C950 WGUPS Algorithm Overview

Linda Vataksi

ID # 011273373

WGU Email: lvataks@wgu.edu

Date: 02/ 26/ 2024

C950 Data Structures and Algorithms II

TASK 1: WGUPS ROUTING PROGRAM PLANNING

# Introduction

TASK 1: WGUPS ROUTING PROGRAM PLANNING

# A. Algorithm Identification

The self-adjusting algorithm used to create my program to deliver the packages is the nearest neighbor algorithm. The algorithm will initialize an empty array “in transit” to keep track of packages in the process of delivery. While the packages are in transit the algorithm will iteratively select the next package to deliver by calculating the distance between the truck’s current location and each package's delivery address. This algorithm seeks to optimize the delivery path by selecting the shortest distances as the next stop for deliveries.

# B1.

A data structure, such as a hash table, is used in coupling with the algorithm identified in part A by storing the data points efficiently for quicker access during the algorithm search. Implementing a hash table as a data structure helps reduce the overall time complexity.

# C1.

Algorithm’s Psuedocode:

|  |
| --- |
| Function truckDeliverPackages(truck):  Initialize an empty array for in-transit packages  For each package in the truck  Search for the package ID in the hash table.  If a package is not found:  Add the package to “In Transit”  Clear packages from the truck hash table  While there are packages in Transit:  For each package in transit:  Calculate the distance between the package street and truck’s current location.  If distance calculated < the next stop,  Update nextStop to distance\_to\_package  Update next\_package to package  //Deliver the nearest package  Add truck to packages  Remove package from in transit  Update the truck’s attributes:  Update truck miles according to next stop  Update truck location according to next stop  Increment truck’s departureTime  Calculate delivery time by considering departTime  Return total miles traveled by the truck |

# C2.

Hardware:

* Windows 11 Home
* System type: 64-bit operating system, x64-based processor
* Processor: Intel(R) Core(TM) i7-10510U CPU @ 1.80GHz 2.30 GHz

Software:

* Visual Studio Code Version 1.86.2
* Python version 3.12.2

# C3.

Space-time complexities of the major segments of the code can be found in in-line comments in the submitted Python files, as well as below.

ChainingHashTable() 🡪

Overall Time Complexity- O(N), Overall Space Complexity- O(M)

|  |
| --- |
| # Creating the hash table  # Source: W-1\_ChainingHashTable\_zyBooks\_Key-Value.py  class ChainingHashTable:      # Initializing = O(N)      def \_\_init\_\_(self, initialcapacity=40):          self.table = []          for i in range(initialcapacity):              self.table.append([])      # Inserts a new item into the hash table and will update an item in the list already      # Insertion time complexity = O(N)      def insert(self, key, item):          bucket = hash(key) % len(self.table)          bucket\_list = self.table[bucket]          # update key if it is already in the bucket          for kv in bucket\_list:              # print (key\_value)              if kv[0] == key:                  kv[1] = item                  return True          # if not in the bucket, insert item to the end of the list          key\_value = [key, item]          bucket\_list.append(key\_value)          return True      # Searches the hash table for an item with the matching key      # Will return the item if founcd, or None if not found        # Search time complexity = O(N)      def search(self, key):          bucket = hash(key) % len(self.table)          bucket\_list = self.table[bucket]          # print(bucket\_list)          # search key in bucket          for kv in bucket\_list:              # print(key\_value)              if kv[0] == key:                  return kv[1]  # value          return None      # Removes an item with matching key from the hash table      # Removal time complexity = O(N)      def remove(self, key):          bucket = hash(key) % len(self.table)          bucket\_list = self.table[bucket]          # removes the item if it is present          if key in bucket\_list:              bucket\_list.remove(key)  # Overall space complexity - 0(M), time complexity - O(N) |

def loadPackageData(filename) 🡪

Overall Time Complexity- O(N), Overall Space Complexity- O(M)

|  |
| --- |
| # Creating loadPackageData() to read packages from packagesCSV file and insert into defined hash function  # OVERALL: Time complexity is O(N) and space complexity is O(M)  def loadPackageData(filename):      # Opening file= time and space complexity of O(1)      with open(filename) as packagess:          packageInfo = csv.reader(packagess, delimiter=',')          next(packageInfo)          # Space complexity- O(M) and time complexity O(N)          for package in packageInfo:              # Space complexity- O(1) and time complexity O(1)              pID = int(package[0])              pStreet = package[1]              pCity = package[2]              pState = package[3]              pZip = package[4]              pDeadline = package[5]              pWeight = package[6]              pNotes = package[7]              pStatus = "At the Hub"              pDepartureTime = None              pDeliveryTime = None              # Inserting Package info into the hash              # Time and space complexity of O(1)              p = Package(pID, pStreet, pCity, pState, pZip, pDeadline,                          pWeight, pStatus, pDepartureTime, pDeliveryTime)              packageHashTable.insert(pID, p) |

def address(address) 🡪

Time complexity = O(N), and space complexity = O(1)

|  |
| --- |
| # Time complexity- O(N) and and space complexity - O(1)  def address(address):      # checks each row in address file      # for loop has time complexity of O(N), while within the time complexity is O(1)      for row in addressCSV:          # if there is a string present in the third row item, return the first row item          if address in row[2]:              return int(row[0]) |

def distanceBetween(address1, address2) 🡪

Time complexity = O (1), and space complexity = O(1)

|  |
| --- |
| # Time and space complexity is O(1)  def distanceBetween(address1, address2):      try:          # Time complexity- O(1)          distanceBtwn = distanceCSV[address1][address2]          # if empty, flip the addresses          # Time complexity- O(1)          if distanceBtwn == '':              distanceBtwn = distanceCSV[address2][address1]          return float(distanceBtwn)      except ValueError:          # Handle the case where distance cannot be converted to float          return None      except IndexError:          # Handle the case where addresses are out of range          return None |

def truckDeliverPackages(truck) 🡪

Time complexity = O(N^2), and space complexity = O(M)

|  |
| --- |
| # Time complexity- O(N^2) and space complexity of O(M)  def truckDeliverPackages(truck):      # initialize an empty array for in transit packages      # space complexity- O(M)      inTransit = []      # Time complexity- O(N)      for packageID in truck.packages:          package = packageHashTable.search(packageID)          if package is not None:  # Check if package is found              # add all packages from hash table to the in transit array              inTransit.append(package)      # clear packages from truck hash, since they are now in transit      truck.packages.clear()      # entire while loop= time complexity of O(N^2)      while inTransit:          # arbitrary value greater than distance\_to\_package          nextStop = float('inf')          next\_package = None          # for loop- time complexity O(N)          for package in inTransit:              distance\_to\_package = distanceBetween(                  address(package.street), address(truck.currentLocation))              # if distance to package  is less, that will determine the next stop for delivery              if distance\_to\_package < nextStop:                  nextStop = distance\_to\_package                  next\_package = package          # append          truck.packages.append(next\_package.ID)          # remove from inTransit array          inTransit.remove(next\_package)          truck.miles += nextStop          truck.currentLocation = next\_package.street          truck.departTime += datetime.timedelta(hours=nextStop / 18)          next\_package.deliveryTime = truck.departTime          next\_package.departureTime = truck.departTime      return truck.miles |

UI: solo input 🡪

Time complexity = O(1), and space complexity = O(1)

UI: Time complexity for all packages 🡪

Time complexity = O(N), and space complexity = O(1)

|  |
| --- |
| # Part D- UI  # Overall time complexity is O(1) id user chooses solo input, and O(N) for second option  # Space complexity is O(1)  # Ask the user if they would like to inquire about the delivery status of a particular  # package or of all packages  user\_solo\_or\_all = int(      input("Enter 1: Solo package status, or 2: All package status --> "))  # Time complexity for solo input= O(1)  # user wants more information on one specified package  if user\_solo\_or\_all == 1:      # user inputs a time      user\_input\_time = input(          "Please enter a time to check status of the desired package. Use the provided format, HH:MM:SS ")      (h, m, s) = user\_input\_time.split(":")      user\_time = datetime.timedelta(          hours=int(h), minutes=int(m), seconds=int(s))      user\_input\_packageID = input("Enter package ID: ")      # search for the packageID in the hash table      solo\_package = packageHashTable.search(int(user\_input\_packageID))      # find the delivery time      solo\_package\_deliveryTime = solo\_package.deliveryTime      # determine package status based off deliverytime      if user\_time < solo\_package\_deliveryTime:          print("In Transit")      elif user\_time > solo\_package\_deliveryTime:          print("Delivered")      else:          print("At the Hub")  # Time complexity for all packages= O(N)  # user wants more information on all packages  elif user\_solo\_or\_all == 2:      user\_input\_time = input(          "Please enter a time to check status of all the packages. Use the provided format, HH:MM:SS ")      (h, m, s) = user\_input\_time.split(":")      user\_time = datetime.timedelta(          hours=int(h), minutes=int(m), seconds=int(s))      # range includes package with id 40      for packageID in range(1, 41):          package = packageHashTable.search(packageID)          if package.deliveryTime is not None and user\_time < package.deliveryTime:              print(f"Package {package.ID}: In Transit")          elif package.deliveryTime is not None and user\_time > package.deliveryTime:              print(f"Package {package.ID}: Delivered")          else:              print(f"Package {package.ID}: At Hub")  else:      # error message, improper user input      print("ERROR: Must enter integer 1 or 2") |

# C4.

The solution provided is capable of scaling and adapting to a growing number of packages due to the efficient package lookup offered by the hash table, which serves as an optimized data structure. Additionally, the algorithm for delivery is optimized for selecting the nearest package for delivery. By iteratively selecting the nearest package for delivery, the algorithm minimizes the number of distance calculations required, reducing the computational complexity and making it well-suited for scaling to a growing number of packages.

# C5.

The software design would be efficient and easy to maintain due to the modularity. The software design has distinct components responsible for package lookup (hash table) and delivery (nearest neighbor algorithm). This modularity enhances maintainability by allowing developers to focus on individual components without impacting the entire system.

Another factor that makes the software design easy to maintain is its code readability. The software design is accompanied by thorough documentation and comments, providing clear explanations of the system architecture, algorithms used, and rationale behind design decisions. This documentation facilitates easier understanding for developers and aids in maintaining the system over time.

# C6.

There are strengths and weaknesses of using a hash table for the self-adjusting data structure of this software design. According to Section 6.1 of the zyBooks textbook, “A hash table's main advantage is that searching (or inserting / removing) an item may require only O(1), in contrast to O(N) for searching a list or to O(log N) for binary search.” This fast lookup time is largely attributed to the flexible key-value storage. One major disadvantage of hash tables is hash collisions. According to Section 6.1 of the zyBooks textbook, “A collision occurs when an item being inserted into a hash table maps to the same bucket as an existing item in the hash table.” Resolving collisions can impact performance and may require additional processing, such as using collision resolution techniques like chaining or open addressing.

# C7.

The most effective choice for efficient delivery management among the provided components is the package ID. As a unique identifier assigned to each package, the package ID ensures fast and direct access to specific packages within the delivery system. This uniqueness facilitates efficient indexing and retrieval operations, regardless of the size of the dataset. Package IDs remain constant throughout the package's lifecycle, providing consistency and reliability for tracking and managing packages at various stages of delivery. Moreover, the independence of package IDs from other package attributes allows for easier management and modification of package information without affecting the key itself.

# D. Sources - Works Cited

Lysecky, R., & Vahid, F. (2018, June). *C950: Data Structures and Algorithms II*. zyBooks.

Retrieved March 22, 2021, from <https://learn.zybooks.com/zybook/WGUC950AY20182019/>