounded by $O(n^2)$ time complexity, the worse of the the two components, the time to find the hull is negligible in comparison to the insertion sort by cost heuristic.

Space Complexity:

The insertion sort used is O(1) additional space complexity because as data is moved into one list it is removed from the other as the path is forming. The Jarvis march algorithm for finding the convex hull is O(m) for storing the hull points.

The path construction requires $O(n^2)$ additional space complexity, which accounts for the use of a graph data structure for performing edge lookups in constant time. Since, the graph is a complete graph, then its space complexity must be $O(n^2)$.

Below is an image representing the Complete Graph $K_8$, the complete graph with our test data is the Complete Graph $K_{27}$. We can see from the representation below that each vertex will 'shake hands' with all others, resulting in the formula k(k-1)/2 to calculate the number of edges. This is determined via Euler's "handshaking lemma". We divide by two to avoid counting edges twice since our graph is undirected. This results in an approximately $O(n^2)$ additional space complexity, as an undirected edge still needs to be stored twice for each vertex pair.

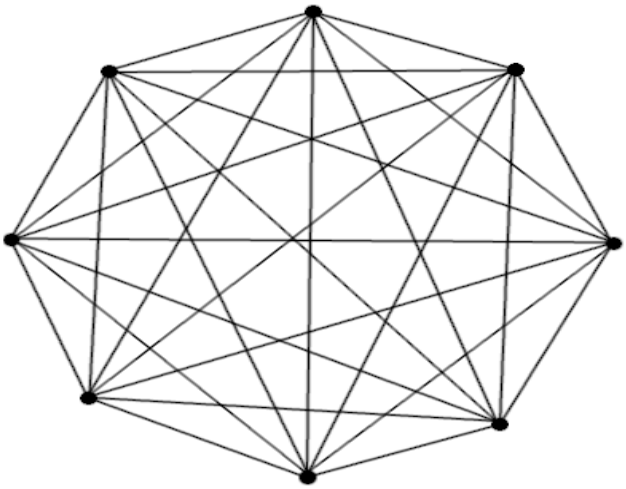

Figure 1: Complete Graph K8

Delivery:

Time Complexity:

O(n), linear time complexity.

The packages a truck contains must be looped and updated. Updating packages can occur in the same loop that distances are computed in.

Space Complexity:

O(mn) additional space is required. The calculated trip distances requires storing the quantity of trips each truck takes. Stored is a truck id combined with its package ids for each trip taken.

Dispatching:

Time Complexity:

$O(m \cdot n^2 \log n)$, the combined time complexity of the above two algorithms.

This algorithm runs somewhere between quadratic and cubic time with an additional logarithmic component. It is guaranteed to run above quadratic time, but whether it... runs in closer to cubic or quadratic time is dependent on the number of trucks undergoing path optimization, and their package capacity.

Space Complexity:

$O(n^2)$ additional space is required. Package addresses must be transformed into a graph data structure, requiring the number of packages by the square of itself to be added as vertices, edges in the graph. The additional space complexity to perform load optimization O(n) is dwarfed by this requirement.

Overall time and space complexity of the program is described by the analysis of the Dispatching routine, $O(m \cdot n^2 \log n)$ time complexity using $O(n^2)$ additional space.

## B4. Scalability and Adaptability

This solution is not so scalable in its present state. However, with some key changes to certain data structures and algorithms within the solution, then the code could become performant enough to begin handling larger data sets. One of the larger issues in its current state is the sorting required to process the data set into a state where priority packages are able to be loaded first.

A better data structure for maintaining an order of priority packages to be used first would be a better candidate than the current data structure for storing package data. A hash table is good for random access, but we need to process a lot of data related to packages sequentially, which requires sorting since a hash table does not preserve insertion order.

Additionally, the algorithm for optimizing delivery routes is an ideal candidate for replacement with something more adaptable, the $O(n^2)$ time complexity it runs in makes it unsuitable for increasingly large data sets.

However, if the data sets were to become large enough the best candidate would be to implement the changes alongside the implementation of a cloud computing solution to offload package/truck subsets to worker machines, and then aggregate the results at a primary location. This would require much extra overhead and time spent refactoring the solution, but since the code footprint is still small, it wouldn't be unfeasible.

## B5. Software Efficiency and Maintainability

Efficiency:

The solution is on the cusp of decent efficiency, as it runs in polynomial time. The combined overall time complexity to transform the data sets is bounded somewhere above $O(n^2)$, but is not quite bad enough to call $O(n^3)$. The upper bound is somewhere like $O(m \bullet n^2 \bullet \log n)$, there is a component derived from the required sorting of packages, another component factoring in the updating of a separate list of remaining packages when loading, rather than using the data in-place, and an additional third component factoring in the time complexity of path optimization. As the number of packages grows large, the truck capacities increase, and the number of trucks in use multiplies then the time complexity is surely nearing the upper bound. It is still probably not quite cubic runtime, but getting very close.

Maintainability:

The code is organized into packages corresponding to their type of information processing.

Those packages are named here:

| | |
|---|---|
| cli | - all code relating to command line interfaces for data onboarding |
| core | - all code containing the core algorithms for data processing i.e. hub, ruler, tsp |
| models | - all code related to objects i.e. truck, package, and address (including coordinate) |
| structures | - all code for maintaining data structures, i.e. graph, hashtable, linkedlist |
| util | - utility functions like string manipulations, time parsing, etc. |
| app.py | - holds the infinite loop, and main interface for interacting with the hub object |
| main.py | - controls walking the user through data onboarding and initiating the app |

This organization benefits software maintainability by segmenting the program into source files of a more comprehensible size. Additionally, where possible if a helper function is only needed by a single other function, the trend is to create it as a subfunction/subroutine, so the function can maintain its own helper. If it would be used more than once in various locations, it gets placed in the `util` package.

Additionally, docstrings were included as needed, and comments placed where more clarification was needed. The code was written to be as self documenting as possible, except for where computations were heavily math based, where naming conventions were oriented towards the math notations.

## B6. Self-Adjusting Data Structures

The self-adjusting data structures present in this solution are:

- Linked List (doubly-linked in the implementation)
- Hash table (with chaining in the implementation)
- Graph (supports directed/undirected edges, but is undirected and complete in its usage)

There are strengths and weaknesses to each of these which we will go through in order, since each of these data structures builds on the last.

1. Linked List

A linked list is a good candidate for creating chains for storing data in our hashtable. Insertion and removal of items operates in O(1) constant time, so 'put' and 'delete' operations will perform in constant time as well once the hash table locates the appropriate linked list 'bucket', and the linked list locates the appropriate node holding the data (for delete only).

The downsides of a linked list implementation are that it is O(n), linear time complexity to search for an item being retrieved. This will require additional code to ensure the linked list 'buckets' or chains don't become excessively large. They should remain small, so there is only ever a known number of items being search when 'put' and 'delete' operations in the hash table access one of these linked list objects.

2. Hash Table with Chaining

   A hash table with chaining is a good candidate for use when the total items placed in it can vary widely. We can implement a simple hashing function and then take the hash of the item and use a modulus operator against the size, or number of buckets, of the hash table to locate its proper placement. The chained nature also handles hashing collisions elegantly.

   Downsides to this are additional code is required to maintain the integrity of the table, so that time complexity on its operations can still amortize to O(1), for 'get', 'put', and 'delete'operations. This means a table resizing routine is required and should be called automatically when storage chains become too large or too small.

3. Graph

   This is a wonderful data structure for accessing large quantities of vertex → vertex (edge) cost values in O(1) time complexity. Since its implementation is a hash table of hash tables, so it receives the same benefit of amortize O(1) time complexity on operations involving vertices, and edges.

   The downsides are that the space complexity is huge for its usage in our solution. It requires $O(n^2)$ additional space in our usage because we require building undirected edges between each vertex and all other vertices in the graph. This makes our graph a complete graph, and represents a mesh if drawn out. Additionally, without storing various initialization parameters, and heavy use of generics, its unwieldy to create the additional code required for calculating vertex degree, in/out degrees, and edge sums. The solution only has the required methods and calculations required for a complete graph $K_n$.

## C. Original Code

Original code was produced for this project with Python 3.10 (using venv) in PyCharm.

## C1. Identification Information

The source code file main.py includes identification information.

## C2. Process and Flow Comments

The source code file main.py includes clarification of where the code flows upon initialization, and the additional locations it flows to include further code comments for additional clarifications on computations that are occuring in those locations.

## D. Data Structure

The hash table implementation is a hash table with chaining. A doubly-linked list using generics is the the underlying structure for holding the packages inside of nodes in these linked list objects. The linked list objects are considered the 'buckets' of the hashtable. The hash table is then built on top of that, also using generics. The implementation includes a resizing function to expand or shrink the number of buckets if their linked lists too keep them from growing too large or becoming too sparse.

## D1. Explanation of Data Structure

Hashing:
According to Cormen et al., p. 258, the 'dictionary' operations on the hash table are:
"

Chained-Hash-Insert (T, x)

1 insert x at the head of the list T[h(x.key]
Chained-Hash-Search(T, k)
1 search for an element with key k in list T[h(k)]
Chained-Hash-Delete(T, x)
1 delete x from the list T[h(x.key)]
"

Cormen et al., on p.258, also describe a "load factor" to be the stored keys, n / m, the number of chains.

The above description was followed for implementing hash table methods, with minor changes.

Following this description a hashing function h, was first produced:
The integer modulo the number of buckets in the table, if the key is an integer.
The result of calling the Python hash() built-in, modulo the number of buckets in the table for all other types.

The load factor was restricted to remain between 4-8, for determining when to resize, and allowing for the time complexities described below. When the delete that would cause the load factor to drop below 4 occurs, the table shrinks by a factor of 2. When insert that would cause the load factor to rize above 8 occurs, the table grows by a facor of 2.

Following the description by Cormen et al. above, and the above load factor requirement, the resizing function was developed, and its triggers were implemented in its operation methods for insert, and delete. The accessing function work by storing (x.key, x) as tuple in the linked list. Since generics were used the code refers to (x.key, x) as the generic types (key, value).

Big-O:
The hashtable follows the time complexity described by Cormen et al., on p.258, Amortized O(1) lookup, Amortized O(1) insertion, Amortized O(1) delete.
This is with the assumption that any item is equally likely to end up in any bucket (though may not be the case in practice). This is inline with the time complexity that should be expected for an appropriately implemented hash table.
There were also some optimizations made in the doubly-linked list structure to ensure the appropriate end of the linked list is searched across, and they're restricted to growing to a maximum of 8 items per linked list 'bucket' in the hashtable.
The worst case is an insert or delete that triggers a resize, where the time complexity is O(n) for that one insert or delete operation, as all items have to undergo a rehash to move to the appropriate bucket number.

Usage:
In our case, the way we end up using the hash table with package data, is that the key is the package ID, and then the value is a custom package object implemented as a Python dataclass. This package ID and package object key get stored as a tuple of (key, value) in the next available node of a given linked list 'bucket'.

## E. Hash Table

The hash table implements __getitem__, __setitem__, __iter__, and its own items() methods for random access and iterating the data it holds. This allows for accessing items using bracket notation, and iterating by calling items() on itself in the loop declaration, to return a generator object. It feels nearly the same as interacting with a Python dict() object, but type hints were included throughout the code, so please feel free to check the type hinting to ensure that it is indeed an original hash table implementation being used.

Storage is maintained by the _hash, and _resize methods, and generic types are used for storing data, making it adaptable to most data types with no code alterations.

There is an additional search method to test if an item exists in the hash table without returning it, and a clear method for wiping its data. This should satisfy all required methods for data access and self-adjustment.

## F. Look-Up Function

Package lookups occur by searching the hash table data structure using the package ID as the lookup key, and the value returned is a custom `Package` object. The hash table searches the... stored `[int, Package]` tuples to find the node with the tuple containing the matching key. If the key is found, then the Package is returned.

The hash table was implemented in such a way to allow accessing and setting items through bracket notation, and an items() method returning a generator, for ease of use.

Hash table lookup typically occurs when a user performs lookups in the cli, but they also happen frequently in the core algorithms within this solution when accessing a package is required.

Data related to delivery trips that trucks take is stored separately, but when a package is required it is retrieved from the hash table containing all packages that was created during program initialization.

The user interface lookup functions are implemented through a `lookup` command in the interface. This command accepts various arguments using a match case statement to check user inputs, then follows with the interface elements for allowing the user to input the search key for the type of search requested. Then, the minimal validation required is performed before calling the lookup functions contained in the hub object.

The lookup cli command allows searching package data by:

- Address
- Deadline
- City
- ID
- Status
- Weight
- Zip

When each required function is called, they pull the requested data from the HashTable that contains package data, then minimal formating is added for printout informing the user of estimated delivery times. Returned search data is ordered by Package ID

The lookup functions also support looking up by a given time key. When this search is called a snapshot function retrieves and transforms the data into a viewable printout. This printout contains package data ordered by status, and in the case of enroute packages, are additionally separated by truck. The packages contained in the truck are sorted in order of delivery. The remaining delivered/undelivered packages are sorted by delivered time in reverse order, or estimated delivery time, respectively. This routine requires separating package from delivery trip data, and programatically determining each package(s) state at the provided time. A copy of each package is updated to the correct state and used to generate the printout.

Additional statistics lookup functions are found by using the `stats` command. It can be called alone for combined distance statistics of all trucks, or with the route option for looking up routing/trip information by truck ID. All lookup operations run in O(n) time, where the search

space is the total number of packages in the data set loaded during onboarding. The exception to this is data ordered by truck which requires $O(n^2)$ time to build the package data for each truck.

## G. Interface

The solution uses a command line interface to walk the user through data onboarding. It displays loading bars and useful information during the process of data onboarding. Afterwards, the user is dropped into a shell style interface with help information to get them started with viewing the transformed data in various ways.

Continued on page 20.

### G1. First Status Check

Snapshot of packages at 08:45 am

```
Type 'help' for command usage:

Available Commands:
help:   'displays help with commands'
lookup: 'looks up delivery information'
stats:  'displays truck statistics'
quit:   'exit the program'
> lookup
Please enter a time in 24-hr format 'HH:MM':
> 08:45
Delivery Snapshot @ 08:45

Remaining:
-------------------------------------------------------------------------------------------------------------
  ID  Street                                   City             ST    Zip    KG   Status
-------------------------------------------------------------------------------------------------------------
  9   300 State St                             Salt Lake City   UT    84103  2    hub
  25  5383 South 900 East #104                 Salt Lake City   UT    84117  7    hub
  24  5025 State St                            Murray           UT    84107  7    hub
  32  3365 S 900 W                             Salt Lake City   UT    84119  1    hub
  6   3060 Lester St                           West Valley City UT    84119  88   hub
  8   300 State St                             Salt Lake City   UT    84103  9    hub
  33  2530 S 500 E                             Salt Lake City   UT    84106  1    hub
  28  2835 Main St                             Salt Lake City   UT    84115  7    hub
-------------------------------------------------------------------------------------------------------------

Enroute on Truck #001:
-------------------------------------------------------------------------------------------------------------
  ID  Street                                   City             ST    Zip    KG   Status
-------------------------------------------------------------------------------------------------------------
  13  2010 W 500 S                             Salt Lake City   UT    84104  2    enroute
  39  2010 W 500 S                             Salt Lake City   UT    84104  9    enroute
  23  5100 South 2700 West                     Salt Lake City   UT    84118  5    enroute
  31  3365 S 900 W                             Salt Lake City   UT    84119  1    enroute
  40  380 W 2880 S                             Salt Lake City   UT    84115  45   enroute
  1   195 W Oakland Ave                        Salt Lake City   UT    84115  21   enroute
  19  177 W Price Ave                          Salt Lake City   UT    84115  37   enroute
  20  3595 Main St                             Salt Lake City   UT    84115  37   enroute
-------------------------------------------------------------------------------------------------------------

Enroute on Truck #002:
-------------------------------------------------------------------------------------------------------------
  ID  Street                                   City             ST    Zip    KG   Status
-------------------------------------------------------------------------------------------------------------
  17  3148 S 1100 W                            Salt Lake City   UT    84119  2    enroute
  36  2300 Parkway Blvd                        West Valley City UT    84119  88   enroute
  12  3575 W Valley Central Station bus Loop   West Valley City UT    84119  1    enroute
  11  2600 Taylorsville Blvd                   Salt Lake City   UT    84118  1    enroute
  18  1488 4800 S                              Salt Lake City   UT    84123  6    enroute
  22  6351 South 900 East                      Murray           UT    84121  2    enroute
  26  5383 South 900 East #104                 Salt Lake City   UT    84117  25   enroute
-------------------------------------------------------------------------------------------------------------

Delivered:
-------------------------------------------------------------------------------------------------------------
  35  1060 Dalton Ave S                        Salt Lake City   UT    84104  88   delivered 08:36:00 AM
  27  1060 Dalton Ave S                        Salt Lake City   UT    84104  5    delivered 08:36:00 AM
  30  300 State St                             Salt Lake City   UT    84103  1    delivered 08:36:00 AM
  37  410 S State St                           Salt Lake City   UT    84111  2    delivered 08:32:20 AM
  3   233 Canyon Rd                            Salt Lake City   UT    84103  2    delivered 08:27:20 AM
  5   410 S State St                           Salt Lake City   UT    84111  5    delivered 08:24:00 AM
  38  410 S State St                           Salt Lake City   UT    84111  9    delivered 08:24:00 AM
  29  1330 2100 S                              Salt Lake City   UT    84106  2    delivered 08:21:20 AM
  7   1330 2100 S                              Salt Lake City   UT    84106  8    delivered 08:21:20 AM
  10  600 E 900 South                          Salt Lake City   UT    84105  1    delivered 08:19:40 AM
  2   2530 S 500 E                             Salt Lake City   UT    84106  44   delivered 08:12:40 AM
  16  4580 S 2300 E                            Holladay         UT    84117  88   delivered 08:08:40 AM
  34  4580 S 2300 E                            Holladay         UT    84117  2    delivered 08:08:40 AM
  15  4580 S 2300 E                            Holladay         UT    84117  4    delivered 08:08:40 AM
  4   380 W 2880 S                             Salt Lake City   UT    84115  4    delivered 08:08:00 AM
  21  3595 Main St                             Salt Lake City   UT    84115  3    delivered 08:04:00 AM
  14  4300 S 1300 E                            Millcreek        UT    84117  88   delivered 08:03:00 AM
-------------------------------------------------------------------------------------------------------------

Press <ENTER> to continue ...
>
```

**Figure 2: Delivery progress at a searched time**

## G2. Second Status Check

Snapshot of package statuses at 10:19 am

```
Type 'help' for command usage:

Available Commands:
help:   'displays help with commands'
lookup: 'looks up delivery information'
stats:  'displays truck statistics'
quit:   'exit the program'
> lookup
Please enter a time in 24-hr format 'HH:MM':
> 10:19
Delivery Snapshot @ 10:19

Remaining:
-------------------------------------------------------------------------------------------------------------
  ID  Street                                   City             ST    Zip    KG    Status
-------------------------------------------------------------------------------------------------------------
  9   300 State St                             Salt Lake City   UT    84103  2     hub
-------------------------------------------------------------------------------------------------------------

Enroute on Truck #001:
-------------------------------------------------------------------------------------------------------------
  ID  Street                                   City             ST    Zip    KG    Status
-------------------------------------------------------------------------------------------------------------
  6   3060 Lester St                           West Valley City UT    84119  88    enroute
  32  3365 S 900 W                             Salt Lake City   UT    84119  1     enroute
  24  5025 State St                            Murray           UT    84107  7     enroute
  25  5383 South 900 East #104                 Salt Lake City   UT    84117  7     enroute
-------------------------------------------------------------------------------------------------------------

Enroute on Truck #002:
-------------------------------------------------------------------------------------------------------------
  ID  Street                                   City             ST    Zip    KG    Status
-------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------

Delivered:
-------------------------------------------------------------------------------------------------------------
  8   300 State St                             Salt Lake City   UT    84103  9     delivered 10:05:00 AM
  33  2530 S 500 E                             Salt Lake City   UT    84106  1     delivered 09:51:00 AM
  28  2835 Main St                             Salt Lake City   UT    84115  7     delivered 09:48:20 AM
  20  3595 Main St                             Salt Lake City   UT    84115  37    delivered 09:33:40 AM
  19  177 W Price Ave                          Salt Lake City   UT    84115  37    delivered 09:32:40 AM
  26  5383 South 900 East #104                 Salt Lake City   UT    84117  25    delivered 09:27:00 AM
  1   195 W Oakland Ave                        Salt Lake City   UT    84115  21    delivered 09:27:00 AM
  40  380 W 2880 S                             Salt Lake City   UT    84115  45    delivered 09:25:00 AM
  22  6351 South 900 East                      Murray           UT    84121  2     delivered 09:22:40 AM
  31  3365 S 900 W                             Salt Lake City   UT    84119  1     delivered 09:21:40 AM
  23  5100 South 2700 West                     Salt Lake City   UT    84118  5     delivered 09:10:00 AM
  18  1488 4800 S                              Salt Lake City   UT    84123  6     delivered 09:08:40 AM
  11  2600 Taylorsville Blvd                   Salt Lake City   UT    84118  1     delivered 09:04:00 AM
  12  3575 W Valley Central Station bus Loop   West Valley City UT    84119  1     delivered 08:54:40 AM
  36  2300 Parkway Blvd                        West Valley City UT    84119  88    delivered 08:50:40 AM
  13  2010 W 500 S                             Salt Lake City   UT    84104  2     delivered 08:47:00 AM
  39  2010 W 500 S                             Salt Lake City   UT    84104  9     delivered 08:47:00 AM
  17  3148 S 1100 W                            Salt Lake City   UT    84119  2     delivered 08:46:20 AM
  35  1060 Dalton Ave S                        Salt Lake City   UT    84104  88    delivered 08:36:00 AM
  27  1060 Dalton Ave S                        Salt Lake City   UT    84104  5     delivered 08:36:00 AM
  30  300 State St                             Salt Lake City   UT    84103  1     delivered 08:36:00 AM
  37  410 S State St                           Salt Lake City   UT    84111  2     delivered 08:32:20 AM
  3   233 Canyon Rd                            Salt Lake City   UT    84103  2     delivered 08:27:20 AM
  5   410 S State St                           Salt Lake City   UT    84111  5     delivered 08:24:00 AM
  38  410 S State St                           Salt Lake City   UT    84111  9     delivered 08:24:00 AM
  29  1330 2100 S                              Salt Lake City   UT    84106  2     delivered 08:21:20 AM
  7   1330 2100 S                              Salt Lake City   UT    84106  8     delivered 08:21:20 AM
  10  600 E 900 South                          Salt Lake City   UT    84105  1     delivered 08:19:40 AM
  2   2530 S 500 E                             Salt Lake City   UT    84106  44    delivered 08:12:40 AM
  16  4580 S 2300 E                            Holladay         UT    84117  88    delivered 08:08:40 AM
  34  4580 S 2300 E                            Holladay         UT    84117  2     delivered 08:08:40 AM
  15  4580 S 2300 E                            Holladay         UT    84117  4     delivered 08:08:40 AM
  4   380 W 2880 S                             Salt Lake City   UT    84115  4     delivered 08:08:00 AM
  21  3595 Main St                             Salt Lake City   UT    84115  3     delivered 08:04:00 AM
  14  4300 S 1300 E                            Millcreek        UT    84117  88    delivered 08:03:00 AM
-------------------------------------------------------------------------------------------------------------

Press <ENTER> to continue ...
>
```

**Figure 3: Delivery progress at a searched time**

G3. Third Status Check

Snapshot of package statuses at 12:07 pm

```
Type 'help' for command usage:

Available Commands:
help:   'displays help with commands'
lookup: 'looks up delivery information'
stats:  'displays truck statistics'
quit:   'exit the program'
> lookup
Please enter a time in 24-hr format 'HH:MM':
> 12:07
Delivery Snapshot @ 12:07

No packages in hub at this time.

Enroute on Truck #001:
------------------------------------------------------------------------------------------------------------
  ID  Street                                   City             ST     Zip    KG    Status
------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------

Enroute on Truck #002:
------------------------------------------------------------------------------------------------------------
  ID  Street                                   City             ST     Zip    KG    Status
------------------------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------------

Delivered:
------------------------------------------------------------------------------------------------------------
  9   410 S State St                           Salt Lake City   UT     84111  2     delivered 11:12:59 AM
  25  5383 South 900 East #104                 Salt Lake City   UT     84117  7     delivered 10:41:19 AM
  24  5025 State St                            Murray           UT     84107  7     delivered 10:36:59 AM
  32  3365 S 900 W                             Salt Lake City   UT     84119  1     delivered 10:27:20 AM
  6   3060 Lester St                           West Valley City UT     84119  88    delivered 10:23:20 AM
  8   300 State St                             Salt Lake City   UT     84103  9     delivered 10:05:00 AM
  33  2530 S 500 E                             Salt Lake City   UT     84106  1     delivered 09:51:00 AM
  28  2835 Main St                             Salt Lake City   UT     84115  7     delivered 09:48:20 AM
  20  3595 Main St                             Salt Lake City   UT     84115  37    delivered 09:33:40 AM
  19  177 W Price Ave                          Salt Lake City   UT     84115  37    delivered 09:32:40 AM
  26  5383 South 900 East #104                 Salt Lake City   UT     84117  25    delivered 09:27:00 AM
  1   195 W Oakland Ave                        Salt Lake City   UT     84115  21    delivered 09:27:00 AM
  40  380 W 2880 S                             Salt Lake City   UT     84115  45    delivered 09:25:00 AM
  22  6351 South 900 East                      Murray           UT     84121  2     delivered 09:22:40 AM
  31  3365 S 900 W                             Salt Lake City   UT     84119  1     delivered 09:21:40 AM
  23  5100 South 2700 West                     Salt Lake City   UT     84118  5     delivered 09:10:00 AM
  18  1488 4800 S                              Salt Lake City   UT     84123  6     delivered 09:08:40 AM
  11  2600 Taylorsville Blvd                   Salt Lake City   UT     84118  1     delivered 09:04:00 AM
  12  3575 W Valley Central Station bus Loop   West Valley City UT     84119  1     delivered 08:54:40 AM
  36  2300 Parkway Blvd                        West Valley City UT     84119  88    delivered 08:50:40 AM
  13  2010 W 500 S                             Salt Lake City   UT     84104  2     delivered 08:47:00 AM
  39  2010 W 500 S                             Salt Lake City   UT     84104  9     delivered 08:47:00 AM
  17  3148 S 1100 W                            Salt Lake City   UT     84119  2     delivered 08:46:20 AM
  35  1060 Dalton Ave S                        Salt Lake City   UT     84104  88    delivered 08:36:00 AM
  27  1060 Dalton Ave S                        Salt Lake City   UT     84104  5     delivered 08:36:00 AM
  30  300 State St                             Salt Lake City   UT     84103  1     delivered 08:36:00 AM
  37  410 S State St                           Salt Lake City   UT     84111  2     delivered 08:32:20 AM
  3   233 Canyon Rd                            Salt Lake City   UT     84103  2     delivered 08:27:20 AM
  5   410 S State St                           Salt Lake City   UT     84111  5     delivered 08:24:00 AM
  38  410 S State St                           Salt Lake City   UT     84111  9     delivered 08:24:00 AM
  29  1330 2100 S                              Salt Lake City   UT     84106  2     delivered 08:21:20 AM
  7   1330 2100 S                              Salt Lake City   UT     84106  8     delivered 08:21:20 AM
  10  600 E 900 South                          Salt Lake City   UT     84105  1     delivered 08:19:40 AM
  2   2530 S 500 E                             Salt Lake City   UT     84106  44    delivered 08:12:40 AM
  16  4580 S 2300 E                            Holladay         UT     84117  88    delivered 08:08:40 AM
  34  4580 S 2300 E                            Holladay         UT     84117  2     delivered 08:08:40 AM
  15  4580 S 2300 E                            Holladay         UT     84117  4     delivered 08:08:40 AM
  4   380 W 2880 S                             Salt Lake City   UT     84115  4     delivered 08:08:00 AM
  21  3595 Main St                             Salt Lake City   UT     84115  3     delivered 08:04:00 AM
  14  4300 S 1300 E                            Millcreek        UT     84117  88    delivered 08:03:00 AM
------------------------------------------------------------------------------------------------------------

Press <ENTER> to continue ...
>
```

**Figure 4: Delivery progress at a searched time**

## H. Screenshots of Code Execution

Various menus, prompting, and loading bars create the user experience.

Polling distance statistics with the `stats` command

```
Type 'help' for command usage:

Available Commands:
help:   'displays help with commands'
lookup: 'looks up delivery information'
stats:  'displays truck statistics'
quit:   'exit the program'
> stats
Distance Statistics:

Truck #001:
Total Distance: 59.9 mi


Truck #002:
Total Distance: 28.2 mi

Combined: 88.1 mi

(Note: To see route information for this truck, use the `stats route` command)

Press <ENTER> to continue ...
>
```

**Figure 5: Combined distance statistics for all deliveries**

Example of loading bars and pertinent information updates while the user loads data

```
Processing Locations

Computing addresses
 |                                                                  | 100%
  27 addresses.

Computing connections
 |                                                                  | 100%
  351 connections.

Done.
Press <Enter> to continue ...
>
```

**Figure 6: Example of keeping the user informed**

Continued on page 24.

Options for choosing which data files to load during data onboarding.

Files found in the data folder are ordered by most recently modified

```
Please select a gps file:
+-------------------------------------------------------------------------+
  #:  File                                      Ext  Last Modified
+-------------------------------------------------------------------------+
  0:  gps_coords                                .csv  2022-03-28 19:01:28

> 
```

**Figure 7: Example of prompting the user for data inputs**

Polling route statistics with the `stats route` command with Truck ID input

```
Type 'help' for command usage:

Available Commands:
help:   'displays help with commands'
lookup: 'looks up delivery information'
stats:  'displays truck statistics'
quit:   'exit the program'
> stats route
Please input truck ID:
> 1
Route plans for Truck #001:
(Note: Packages are in sorted delivery order.)

Trip 1:
Planned Mileage: 29.3 mi
  14  4300 S 1300 E                       Millcreek         UT     84117  88    hub
  16  4580 S 2300 E                       Holladay          UT     84117  88    hub
  15  4580 S 2300 E                       Holladay          UT     84117  4     hub
  34  4580 S 2300 E                       Holladay          UT     84117  2     hub
  29  1330 2100 S                         Salt Lake City    UT     84106  2     hub
  7   1330 2100 S                         Salt Lake City    UT     84106  8     hub
  37  410 S State St                      Salt Lake City    UT     84111  2     hub
  30  300 State St                        Salt Lake City    UT     84103  1     hub
  13  2010 W 500 S                        Salt Lake City    UT     84104  2     hub
  39  2010 W 500 S                        Salt Lake City    UT     84104  9     hub
  23  5100 South 2700 West                Salt Lake City    UT     84118  5     hub
  31  3365 S 900 W                        Salt Lake City    UT     84119  1     hub
  40  380 W 2880 S                        Salt Lake City    UT     84115  45    hub
  1   195 W Oakland Ave                   Salt Lake City    UT     84115  21    hub
  19  177 W Price Ave                     Salt Lake City    UT     84115  37    hub
  20  3595 Main St                        Salt Lake City    UT     84115  37    hub
Trip 2:
Planned Mileage: 20.0 mi
  28  2835 Main St                        Salt Lake City    UT     84115  7     hub
  33  2530 S 500 E                        Salt Lake City    UT     84106  1     hub
  8   300 State St                        Salt Lake City    UT     84103  9     hub
  6   3060 Lester St                      West Valley City  UT     84119  88    hub
  32  3365 S 900 W                        Salt Lake City    UT     84119  1     hub
  24  5025 State St                       Murray            UT     84107  7     hub
  25  5383 South 900 East #104            Salt Lake City    UT     84117  7     hub
Trip 3:
Planned Mileage: 10.6 mi
  9   410 S State St                      Salt Lake City    UT     84111  2     hub
```

**Figure 8: Example of viewing route data**

## I1. Strengths of Chosen Algorithm

The key strength of the chosen algorithm is that by using GPS data, and some math for calculating geographic distances between given latitude and longitude coordinates, we can build an optimal path for a given truck in one run of the algorithm. Other algorithms like nearest neighbor can require an algorithms for performing inversions on points to resolve them to an optimal round trip with no crossovers. We can be assured that by building a convex hull and constructing the path by inserting points on a cost basis, that points are inserted in their correct locations, without having to perform additional point inversions until crossovers in the path are removed.

## I2. Verification of Algorithm

From the screenshots of code execution we can see that the algorithm performed properly to the pseudocode specification.

We have trip distances calculated for both trucks. We can see the subtotals in the planned route statistics, and verify that they indeed total to 88.1 miles, this assures us that the computations are being performed correctly for adding trip distances. There are no trips with a distance of 0.0, and they are all properly included in the totals.

Additionally, inspecting the route information and the snapshots by time, we can see that no package was ever loaded more than once, i.e. no package was placed onto both trucks at the same time. This assures us that our loading algorithm is correctly reserving back packages that have already been loaded on a truck.

We can see package with ID of 9, is has the invalid address at 10:19 am, and then at, 10:20 am, and afterwards, the address is shown to be the updated address. The update to the address occurs before the package is loaded and by verifying with an online maps tool, we can see that the distances to and from that package are properly computed. No delayed packages are loaded before their arrival. Packages with truck requirements go on the proper truck, and all dependencies are loaded onto the same truck on the same trip.

We can additionally verify through the interface through delivery timestamps that all packages with a deadline requirement have a delivered timestamp before the deadline.

By plotting addresses on a map and drawing connecting lines we can verify that the path optimization algorithm indeed optimizes the paths properly in one run of the algorithm specified by the pseudo code. No path crossovers are present in the path produced.

All output is complete and correct according to the solution specifications.

Continued on page 26.

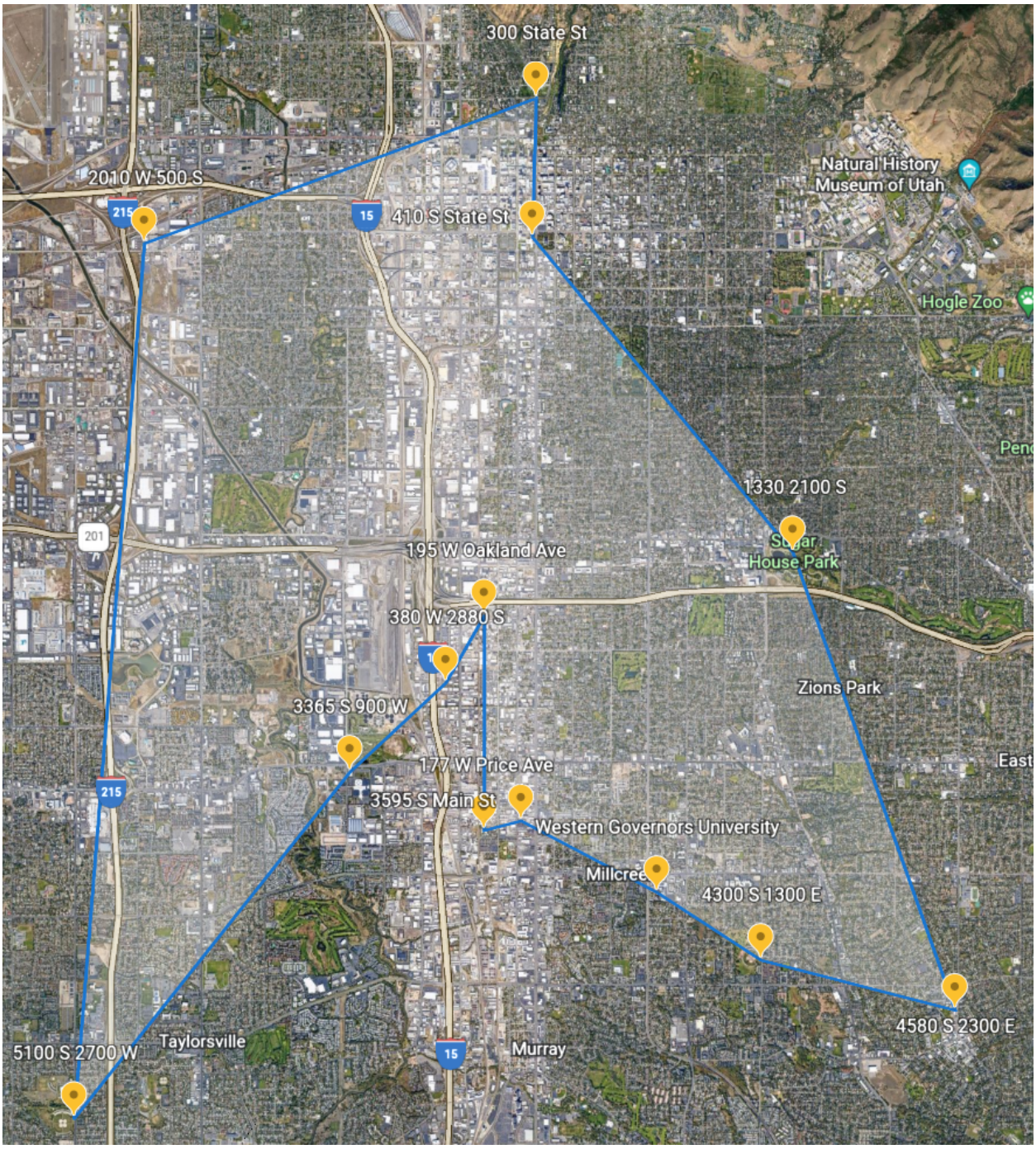


**Figure 9: Optimized route produced by the solution**

## I3. Other possible Algorithms

An exciting other possible algorithm for solving the vehicle routing problem is Ant Colony Optimization, a metaheuristic algorithm (Dorigo & Stützle, 2004). This algorithm would have a good application for usage in cloud computing as the problem set grows very large. The worker ants described by Dorigo and Stützle, could be distributed worker machines and the colony the described, which performs the daemon actions could be a central server that aggregates worker results.

Paraphrasing, Dorigo and Stützle identify an Ant Colony as:

“

```
Set parameters, initialize pheromone trails
SCHEDULE_ACTIVITIES
   Construct Ant Solutions
   Daemon Actions    {optional}
   Update Pheromones
 END_SCHEDULE_ACTIVITIES
```

“ (Dorigo & Stützle, pp.125-126, 2004)

The algorithm consists of generating worker ants having access to a graph structure, then they explore a path in a pseudo-random order. These are the ant solutions constructed.

After the ants explore solutions, there are some daemon actions that occur, possibly ranking the solutions by evaluated cost are generating other metrics for determing the best solution overall.

Then, a “pheromone” is applied to the best solution found, so far. On the next iteration, ants explore again, with an incentive to stick to the previous path “pheromone” was applied to, with some random variability so the path isn’t completely reinforced. Additionally, a “pheromone” decay rate is used to ensure old paths explored that no longer have the best cost fade away. Over time, a close to optimal path is formed by the exploring ants, though it can never be said to be perfect.

Another second algorithm, borrowing again from nature, is the genetic metaheuristic algorithm. It is similar to the Ant Colony Optimization algorithm, but instead of using ant workers, paths are simply explored at random, “genetic variability”. Then, the best ones are chosen, “natural selection”, and used to mutate, or “cross-over”, with older explored paths, to provide the incentive to find new paths. Over time, the algorithm is expected to find a close to optimal solution. This algorithm would be great for the same reasons stated above with Ant Colony Optimization.

## I3A. Algorithm Differences

The two algorithms above are quite similar, to each other.

One key difference is the naming conventions. One refers to its workers as ants, the other simply has workers.

Instead of a pheromone trail with a decay rate, the genetic algorithm, uses the idea of "genetic mutation", and "natural selection" to create varying paths explored.

The "crossover" mechanism also works differently from the "pheromone" trails in Ant Colony Optimization. The genetic algorithm splices parts of the new path with parts from the best so far to introduce further pseudo-random variances.

## J. Different Approach

This task would be approached differently the next time around, in a few key ways, some of which include forethought included by this solution.

One of which, is a feature to allow the user to select the path construction algorithm. One of the above different algorithms would be introduced into the wrapper class for path optimization. This would simply build on top of the existing code base, so the current code is already extensible enough to support this.

Another change that would be good to make is use a different data structure that supports the priority queue required in truck loading. A structure like an AVL tree would be a good candidate to implement, because it supports items "bubbling" to the top. The tradeoff is O(logn) insertions, but the reward is packages would already be in order for loading. Since the package data only need be loaded once, this increase in time complexity upfront for insertion would reduce time complexity later on where performing sorting is normally required. Packages could be pulled off as they are loaded, and then placed in a hash table or dictionary keyed by truck and trip to easily retrieve packages handled by a given truck. This would make lookups O(1) when users request truck specific data and O(n) for requesting data of all trucks.

Overall, the second set of changes would reduce the overall runtime complexity to just $O(m \cdot n^2)$.

If further modification could lower the time to construct an optimal path, then the solution could go below $O(n^2)$, but no further modifications have been identified, thus far.

## K1. Verification of Data Structure

The data structures meet the solution requirments by achieving the following:

- A data structure exists to hold all trip data, allowing trip mileage to be totaled by trip, by truck, and by all combined trucks.
- An efficient hash table, amortized O(1) time complexiy on all operations, with a lookup function by key, is present for maintaining package related data.

- All packages are delivered on time according to delivery specifications. By checking the hash table that is used as a master lookup table, we verify that this is true. Additionally, once packages are delivered this hash table is unchanging. After delivery, only copies are returned to the calling functions, to ensure the data is not tampered after delivery.
- By the same preceding point, all packages are verified top be delivered on time.
- All required functions to retrieve package objects and there data by the aforementioned points are in place. Additionally, the produce results that are accurate and true.

## K1A. Efficiency

All lookup functions are directly affected by the total number of packages present in the master hash table. If n items exist, n items must be searched, resulting in O(n) time complexity. These lookup functions are deemed to be efficient due to their linear time complexity.

## K1B. Overhead

Increasing the data input by n packages, affects the space complexity directly by n, the additional space required is linear.

Increasing the data input by n addresses, n cities, etc. affects the space complexity in a quadratic fashion, denoted by $O(n^2)$.

Adding packages adds minimal impact, but as inputted address locations grows, the storage required is greatly affected.

## K1C. Implications

Hash table performance is not affected by adding n additional locations, or *n* additional packages, or *n* additional trucks. The amortized time complexity O(1) is still the average case. Inserts, lookups, and deletes will all still operate in constant time, and remain performant.

Additionally, adding *n* additional items from each of these categories will not affect space complexity in any way other than linearly, with the exception of the graph structure which grows by $O(n^2)$ additional space complexity. The additional space complexity for the graph is due to the relationship between all vertices being added as neighbors of every other.

However, interacting with package data is decoupled from location data, and is unimpacted.

Thus, the largest implication is that adding *n* additional locations in terms of space complexity will affect space usage for the graph object tremendously.

## K2. Other Data Structures

An AVL tree is an additional type of data structure that could have been used in place of the hash table. It works by bubbling objects with certain properities to the top, is self-balancing, and is thus, self-adjusting. A priority queue requirement exists in the solution that requires the use of sorting. Replacing the hash table with an AVL tree would add O(logn) additional time

complexity up front, for a reduction overall as packages are removed from the top of the AVL tree via the elimination for the need to sort packages by deadline. The space complexity of this leg of the solution would remain the same, O(n).

A hash table implemented with arrays vs. linked lists could have also been used in place of the specification in part D.

## K2a. Data Structure Differences

An AVL tree would be quite different from a hash table. Items are unordered in a hash table, but they are ordered in an AVL tree, by some property of the objects stored in it. Time complexities change to O(logn) for insert, and delete, with O(n) for search. Its structure is also different as nodes are organized into a tree structure, where it is ordered by parent-child node relationships. This differs greatly from the hash table specification in part D, but having this sorted, ordered access to data is beneficial, where time complexities stand to be reduced overall.

A hash table implemented with an array vs. one implemented with linked lists is different by how the buckets are implemented. An array requires computing the index of an object's location in memory multiplied by some memory address offset. A linked list on the other hand stores pointers to those same memory address, where each node stores pointer to the next or previous objects. This implementation would provide added advantage of guaranteed O(1) lookup if the load factor of the hash table becomes very large. However, removals could be impacted negatively and run in O(n) time if the delete takes place at the front of the array. Inserts would also be impacted if it were implemented poorly. An example of poor implementation would be if the array (bucket) inserts items instead of appending, this adds O(n) time complexity to the implementation because indices of arrays need updated for each item.

Overall, this second type of structure is deemed to be a worse design than the specification in part D. The data is not required to be ordered in a hash table data structure, so implementing buckets using arrays are a poor design choice, as an array would maintain insertion order in a place where it is unneeded.

## M. Professional Communication

All declarations of naming conventions of items has been clarified in-place throughout this text.

All content has organized appropriately for describing concepts in a linear fashion.

All in-text citations have been added where text by other authors is quoted, referred to, or paraphrased. build upon one another.

All works by other authors quoted, referenced, or paraphrased here have been appended as works cited in section L. using APA format.

All figures, pictures, and/or diagrams have been numbered, and labeled.

All grammar, spelling, and punctuation has been thoroughly checked.

## L. Sources - Works Cited

Auger, N., Jugé, V., Nicaud, C., & Pivoteau, C. (2018). On the Worst-Case Complexity of TimSort. *Dagstuhl Research Online Publication Server*, 1. https://drops.dagstuhl.de/opus/volltexte/2018/9467/

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms, third edition*. The MIT Press.

Dorigo, M., & Stützle, T. (2004). *Ant Colony Optimization (A Bradford Book)* (First Edition, First Printing ed.). A Bradford Book.

Jarvis, R. (1973). On the identification of the convex hull of a finite set of points in the plane. *Information Processing Letters*, *2*(1), 18–21. https://doi.org/10.1016/0020-0190(73)90020-3