C950 WGUPS Algorithm Overview

Joshua gibson

ID: 002204262

WGU Email: jgib194@wgu.edu

6/5/2023

C950 Data Structures and Algorithms II

A. Algorithm Identification

This project uses the “Greedy algorithm” to deliver the packages

B1. Logic Comments

The base function of this greedy algorithm is to search through a list of distances and find the smallest distance between a starting package and a potential next package. The next package is determined only if it has the shortest distance from the previous package. The rudimentary code is as follows:

```
While x < truck package list length
    For all packages in list
        If length of package list is equal to 1
            Next  package is “HUB”
        Otherwise next package = current package address

    Compare packages
    If new package start is not same and distance is less than the current
        Change smallest distance to current distance

Update the next package delivery time
```

Big O for the greedy algorithm is O(n).

Hash table:

```
Initialize hash table
Insert using key and package object
    Modulo package id with table length to find bucket
    Create key value pair using package id and package object respectively
    If key is present in package list update package
        Else add package to end of list.
```

Big O for the hash table is O(n)

Read in packages:

```
Load package file
For each entry in file
    Create new package with associated data
    Insert each package into the hash table
```

Big O for this section is O(n)

Load distances:

Load distance file
For each row in file
Add data to a distance table

Big O for this section is 0(n)

Delivery routine:
For each truck
Loop through each truck list
Update package status

Big O for this section is O(n)

Display routine
While loop to refresh options to users
Option 1: final details of package
Loop through packages and update status using a time that indicates all packages were delivered
Print all packages
Option 2: search a package status
Insert package id
Insert time parameter
Update package status
Print package
Option 3: Print package information based on time
Input for time
Update packages based on time
Loop through packages
Print package information
Option 4: Exit
Break from loop

This section contains a while loop with embedded if statements making this section of code O(n^2).

B2. Development Environment

The programming environment used to make this application was PyCharm Community Edition 2022.3. The was developed on an Alienware 17 R5 that contains 16 GB of installed RAM and an Itel(R) Core(™) i7-8750H CPU @ 2.20GHz.

B3. Space-Time and Big-O

The most complex items utilized throughout the entire project are for loops and if statements. From inserting and searching the hash table, to the greedy algorithm, to the code written to display the information requested. Therefore the Big O for this project is O(n).

B4. Scalability and Adaptability

The hash table provides an excellent structure for future scaling. More packages can be added into the hash table quickly and efficiently even if the package volume increases significantly. Furthermore, the algorithm could become even more efficient with larger volumes of packages by increasing the number of “buckets” allowing for faster searches.

B5. Software Efficiency and Maintainability

The software is efficient and easily maintained because with minimal changes (mostly the number of “buckets” for the hash table) it can accommodate an increased number of packages.

B6. Self-Adjusting Data Structures

The primary advantage of a hash table is that it provides quick access to objects contained within. However, a disadvantage is that as entries into the hash table increase there could be potential collisions based on overlapping keys. However, that could potentially be averted by increasing the number of “buckets” which may or may not impact search efficiency based on the number of items contained in the hash table.

C. Original Code

```
#Greedy algorithm
def greed_algorithm(truck):
   #initialize variables
   total = 0
   return_distance = 0
   start = "HUB"
   next = None
   smallest = 20.0
   new_start = None
   x = 0
   delivered_id = None

   #cycle through each package in the trucks package list
   while x < len(truck.package_list):

       for package_id in truck.package_list:
           package = hashed_packages.search(package_id)
```

```
            # if only one package left return to hub after delivery
            if len(truck.package_list) == 1:
                next = "HUB"
            else:
                next = package.address

            #find the distance between the last package and the next package
            distance = float(retrieve_distance(start, next))
            #if last package calculate distance back to hub
            if len(truck.package_list) == 1:
                smallest = float(retrieve_distance(start, next))
                delivered_id = package.id_num
            #else calculate distance to next package
            else:
                if package.address != start and distance < smallest:
                    delivered_id = package.id_num
                    new_start = next
                    smallest = distance
                    package_object_id = package.id_num

        total = total + smallest

        #update package
        update_package = hashed_packages.search(delivered_id)
        min = total/truck.average_speed * 60
        time = convert_to_time(truck.starting_time)
        delivered_time = time + timedelta(minutes=min)
        update_package.change_delivered(update_package, smallest,
       convert_to_time(delivered_time.strftime("%H:%M %p")))

        hashed_packages.insert(update_package.id_num, update_package)
        #print(hashed_packages.search(delivered_id).delivery_time)
        start = new_start

        #if one item left delete list
        if len(truck.package_list) == 1:
            del truck.package_list[-1]
        #otherwise delete the package object from the truck list
        else:
            truck.package_list.remove(package_object_id)
            smallest = 30

    return total
```

C2. Process and Flow Comments

The program loads package information into a hash table and loads distance information between various locations from another file. Packages are “loaded” onto each truck within the program logic based on priority. Truck 1 and 2 are ran through the greedy algorithm and then the one package that needs to be updated due to information changing as well as truck 3’s starting time based on when truck 1 finishes are updated. A function to simulate package delivery based on an input time was created. The command prompt is displayed. If option 1 is pressed, it will show all packages in their final state. If option 2 is selected a package can be selected and a desired time and then the information is ran through the delivery function and the packages status at that time is displayed. If option 3 is selected the user will be asked for a time and then the program will display the information and status of the packages at that time. Option 4 will exit the program

D. Data Structure

A linear search hash table was utilized as the data structure for this project.

D1. Explanation of Data Structure

The packages are sorted into the buckets of the hash table using a packages id number. Each package number is a unique base 10 number. The hash table utilizes an initial capacity of 10 buckets, correlating with the base 10 aspect of the package id. Collisions are therefore eliminated because each package will be sorted into a category of 0 through 9. No package id number is utilized more than once, therefore no collisions.

E. Hash Table

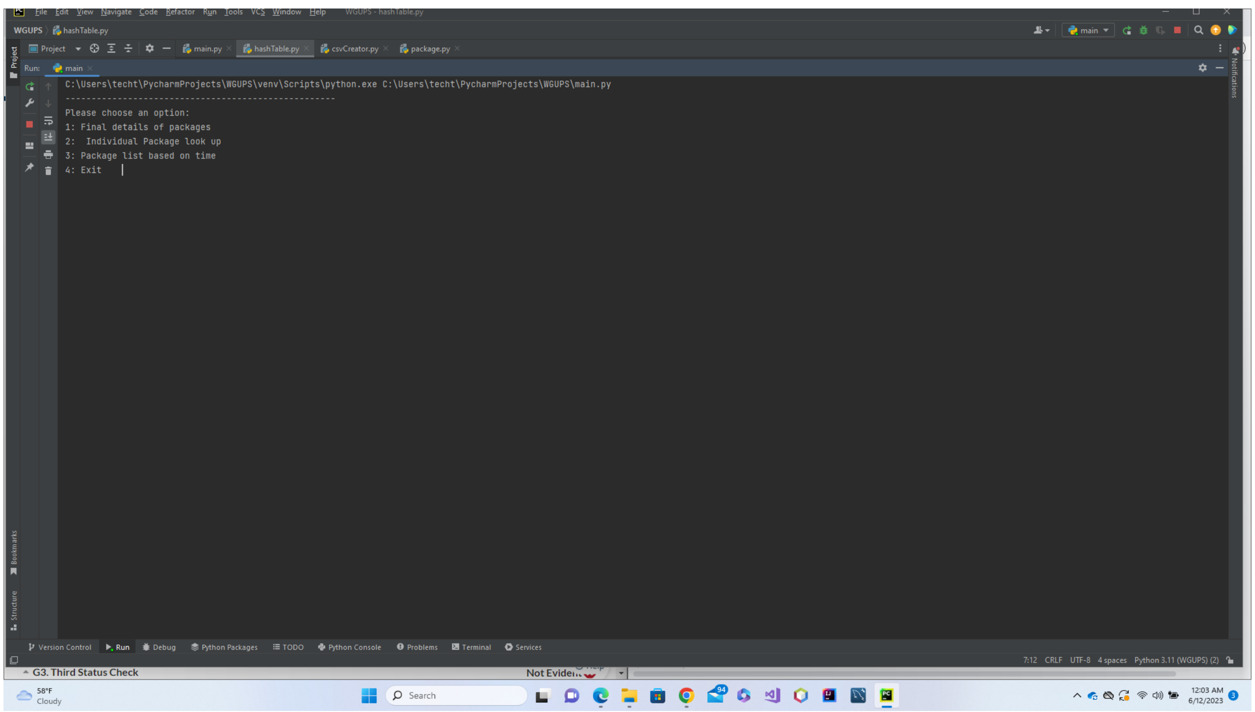
```
class HashTable:
   #initializor for the hash table object
   def __init__(self, initial_capacity = 10):
       self.table = []
       for i in range(initial_capacity):
           self.table.append([])

   #function to insert a package into the hash table
   def insert(self, key, package):
       bucket = hash(package.id_num) % len(self.table)
       bucket_list = self.table[bucket]

       key_value = [key, package]

       #if key is present in list update associated package
       for value in bucket_list:
           if value[0] == key:
               value[1] = package
               return True

       #if not, insert package to end of bucket list
       bucket_list.append(key_value)
       return True


   #remove a package from the hash table
   def remove(self, key):
       bucket = hash(key) % len(self.table)
       bucket_list = self.table[bucket]

       #if key found in bucket list remove the key and associated package
       if key in bucket_list:
```

```
            bucket_list.remove(key)
```

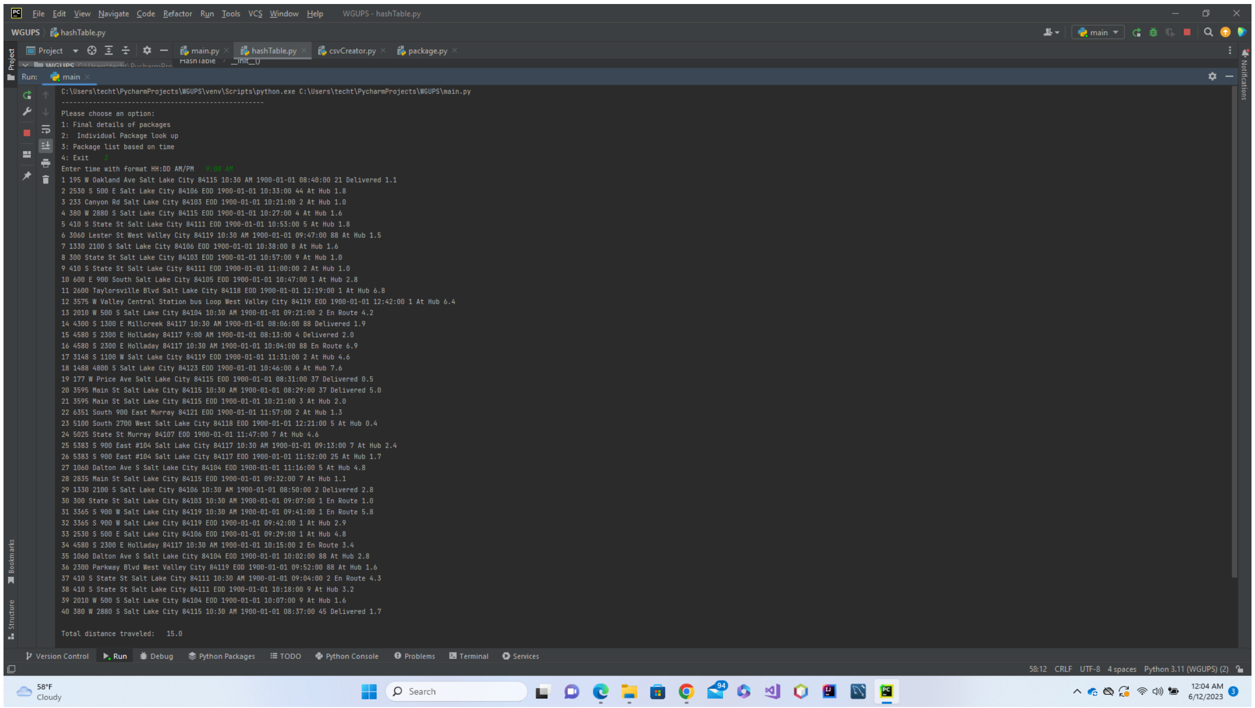
F. Look-Up Function

```
#search for a package in the hash table
   def search(self, key):
       bucket = hash(key) % len(self.table)
       bucket_list = self.table[bucket]

       #if key is found in bucket list return associated package
       for value in bucket_list:
           if(value[0] == key):
               return value[1]

       #no key value found in bucket list so return nothing
       else:
           return None
```

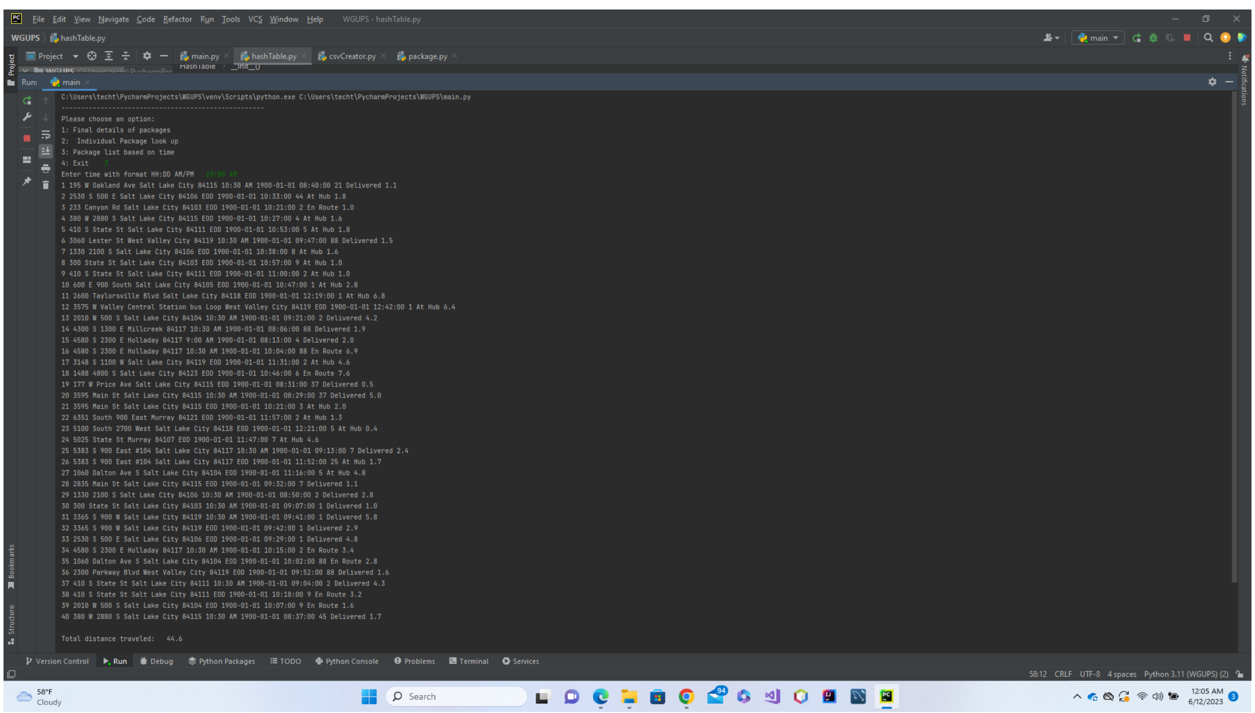
G. Interface



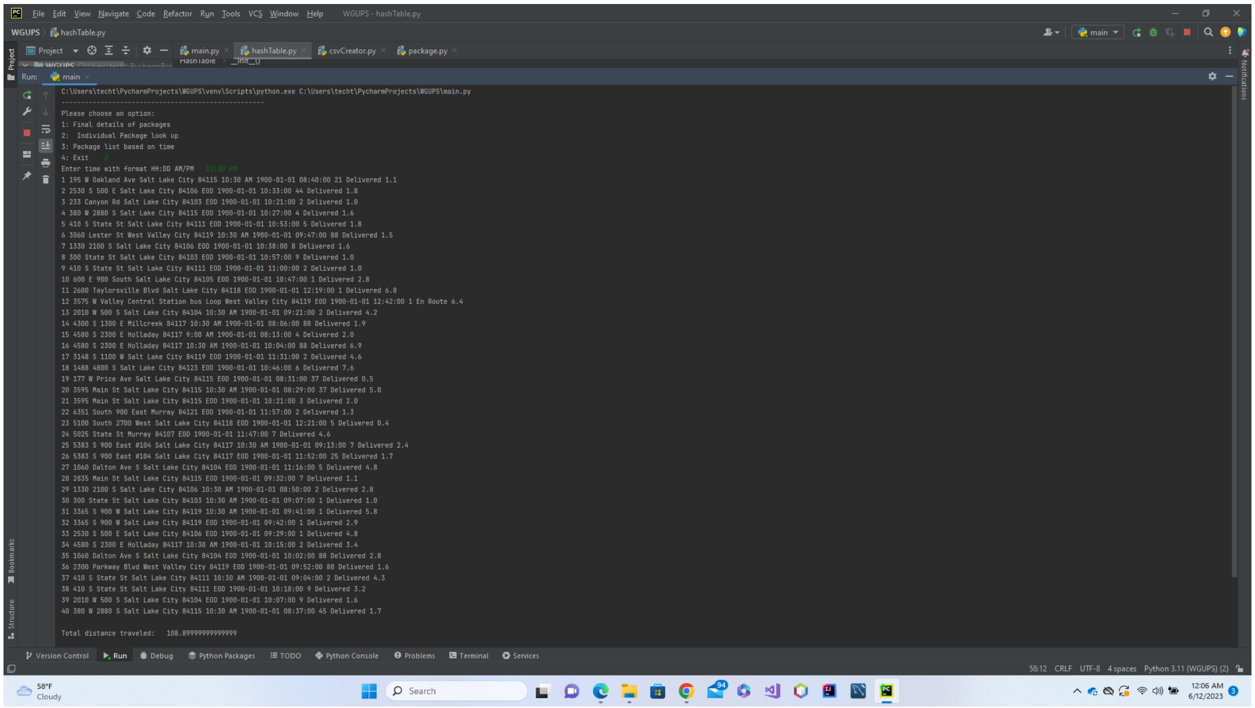
G1. First Status Check



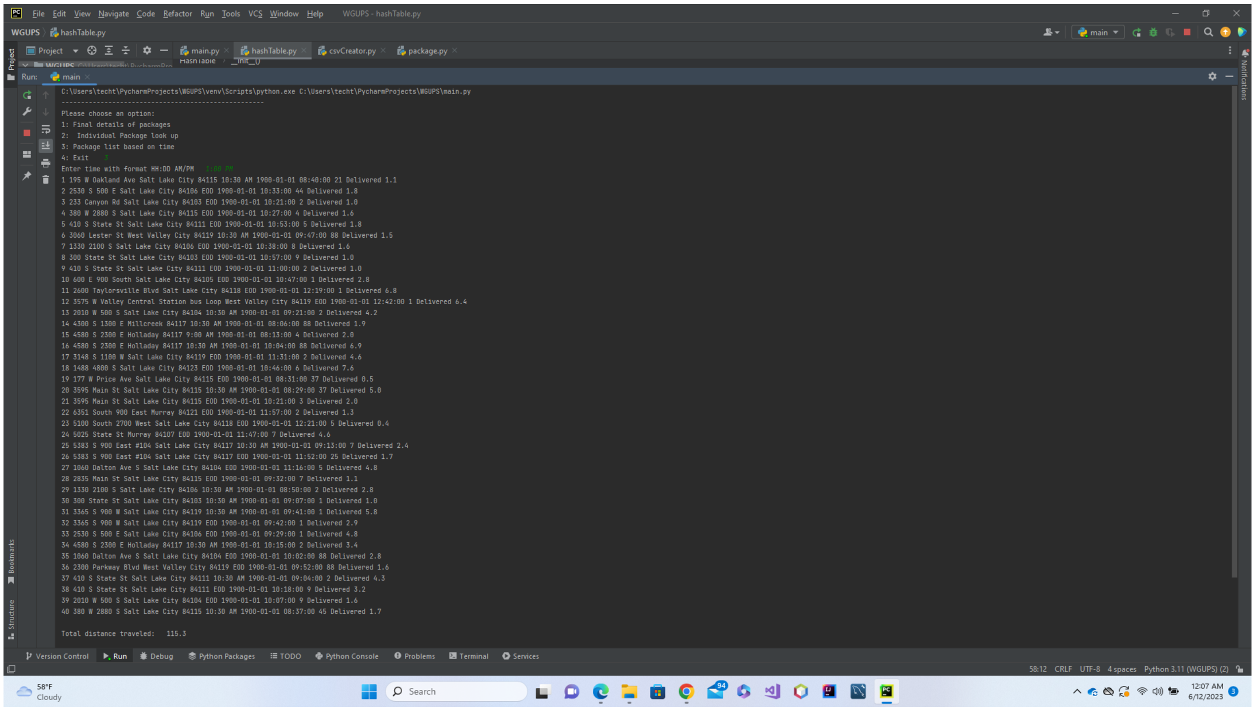
G2. Second Status Check



G3. Third Status Check



H. Screenshots of Code Execution



I1. Strengths of Chosen Algorithm

One strength of the greedy algorithm is that it was easy to implement. The programming portion of it is relatively simple and small. Another strength is that it is efficient and can find the "greediest" choice within a set of data very quickly.

I2. Verification of Algorithm

The greedy algorithm implemented in this project delivered all the packages and kept the total drive distance under 140 miles.

I3. Other Possible Algorithm

Another algorithm that could have been implemented is Dijkstra's shortest path algorithm. This algorithm compares distances between vertices to find the shortest path in order to meet each graph. It is more complicated, though, because more information is needed as to the relationships between the different vertices. Whereas, the greedy algorithm only searches for the smallest option available. Dijkstra's algorithm could provide a more optimal solution.

A second algorithm that could have been used instead is the Nearest Neighbor algorithm. This algorithm sets all data elements to unvisited and then randomly selects one of them as a starting point. It then proceeds to "visit" each data element changing it's status to visited and goes through all data elements until all have been visited.

J. Different Approach

If I were to do this project again I would probably try to incorporate elements into the greedy algorithm that would strive to obtain a more optimal solution.

K1. Verification of Data Structure

The hash table data structure provides a quick and easy interface to store package data. One piece of information, the package number, is used to quickly identify a package within the system and all the information that is associated with that package. It also provides the foundation for changes to be made quickly while only changing a small amount of code to accommodate a larger volume of packages that are being delivered. For example, if the bucket lists start to become too large, a simple change in the starting number of buckets can spread the data out more. Furthermore, if other cities were added a second hash table could be quickly created and added to where each bucket of that hash table was designated to a particular city. The number of trucks wouldn't really have an impact on the hash table because the hash table only contains information specific to each individual package. (Lysecky & Vahid, 2023)

K2. Other Data Structures

A binary tree could have been used as the data structure for this program. This data structure consists of nodes that can each have two “children” nodes that are associated with it. The top of the tree consists of a root node that can have up to two children nodes pointing to each, and each child node can have up to two different children nodes, thus becoming a parent node. A big difference between this function and the greedy algorithm implemented is the logic needed in order to complete a search in a binary tree as well as the amount of time to search through the binary tree versus the greedy algorithm. Another issue is that in order to optimize searches you would have to implement a tree that is balanced to where the data elements are spread evenly across the tree. (Lysecky & Vahid, 2023)

A linked list could have also been implemented as a data structure. This type of data structure consists of a chain of nodes linked together in a row. A start node points to the next node and this continues until all nodes are linked together in a long line. Nodes can be easily moved around and rearranged by simply changing what the nodes are pointing to. However, a search in a link list requires the program to visit each node along the list until it finds the particular node it is looking for. So if the desired node is at the end of the list then all nodes will have been searched. This can become particularly cumbersome and problematic as the list becomes larger and larger. (tutorialspoint)

L. Sources - Works Cited

Lysecky, R., & Vahid, F. (2018, June). C950: Data Structures and Algorithms II. zyBooks. Retrieved June 5, 2023, from https://learn.zybooks.com/zybook/WGUC950AY20182019?modal_name=about-zybook

tutorialspoint. (n.d.). *Data Structure and algorithms - linked list*. Learning Data Structure: organizing data. https://www.tutorialspoint.com/data_structures_algorithms/linked_list_algorithms.htm#