Name: Ryan Kruse

Student ID: #001099380

Date: 5/26/2019

Course: Data Structures and Algorithms II

Course Instructor: Charles Lively

**WGUPS Project Essay & Discussion**

1. An algorithm that can be used to create a program to deliver the packages and meet all requirements specified in the scenario would be the Hamiltonian Cycle algorithm. This algorithm finds the shortest route to deliver all packages to their locations.
2. Core Program Overview – O(M · N!)

**Do\_Not\_Ship**

Loop through all packages – O(N²)

If package is delayed or has bad address then

Remove package from selection pool

Save address in delayed address list

Loop through all packages – O(N²)

If package address in delayed address list then

Remove package from selection pool

*The first algorithm Do\_Not\_Ship separates which packages should and should not be available for loading onto the trucks. Packages that are delayed or have a bad address become unavailable for loading. Packages that share an address with one of the delayed packages also become unavailable for loading. This is programmed such that the trucks can be loaded with as many packages that are delivered to same address at once.*

**Load\_Truck**

Loop through all packages – O(N²)

If package cannot be loaded due to truck number then

Remove package from selection pool

Loop through all packages – O(N²)

If package has a deadline then

Load package on truck

Loop through all packages – O(N²)

If package address shares address with package loaded on truck then

Load package on truck

If truck is over capacity then

Loop through all packages loaded on truck – O(N²)

Remove non-address-sharing packages

Stop if below capacity

Loop through all packages loaded on truck – O(N²)

Remove address-sharing packages

Stop if below capacity

If truck is under capacity then

Save truck state – O(K)

Loop M number of times – O(M · N!)

Reset truck state

Load truck with random packages

Load truck with shared address packages

If truck is over capacity then

Remove packages

Call Hamiltonian\_Cycle

If route is lowest miles

Save package IDs

Load truck with saved package IDs – O(N)

*The second algorithm Load\_Truck will prioritize which packages should be loaded onto the truck. It starts by selecting which packages have delivery deadlines and loads these packages first. Next, it loops M number of times and begins randomly selecting packages to load onto the truck. Each loop calls the Hamiltonian Cycle algorithm to find the lowest route miles to deliver all packages to their destinations. If the route miles is the lowest seen out of all M loops it will save the package IDs. When the loop finishes, the truck is loaded with the saved package IDs.*

**Hamiltonian\_Cycle**

Construct distance matrix – O(N²)

Construct bitmap – O(N)

Run Hamiltonian Cycle – O(N!)

If all locations visited then

If route is lowest miles then

Save total miles

Save route history

Save miles history

Loop through all locations

If location is unvisited then

Visit location

Increase total miles

Update route history

Update route miles

Call Hamiltonian\_Cycle

*The third algorithm Hamiltonian\_Cycle finds the lowest mileage route to deliver all packages to their destinations. The algorithm begins with only the HUB visited and all other locations unvisited. It then recursively visits every location from every other location, in every possible sequence, until all locations are visited. Each time the base case is reached, where all locations are visited, the algorithm checks if the route traversal has the lowest number of miles. If the route is the lowest, it saves information about the route. Information saved includes the total number of miles, the address IDs traversed in sequence, and the miles incurred for each address ID. This information is later passed onto the truck.*

The communication protocol used in this project is by storing all data in class variables and executing all algorithms in class functions. Should one class need another variable from another class, a reference is made for that class and the variable is copied from it. The variable contained in each class are useful for executing the functions located in that class. The Hub class contains loading algorithms, the Truck class contains delivery algorithms, the Simulation class contains ticking algorithms, and the Clock class contains time algorithm.

There is no target host environment used to host a server from current knowledge. The environment runs in Python 3.6, requires no external libraries or APIs, and the IDE is Pycharm. The entire simulation runs in this simulation execute function, which appears as below:

**Execute**

Setup Simulation

While True:

GUI Algorithm

Tick Algorithm

Special Algorithm

For trucks in all trucks:

Drive Truck Algorithm

Load Truck Algorithm

Deliver Truck Algorithm

This function handles all the looping events every second. When it determines all packages are delivered, the while loop ends. All simulation connections and data exchanges occurs in the algorithms that appears in this execute function. The Load Truck Algorithm will call the Hub object to load packages. The Deliver Truck algorithm will call the Truck object to deliver packages. The Tick Algorithm connects with the Clock object to increment time. The Drive Truck Algorithm will reference Settings to get the distance a truck travels each second. The GUI Algorithm will accept data exchanges with inputs from the user and disconnect upon certain inputs. The Special Algorithm will check for special events and end the simulation upon specific conditions (Matthes, 2016).

In each algorithm are lower level algorithms that handle lower level details. At a high level overview, these algorithms mentioned are the primary communication protocols to run the simulation and exchange data for each simulation tick.

Upon a growing amount of work, such as increased package count, the code has good scalability. Packages that have deadlines are loaded first, preventing resource starvation from less urgent packages. The random selector prevents bias in selecting packages for loading and also finds the most optimal package IDs to load onto each truck. As package count increases, this random selector will still work just as efficiently because it only selects 16 or less packages upon each loop. Should special event like Black Friday occur, which have multiple packages delivered to the same address, the software will ensure these packages get grouped on the same truck load. Should packages have special notes, the software can automatically parse information from these notes and adjust the package selection pool accordingly.

The software can scale as package size and package data increases. The most costly loops handling packages is O(N²), which does not take more than a few milliseconds to execute. The software can scale as truck count and drivers increases as well. Looping through each truck only costs O(N) and is primarily composed of short Boolean statements. The software cannot scale if truck storage capacity increases, as the Hamiltonian Cycle is O(N!), with N being the number of address to deliver to. If truck capacity increases, in the worst case scenario a truck gets loaded with packages that are each delivered to a different address. In this case, the Hamiltonian Cycle will have too great of a time cost to finish executing. Dynamic programming could be used to optimize this algorithm to reduce the runtime cost and make it more scalable as truck capacity increases (Lysecky and Vahid, 2018).

The python code written in this software follows industry standard formatting. Classes are divided into lower level function calls, which consists of small blocks of code that each accomplish a small task. This makes it easier to read and maintain the code. Functions and variable names are descriptive and relatable to their contents so the code is readable. Comments in the code and comments below each function also assist readers in efficiently deducing what each block of code does. For very complex blocks of code, comments that contain diagrams are used to assist the reader in understanding lower level details (Docs.python.org, 2019).

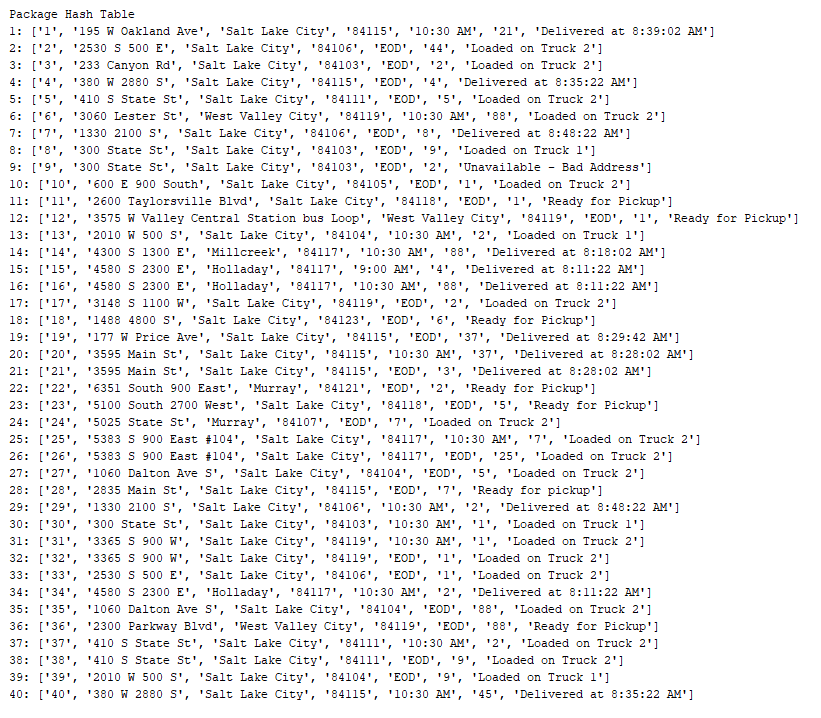
The software is efficient in that each block of code is concisely written and does not contain unnecessary variables. List comprehension is used to avoid cluttering the code with for-loops, and object oriented programming is used to avoid writing the same code twice. Class variables are used so that the code can efficiently calculate a given piece of data and store it for use later. The software is also efficient to debug, as function calls make it easy to observe line-by-line how each function changes the variables. The software is efficient to adjust for scale, as the setting panel contains all code inputs that impact the result of the program. Some algorithms can automatically parse and read special notes so that they don’t have to be identified manually (Docs.python.org, 2019).

There are self-adjusting data structures located throughout the software. The hash table contains a block of code that will double the hash table size should load factor exceed a threshold. The resizing takes O(N). The random package selector also self-adjusts given the number of urgent packages selected. As the count of urgent packages increases, the number of packages randomly selected decreases. The truck capacity is self-adjusting in adapting to overloading. The truck can unload packages beginning from lowest priority to highest priority packages until no longer overloaded. The hub is a self-adjusting data structure when determining which packages are available in the selection pool. A flight delay will remove the package IDs from the selection pool. A flight arrival will add the package IDs to the selection pool. These adaptations do not impact run time beyond several milliseconds. The time complexity cost is O(N) or O(N²) for these additional functionalities (Lysecky and Vahid, 2018).

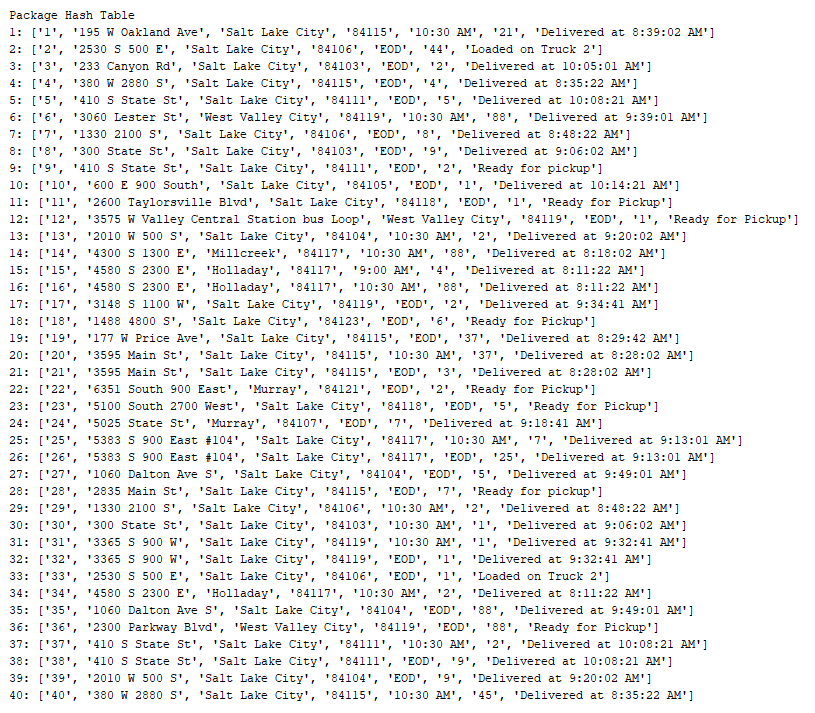
The self-adjusting data structure that has the highest time complexity cost is the Hamiltonian Cycle, which is O(N!). This cycle has a bitmap and distance submatrix that can self-adjust to the number of unique addresses being delivered to. This algorithms works optimally for delivering 16 packages just as well as it could for delivering 2 packages. The algorithm is self-adjusting as package count increases or decreases. It will have higher time and space cost should count increase however. Dynamic programming could be used to reduce recursive calculations and terminate recursive cycles early (Miller and Ranum, 2011).

1. Original code is attached to the project files.
2. A data structure that can be used to store package data is a hash table with a linear rehash function and resize functionality should load size exceeds 70%.   
     
   The relationship between the data point and the data being stored is similar to a dictionary relationship. The package ID gets inputted into the hash table as a key. The key then gets transformed by the hash function into a hash value. The slot number is checked to find an empty slot, or a slot identical to the key. Upon a collision the key can get rehashed linearly until a valid slot is found. When a slot is found, a data list corresponding to the slot index can have its index access. This index contains the data being stored for the respective index slot. The data can then be returned to the function that called the hash table. The slots being stored in this project are package IDs and the data being stored is the package data (Miller and Ranum, 2011).  
     
   The hash table class functions of \_\_len\_\_, \_\_contains\_\_, \_\_getitem\_\_, \_\_setitem\_\_, and \_\_str\_\_ make it easier to call the functionalities of hash table in python code. A simple line like hashtable[30] = [1, 2, 3] is identical to python dictionary calls. It is also more readable than hashtable.put(30, [1, 2, 3]).
3. Hash table is included in original code in the hash table class.
4. Look-up function are included in original code under the simulation class GUI functions.
5. The interface is included in original code under the simulation class \_\_str\_\_ function and GUI functions.

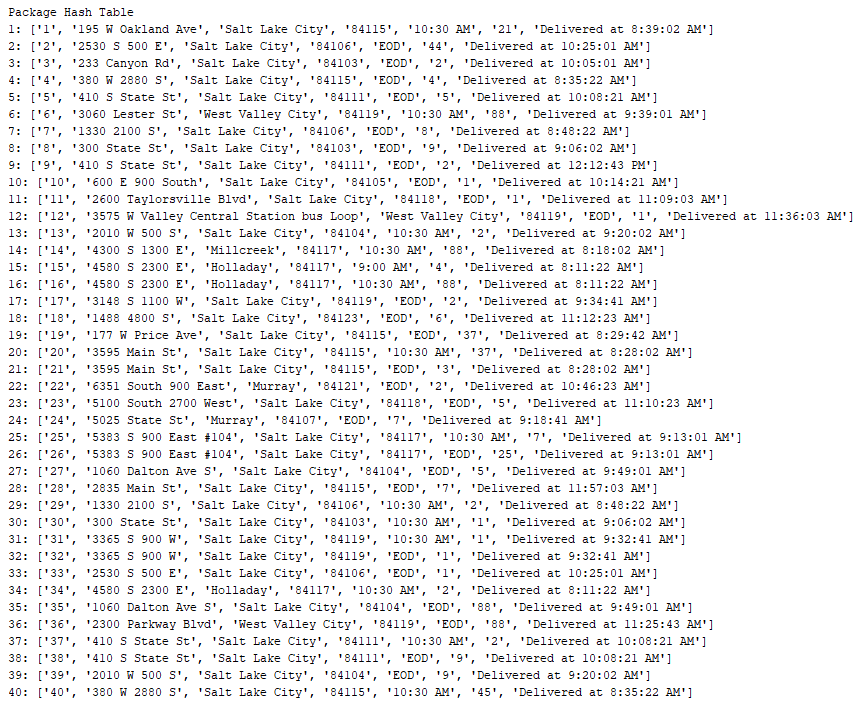
**First Status Check at 9:05:00 AM**



**Second Status Check at 10:20:00 AM**



**Third Status Check at 12:12:43 PM**



1. A screenshot of complete code execution can be found in the screenshot folder.
2. There are several strengths to the software’s chosen algorithm. It recursively checks every possible way to traverse all locations and guarantees to find the lowest mileage route. It not only finds the lowest mileage route but also saves the route path and route weights. This information can be passed onto the truck for calculating the number of miles driven for each delivery. Lastly, the algorithm works on a subset matrix rather than the full matrix, which avoids the cost of doing calculations on irrelevant data.  
     
   The algorithm is verified to make the deliveries on time and meet the requirements given the scenario. There are two other algorithms that can be used to meet the requirements given the scenario. The greedy algorithm and the random algorithm. An explanation of both algorithms is given below.  
     
   The greedy algorithm will travel to the next location that has the lowest miles to travel to. This loops until all locations are visited. Compared to the Hamiltonian Cycle algorithm, the greedy algorithm is optimal for the first several locations visited. However, it has no foresight for traveling to future locations. The greedy algorithm may discover that the last locations take 10+ miles to travel to each. The greedy algorithm is not optimal in the long run given the potentially high miles to travel to the last locations (Lysecky and Vahid, 2018).  
     
   The random algorithm will randomly select an unvisited location to visit. This loops until all locations are visited. Compared to the Hamiltonian Cycle algorithm, the random algorithm is optimal due it its low time-space complexity. The random algorithm can loop several thousand times to find best set in a very short amount of time. The lowest mileage route to travel to all locations is considered the best set. The random algorithm is similar to the Hamiltonian Cycle algorithm except it is O(M · N) with M being the number of times looped and N being the number of unique addresses (Miller and Ranum, 2011).
3. If I could do this project again, I would like to experiment with handling the bad package address in a more automated fashion. The simulation currently handles a bad package address by removing the package from the package selection pool. In an actual simulation, this information would not be known beforehand until a delivery is actually made (in which case the package would be rejected). I would rewrite the Truck delivery functions so that if a package is delivered and rejected due to a bad address, it remains on the truck. When the truck returns to the Hub, the package can then be removed from the truck and package selection pool. When the address is fixed, the package can then be added back to the package selection pool.  
     
   Another feature I would like to add is having the truck dynamically change its path when delivering packages. Say a truck has packages changing addresses mid-delivery. It would be costly to return these packages back to the Hub, restart the Hamiltonian Cycle, and find a new delivery route. The truck delivery function could have a Hamiltonian Cycle algorithm built into it so that if a package address does get updated mid-delivery, the Hamiltonian Cycle algorithm can reconstruct a new truck route without returning to the Hub.
4. All criteria’s have been met. The data structures used are specialized classes. Each class contain class variables and class functions that are specific to the functionality of that particular class. The data being used is the data that is most relevant to that class. A Truck class will contain data about the packages, routes, and miles between locations. A Hub class will contain data about recursive variables, the package selection pool, and unavailable packages. A clock class will contain time variables such as hours, minutes, and seconds. This is efficient because classes contain only the data it needs (Matthes, 2016).  
     
   Overhead is covered in the code execution overview. Big O notations are included for each block of code. The computational time is around 10 seconds for the entire program. The slowest calculations are done in the O(M · N!) random package selector plus Hamiltonian Cycle algorithm. There is a significant amount of memory used in the Hamiltonian Cycle algorithm. It not only needs to keep track of the lowest mileage times but of all the lists that are stored in memory as it traverses through N! nodes. This has a high memory cost and the memory is only freed up when the algorithm completes. To my current knowledge there is no bandwidth aspects to this code as there is no networking functionary required. The only bandwidth that would occur would be downloading the code and downloading the .csv files to execute the simulation (Miller and Ranum, 2011).

Scale is covered in the code execution overview. When the number of packages increases, the cost is minimal as O(N²) only takes a few milliseconds to execute. Unless there are hundreds of thousands of packages to sort, it is unlikely that an increase in package count will impact the execution time of the software. As number of trucks increases, the cost is minimal as it is O(N), with N being the number of Trucks. Trucks can only hold up to 16 packages so the number of times a Truck needs to loop through its packages is capped at the constant of 16. Trucks also spend most of the simulation driving. Executing each truck for every simulation tick is composed of quick Boolean statements that take fractions of a millisecond to execute. As the number of cities increases, this algorithm can scale to deliver to any location in any city O(N²). The software indifferent between the layout of each city or the number of addresses it is delivering to. As long as the distance matrix contains the miles to travel from every location to every location, the algorithm will scale without issues.

A data structure that can be used to meet the requirements given the scenario is to eliminate the classes and to have instead hundreds of functions that have input and return parameters. This structuring of the simulation is not optimal as there are dozens of variables being updated, stored, and passed. Functions would have over a dozen input statements and it would be difficult to rename variables and update the simulation without breaking other functions. If the simulation was simpler and the scale was smaller, then this structuring would be optimal. Global variables can be used also to make it easier to keep track of certain variables, but for long term use and scale this is not good practice (Matthes, 2016).

Another data structure that can be used to meet the requirements is to build a graph class and a vertex class to contain the locations, and establish links between locations so that a graph traversal algorithm can be used on it. The attributes of this data structure is that it consumes more memory, and a significant number of objects must be created and kept track of. Additionally, algorithms that are written to traverse the graph must keep in mind special conditions such as deadlines, which is more difficult to implement. The graph and vertex data structure is difficult to debug, as debugging the ADT is best for examining one object at a time rather than 27 objects. Additionally, it is not possible to create a subset graph without reconstructing another graph itself, which consumes additional time and memory. With 27 vertexes and hundreds of links between each vertex, this data structure was abandoned for the subset distance matrix (Lysecky and Vahid, 2018).

Work Cited

1. Lysecky, R. and Vahid, F. (2019). C950: Data Structures and Algorithms II. [online] ZyBooks.com. Available at: https://learn.zybooks.com/zybook/WGUC950AY20182019 [Accessed 20 May 2019].
2. Lysecky, R. and Vahid, F. (2018). C949: Data Structures and Algorithms I. [online] ZyBooks.com. Available at: https://learn.zybooks.com/zybook/WGUC9492018 [Accessed 16 May 2019].
3. Matthes, E. (2016). Python Crash Course. 1st ed. San Francisco: No Starch Press, Incorporated, pp.118 - 214.
4. Miller, B. and Ranum, D. (2011). Problem solving with algorithms and data structures using Python. 2nd ed. Sherwood: Franklin, Beedle & Associates, Incorporated, pp.1 - 80, 149 - 230.
5. Docs.python.org. (2019). 3.7.3 Documentation. [online] Available at: https://docs.python.org/ [Accessed 18 May 2019].
6. Stack Overflow. (2019). Stack Overflow - Where Developers Learn, Share, & Build Careers. [online] Available at: https://stackoverflow.com/ [Accessed 16 May 2019].