Adam Scott

1/30/2023

C950 Data Structures and Algorithms 2 Write Up

**Pseudocode**

**Main.py**

**Def Load Package Info**

* function load\_Package\_Info(package\_Data)
* open the file "package\_Data" and read its content as CSV
* for each row in the CSV content do the following:
  + extract package information from the row:
    - pID = the first value in the row as integer
    - pAddress = the second value in the row
    - pCity = the third value in the row
    - pState = the fourth value in the row
    - pZip = the fifth value in the row
    - pDeadline = the sixth value in the row
    - pWeight = the seventh value in the row
    - pStatus = "At Hub"
  + create a package object with the extracted information:
    - package = Packages.Packages(pID, pAddress, pCity, pState, pZip, pDeadline, pWeight, pStatus)
  + insert the package object into a hash map using pID as the key:
    - HashMap.my\_Hash.insert(pID, package)
* end for
* end function
* load\_Package\_Info("Package\_Data.csv")

**Def address Lookup**

* function address\_Lookup(address)
  + open the file "Address\_Data.csv" and read its content as CSV
  + convert the CSV content into a list of rows
  + for each row in the list do the following:
    - if the given "address" is found in the second column of the current "row", then:
      * return the index of the current "row"

**Def Distance**

* function Distance(address\_X, address\_Y)
  + open the file "Distance\_Data.csv" and read its content as CSV
  + convert the CSV content into a list of rows
  + retrieve the distance between "address\_X" and "address\_Y" from the list:
    - distance\_Between = distances[address\_X][address\_Y]
  + if the retrieved distance is an empty string, meaning there is no distance data available for this pair of addresses, then:
    - retrieve the distance between "address\_Y" and "address\_X":
      * distance\_Between = distances[address\_Y][address\_X]
  + return the distance as a floating-point number:
    - return float(distance\_Between)

**Greedy Algorithm**

* function greedy\_algorithm(start\_address, start\_Packages)
* current\_address = start\_address
* current\_Packages = start\_Packages
* visited\_Packages = []
* total\_Distance = 0.0
* while there are still packages left to deliver (i.e., len(current\_Packages) > 0), do the following:
  + find the package that is closest to the current address:
    - next\_Package = HashMap.my\_Hash.search(current\_Packages[0])
    - next\_distance = Distance(address\_Lookup(current\_address), address\_Lookup(next\_Package.address))
    - for each remaining package\_ID in current\_Packages[1:], do the following:
    - current\_Package = HashMap.my\_Hash.search(package\_ID)
    - current\_distance = Distance(address\_Lookup(current\_address), address\_Lookup(current\_Package.address))
    - if the current package is closer to the current address than the previously found package, then:
    - next\_Package = current\_Package
    - next\_distance = current\_distance
    - if the current package is package #25, then:
    - next\_Package = current\_Package
    - next\_distance = current\_distance
    - break out of the loop
  + update the current address and the list of visited packages:
    - current\_address = next\_Package.address
    - visited\_Packages.append(next\_Package.id\_Num)
    - current\_Packages.remove(next\_Package.id\_Num)
    - total\_Distance += next\_distance
* return the list of visited package IDs and the total distance traveled:
  + - return visited\_Packages, total\_Distance

Top of FormBottom of Form

**Def update Time**

* function update\_Time(truck):
  + for each package\_ID in truck.packages\_In\_Truck:
    - departure\_Package = HashMap.my\_Hash.search(package\_ID)
    - departure\_Package.Departure\_Time = truck.departure\_Time
    - departure\_Package.truck\_Num = truck.truck\_Num
  + delivery\_Time = truck.departure\_Time
  + for each package in truck.packages\_In\_Truck:
    - current\_package = HashMap.my\_Hash.search(package)
    - distance = Distance(address\_Lookup(truck.delivery\_Address), address\_Lookup(current\_package.address))
    - delivery\_Time += datetime.timedelta(hours=distance / truck.max\_Speed)
    - current\_package.Delivery\_Time = delivery\_Time
    - truck.delivery\_Address = current\_package.address

**Def Status**

* function Status():
  + text = input("Would you like to view a specific package or all packages? (Type 'One' or 'All'): ")
  + if text == "One":
  + try:
    - package\_Num = input("Please enter the ID number of the package you want to view: ")
    - selected\_Package = HashMap.my\_Hash.search(int(package\_Num))
    - time = input("Please Enter A Time In The format 'HH:MM': ")
    - (h, m) = time.split(":")
    - convert\_Time = datetime.timedelta(hours=int(h), minutes=int(m))
    - package = HashMap.my\_Hash.search(int(package\_Num))
    - package.update\_Package\_Status(convert\_Time)
    - print(str(selected\_Package))
    - text = input("If you would like to view the status of another package type 'Yes' or 'No': ")
  + if text == "Yes":
    - Status()
  + else:
    - print("Goodbye")
    - exit()
  + except ValueError:
  + print("Invalid Entry, please try again")
  + Status()
  + else if text == "All":
    - time = input("Please Enter A Time In The format 'HH:MM': ")
    - (h, m) = time.split(":")
    - convert\_Time = datetime.timedelta(hours=int(h), minutes=int(m))
    - for i in range(1, 41):
      * package = HashMap.my\_Hash.search(int(i))
      * package.update\_Package\_Status(convert\_Time)
      * print("Package: {}".format(HashMap.my\_Hash.search(i)))
      * text = input("If you would like to view the status of another package type 'Yes' or 'No': ")
  + if text == "Yes":
    - Status()
  + elif text == "No":
    - print("Goodbye")
    - exit()
  + else:
    - print("Invalid entry, please try again")
    - Status()

**Def Main**

* function Main():
  + print "To view the current total mileage type 'Mileage'"
  + print "To view package status type 'Status'"
  + text = input("Which would you like to view?: ")
  + if text equals "Mileage" then
    - Mileage() function call
  + else if text equals "Status" then
    - Status() function call
  + else
    - print "Invalid entry, please try again"
    - Main() function call

**Class Main**

* class MainClass:
  + print "Welcome to the WGU Mail Carrier Service"
  + Main() function call

**Hashmap.py**

* Create a Class called ChainingHashTable
  + Initialize a list with size of "starting\_capacity" (10 by default)
    - set self.list to an empty list
    - repeat from i = 0 to i < starting\_capacity:
      * append an empty list to self.list
* Define function insert(key, value):
  + bucket = result of hash(key) modulo len(self.list)
  + set bucket\_array to the bucket-th item in self.list
  + for key\_value in bucket\_array:
    - if key\_value[0] equals key:
      * set key\_value[1] to value
      * return True
  + append [key, value] to bucket\_array
* Define function search(key):
  + bucket = result of hash(key) modulo len(self.list)
  + set bucket\_array to the bucket item in self.list
  + for match in bucket\_array:
    - if key equals match[0]:
      * return match[1]
* Define function delete(key):
  + Empty\_Value = result of hash(key) modulo len(self.list)
  + set Address to the Empty\_Value-th item in self.list
  + if key is in Address:
    - remove key from Address

**Packages.py**

* Create a class "Packages" with attributes:
  + ID\_Num
  + Address
  + City
  + State
  + Zip
  + Deadline
  + Weight
  + Delivery\_Status
  + Departure\_Time
  + Delivery\_Time
* Implement the **str** method to return a formatted string representation of the object.
* Implement the update\_Package\_Status method to update the delivery status of the package based on the input time "change\_Status".
  + If Delivery\_Time is less than change\_Status, set the status to "Delivered"
  + If Departure\_Time is greater than change\_Status, set the status to "En Route"
  + Otherwise, set the status to "At The Hub"
* Input: convert\_timedelta
* If Delivery\_Time < convert\_timedelta, set Status to "Delivered"
* Else If Departure\_Time > convert\_timedelta, set Status to "En Route"
* Else, set Status to "At The Hub"
* Output: change\_Status

**Trucks.py**

* Class Trucks: - Create a constructor with parameters: Packages, Mileage, Address, Depart\_Time, Capacity, Speed, Load
* Initialize instance variables: Packages, Mileage, Address, Depart\_Time, Capacity, Speed, Load Set Time equal to Depart\_Time
* Create a method **str** - Return a formatted string that includes the values of all instance variables: Packages, Mileage, Address, Depart\_Time, Capacity, Speed, Load

**Programming Environment**

Version of Python – Version 3.9

Version of Operating System – Windows 10 Pro x64

Hardware Description: CPU – Ryzen 7 2700X 8 Core Processor

GPU – Nvidia RTX 2060 Super

Memory – 16GB RAM

**Space-Time Complexity**

**Space-Time complexity of the elements in the main file, Main.py, are as follows:**

**def load\_Package\_Info:** This code has a space-time complexity of O(N), where n is the number of rows in the input CSV file. This is because the code reads the file line by line using a for loop and performs operations for each line of the file. The operations performed for each line (creating an object called package\_Data and inserting it into a hash table) are constant time operations, and their time complexity does not depend on the size of the input data. As a result, the overall time complexity is determined by the number of rows in the file, and it increases linearly with the size of the input. The space complexity is proportional to the number of packages being processed which in this case, is 40.

**def Distance ():** This block of code if referring to the calculation of distances between two addresses. The first being the current address and the next being the address of the next delivery. The space-time complexity of this block is O(1) since it involves only a constant number of operations, regardless of the size of the input data. The space complexity of this code is constant since it is only storing variable values.

**Def address\_Lookup ():** This block of code is referring to the programs ability to specifically gather all addresses in the given CSV files. The space-time complexity of this code is O(N), where n is the number of rows in the CSV\_Address\_Data list. This is because the code performs a linear search over the list, and the running time grows linearly with the size of the input data. The space complexity of this code is constant since it is only storing variable values.

**Truck Objects:** These three pieces of code refer to the creation of the three required truck objects. These simply assign variables to specific attributes of code pertaining to each truck using an array. The space-time complexity of each of these is O(1) since the processing time is constant regardless of input size. The space complexity of this code is also constant since it is only storing array data.

**Def update\_Time():** The time complexity of this code is O(N), where N is the number of packages in the truck. The code consists of a loop that iterates over the packages in the truck, which has a time complexity of O(N). The loop updates the departure time for each package and calculates the delivery time for each package based on the truck's departure time, delivery address, maximum speed, and distance to the package's address. Since the loop iterates N times, the total time complexity of the code is O(N).

**Def greedy\_Algorithm():** The time complexity of this code is O(N^2), where N is the number of packages. The code consists of a while loop that iterates over the current packages until there are no more packages left. Within this loop, there is a nested loop that iterates over the current packages to find the next closest package to the current location. The nested loop has a time complexity of O(N) for each iteration of the outer loop. Since the outer loop runs N times, the total time complexity is O(N^2). The space complexity of this code is O(N), where N is the number of packages. The code uses several variables to store the current state, such as current\_address, current\_Packages, visited\_Packages, total\_Distance, and next\_Package. The size of these variables depends on the size of the input data and the number of packages, and since the code stores the visited packages in a list, the space complexity is linear with respect to the number of packages.

**Def Status():** The time complexity of this code is O(n), where n is the number of packages in the HashMap. The program loops through each package and updates its status based on user input, so the time complexity scales linearly with the number of packages.

**def Main:** This is the main class in the program and a part of the User interface. The space-time complexity of this code is O(1), where N is the number of packages in the chaining hash table. This is because the code performs a loop through all the packages to update their status, and the time it takes to update each package is constant, regardless of the number of packages. Space complexity is constant and always uses the same amount of memory regardless of input.

-------------------------------------------------------------------------------------------------------------------------------

**Space-Time complexity of the elements within the HashMap.py file are as follows:**

**Class ChainingHashTable:** For the “**init**” method, the space-time complexity is O(N), where n is the starting capacity. This is because the code performs a loop that appends an empty list to the "self.list" array n times, so the time it takes is directly proportional to the value of N.

For the insert method, the space-time complexity is O(N), where n is the number of elements in the bucket of the hash table corresponding to the key being inserted. This is because the code performs a loop that iterates through the elements in the bucket to check if the key already exists, and if it does, it updates its value. The loop takes O(N) time, and since it is performed twice in the code.

For the search method, the space-time complexity is O(N), where n is the number of elements in the bucket of the hash table corresponding to the key being searched for. This is because the code performs a loop that iterates through the elements in the bucket to find a match with the key. The loop takes O(N) time, and since it is performed twice in the code.

For the delete method, the space-time complexity is O(N), where n is the number of elements in the bucket of the hash table corresponding to the key being deleted. This is because the code performs a removal operation on the "Address" list, which takes O(N) time.

The space complexity is also O(N) where N is the number of items inserted into the hash table.

-------------------------------------------------------------------------------------------------------------------------------

**Space-Time complexity of the elements within the Trucks.py file are as follows:**

**Class Trucks:** This class is referring to the constructor that gathers and manages all data about each of the 3 trucks as required. The space-time complexity of this code is O(1), as it is just initializing and setting values to the instance variables of the Trucks class. The Space complexity is dependent on the size of the memory occupied by the object which is constant and does not depend on the size of the input data. It only creates variables for each instance with a constant size of storage.

------------------------------------------------------------------------------------------------------------------------------

**Space-Time complexity of the elements within the Packages.py file are as follows:**

**Class Packages:** This class is referring to the constructor that gathers and manages all data about each of the 40 packages. The space-time complexity of the “init” method of the Packages class is O(1), since it performs a constant amount of work, regardless of the size of its inputs. The time complexity of the “str” method is O(1). The method is just formatting and returning a string and this operation takes a constant amount of time, regardless of the size of the input. The time complexity of the update\_Package\_Status method of the Packages class is O(1) as it performs a constant number of operations, regardless of the size of the input data. The space complexity is O(1) since it only creates a single instance of the package with its attributes and the size of the data structure does not depend on the size of the input.

-------------------------------------------------------------------------------------------------------------------------------

**Space-Time Complexity of The Program as a Whole**

The space-time complexity of the entire program is O(N^2). This is because there is no higher order complexity than O(N^2 )within this program. This complexity comes from the greedy\_Algorithm function. The space complexity though is linear (O(N)) and is entirely dependent on the given data set. As far as space complexity is concerned, the entire program has a space complexity of O(N) which is linear complexity. This is the only space complexity within each function or class within this program.

**Capability and Scalability**

**Scalability**: The greedy algorithm with a time complexity of O(N^2) and a data set in a hash table containing 40 elements is poor. As the size of the dataset increases, the running time of the algorithm will grow quadratically, which can lead to slow performance. For example, if the dataset size doubles to 80 elements, the running time of the algorithm will increase by a factor of 4. In general, this makes the greedy algorithm less suitable for large-scale datasets or applications that require fast query times.

**Capability:** The capability of a greedy algorithm with a time complexity of O(N^2) depends on the problem being solved and the size of the input. For relatively small data sets such as this one for this project, a O(N^2) greedy algorithm can be an effective and efficient solution, providing good results with reasonable running times. However, as the size of the input grows, the time required to run the algorithm will also grow quadratically, making it less practical for large-scale problems.

**Efficiency and Maintainability**

**Efficiency:** The efficiency of a greedy algorithm with a time complexity of O(N^2) is adequate for small or medium sized data sets. However, for larger input sizes, the algorithm may become too slow and inefficient to provide a practical solution. In general, algorithms with polynomial time complexity, such as O(N^2), can handle problems of small or moderate sizes, but may not be scalable to handle larger or more complex problems.

**Maintainability:** In general, greedy algorithms can be relatively simple and easy to understand, which can make them easier to maintain than more complex optimization. However, as the size and complexity of the problem being solved increases, the code implementing the algorithm can become more complex and difficult to understand and maintain.

**Strengths of the greedy algorithm**

**Simplicity**: The greedy algorithm is simple and easy to implement. It requires no complex mathematical formulas or knowledge of advanced data structures, making it a great choice for solving many problems.

**Speed:** The greedy algorithm is often very fast and efficient, making it an ideal choice for solving large-scale problems in real-time.

**Scalability:** The greedy algorithm scales well with the size of the problem, meaning that it can handle problems with large input sizes without compromising on efficiency or accuracy.

**Strengths of a Self-Adjusting Data Structure**

**Dynamic Performance:** Self-adjusting data structures can dynamically adapt to changes in the data which leads to improved performance over time. They can automatically adjust their internal structure to optimize for the current access pattern.

**Ease of Use**: Self-adjusting data structures can be easier to use than static data structures since they don’t require manual tuning or optimization for different access patterns.

**Flexibility:** Self-adjusting data structures can handle a wide range of access patterns which makes them well-suited to use in dynamic and unpredictable environments.

**Weaknesses of the greedy algorithm**

**Local optimum:** The greedy algorithm makes the locally optimal choice at each step without considering the overall solution. Therefore, it may not always produce the global optimum solution.

**No backtracking:** The greedy algorithm does not backtrack to consider alternatives once a decision has been made. Therefore, it can get stuck in a suboptimal solution that is difficult to improve.

**Inefficiency:** In some cases, the greedy algorithm may need to evaluate all possible options to make the best choice at each step, resulting in an inefficient solution.

**Not adaptable:** The greedy algorithm is not adaptable to changing input, and a small change in the input data can significantly affect the solution.

**Weaknesses of a Self-Adjusting Data Structure**

**Performance Overheads:** The self-adjusting process can be time-consuming, and the overhead of performing the self-adjustments can affect the overall performance of the data structure.

**Memory Usage:** Self-adjusting data structures often require additional memory to store the metadata used in the self-adjusting process, leading to increased memory usage.

**Complexity:** Self-adjusting data structures can be complex to design and implement, compared to their static data structures.

**Greedy Algorithm Compared to Dijkstra's and Nearest Neighbor Algorithm**

**Nearest Neighbor Algorithm:** The nearest neighbor algorithm is a simple algorithm used to find the shortest path between two nodes. It works by starting at the source node and repeatedly moving to the nearest unvisited node until the destination node is reached. This algorithm is simple and fast, but it might not produce the best solution.

**Dijkstra's Algorithm:** Dijkstra's algorithm is also an algorithm for finding the shortest path between two nodes. It works by maintaining a priority queue of unvisited nodes, where the node with the shortest distance from the source node is always processed first. Dijkstra's algorithm always finds the optimal solution, but it can be slower than the greedy algorithm for very large data sets.

**Greedy Algorithm:** Greedy algorithm is an algorithm that makes the locally optimal choice at each step in the hope of finding the best solution. Greedy algorithms can be used to solve many optimization problems including finding the shortest path. However, they do not always guarantee an optimal solution and in some cases, it may produce suboptimal solutions compared to the Nearest Neighbor algorithm. I chose this algorithm for this project because it is relatively efficient with a data set of this size and would provide optimal results that would be comparable to the nearest neighbor algorithm and especially Dijkstra's algorithm.

**Data Structure overview**

I believe that using a hash table in this instance is very relevant for multiple reasons. These reasons include that they allow for constant time data look up regardless of the number of elements in the hash. They also can be resized dynamically, and they are efficient with space as well. The time needed to perform the look up function, or time complexity, for this project would increase quadratically as more packages were added to the data set. For instance, if I were to double the number of packages in this data set, the runtime would be increased by a factor of 4. When referring to the space usage in this data structure the space complexity of the data structure will also increase because more memory is needed to store the additional data elements. This increase in memory use is due to several factors including the size, number of pointers, and the overhead of the program. Increasing the number of trucks available to you would have some benefits as well as some draw backs. Benefits include each truck being able to have less packages and therefore being able to deliver them faster, but this would also be reliant on the constraints in the requirements for this project. The draw backs would include that there would be more data to process overall thus slowing down the program’s algorithm since it would need to create more routes for the number of trucks so the utilization of space is a factor to be considered as far as look up time is concerned it wouldn’t change much since the package data isn’t changing but this depends on the development of the program as well. An increase of the number of cities would affect the time it takes for these packages to be delivered. This increase would also affect the space complexity of this program and would utilize more space to store data. As stated before, an increase in the data set in my given project would increase the runtime by a factor of 4 if you were to only double the number of packages. This is also true with other aspects of increasing data in this data set. I would say this data structure (Hash Table) is a good choice for this data set since it is efficient time wise for data lookups, they are efficient with space as well as are able to be resized dynamically. You could have completed this project using queues or stacks as the data structure but using either of these options would be a detriment to the program. While they are both efficient as well, they do not contain any key values or any key value pairs making them linear in nature while hash tables are essentially associative arrays, or maps. Additionally, stacks have a last in first out (LIFO) pattern and queues have a first in first out (FIFO). Hash tables do not contain these patterns since, again, they are associative arrays. This Hash table does not account for the relationship between data points though. This hash table stores key-value pairs and use a hash function to map the keys to indices in an array. The relationship between data points is not explicitly represented in the hash table but can be implemented through the values associated with each key. As far as collisions go, I was able to manage this by using a chaining hash table so when a collision occurs, a new key value pair is added to the linked list at the index.

**What I would do differently**

If I were to do this project again, I would consider being more specific with the user interface and make it clearer than it currently is. I would do this by printing out the status of each package at any specific time in a more detailed manner rather than simply printing out the stored array values. I would also consider adding more attributes to the “Trucks” Class allowing for a more detailed and thorough representation of the status of each truck. For example, I would consider adding in which truck each package was on when it was delivered. Additionally, I would also take into consideration the time complexity more than I did when starting this project. The reason for this is because as the number of packages increases, the time it takes to process all the data increases dramatically due to the chosen algorithm (Greedy Algorithm). I’d likely consider using Dijkstra's algorithm since this algorithm always produce the optimal result though, at a deficit in time. +