By: Salvador Amaya

WGU C950 - DSA II

Algorithm Recognition

To efficiently sort and deliver parcels to trucks, I used a greedy algorithm to create a distribution path with the shortest path for each parcel. The order of division is as follows.

- The truck-specific list of available parcels is allocated based on the distance between each parcel with find_nearest_package_in_list() until no more parcels can be allocated to the truck.
- When a truck has loaded the maximum number of parcels it can carry or when all parcels have been delivered, the truck is ready to make a shipment.
- The truck then delivers all the parcels in the order they were added "**deliver_all_packages(ht, truck_list)**" until the parcels are proper and returned to the distribution center.
- If there are still parcels left undelivered, the truck will load as many parcels as possible until the maximum number of parcels are delivered or all parcels are delivered. The truck will then make another dispatch point or more if necessary until all parcels are delivered.

Logic Commentary: Pseudo code

```
Initialize Trucks
Initialize Drivers
  Delay the start time of one Truck to the earliest time of delayed parcels arriving at the depot

Assign parcels
  For each Truck of Trucks
    Until the Truck is full or no parcels are available to assign
       Find and Assign the parcels with the shortest distance to the last added parcels
Deliver parcels
  While all parcels are not delivered
    For each Truck of Trucks
      Until all current parcels loaded onto the Truck is delivered
        Deliver the parcels in the order they were added
    Return to the Hub
    Assign additional parcels
```

-https://pseudoeditor.com/app/

Pseudo code is the essence of how the structure or rather blueprint of a program is written. This step is critical to how we can utilize logic to begin implementing code on a step-by-step basis.

Dev. Environment

Integrated Development Environment: PyCharm 2024.3.1.1 (Professional Edition)

Build - 243.22562.220

Runtime - 10.0.6+7-b525.65

Programming Language - Python Version 3.8

Components:

CPU: Intel® i7 7700k CPU @ 3.60 GHz

By: Salvador Amaya

WGU C950 - DSA II

Storage: 32.0 GB 3500 MHz DDR5

Space-Time & Big-O

The time and space complexity for key sections of the program are provided as comments above each corresponding function

Time and Space Complexity Analysis

1. **Hash Table Insertion (insertPackage())**

Time Complexity:

Average: **O(1)** (assuming good hash function, low collisions)

Worst case (collisions): **O(n)** (chaining or open addressing)

Space Complexity:

O(1) for insertion; overall **O(n)** for storing parcels in the hash table.

2. Hash Table Lookup (getPackage())

Time Complexity:

Average: **O(1)**

Worst case (collisions): **O(n)**

Space Complexity:

O(1)

3. Resizing the Hash Table (resizeTable())

Time Complexity:

O(n) (re-inserting all elements when resizing)

Space Complexity:

O(n) (new array for larger table)

4. Traversing the Hash Table

Time Complexity:

O(n) (visiting each element)

Space Complexity:

O(1)

By: Salvador Amaya

WGU C950 - DSA II

Overall Program Complexity:

Time Complexity:

Average: **O(1)** for insertions and lookups, O(n) for resizing.

Space Complexity:

O(n) for storing all parcels.

Scalability & Adaptability

The HashTable is initially designed to hold 40 parcels. If more parcels are added, the HashTable automatically resizes by doubling its capacity. All existing parcels are then reinserted into the expanded HashTable before any new parcels are added.

Time Complexity:

1. **Insertion (Before Resize)**:

   Average: **O(1)**. The table can accommodate parcels without resizing.

2. **Insertion (During Resize)**:

   O(n), where *n* is the number of parcels. Resizing involves creating a new array and reinserting all elements.

   Although resizing is costly, it happens infrequently, keeping the average insertion time close to **O(1)**.

3. **Lookup**:

   Average: **O(1)**. No performance degradation during lookups.

Space Complexity:

- **Before Resize**: **O(n)** for storing parcels.
- **During Resize**: Temporary **O(2n)** as the new array is allocated.

Adaptability:

- The hash table adapts to growing datasets by resizing dynamically, maintaining a low load factor and ensuring efficient operations. However, frequent resizing could degrade performance temporarily.

While resize operations introduce **O(n)** complexity, overall, the hash table performs **O(1)** on average for insertion and lookup, with **O(n)** space complexity, ensuring scalability and adaptability for increasing parcel volumes.

By: Salvador Amaya

WGU C950 - DSA II

Software Efficiency & Maintainability

The software demonstrates efficiency by meeting key distribution criteria, including adhering to distribution deadlines, maintaining a mileage limit of 140 miles, and effectively managing edge cases like parcels arriving late at the depot or parcels with incorrect addresses.

Additionally, the software is effortlessly simple to maintain due to comprehensive documentation, which enhances user usability and facilitates the modification of individual components without affecting the rest of the system.

Key-Choice Justification

For efficient distribution management, the most suitable key choices are:

1. Parcel ID

- **Justification**: Unique to each parcel, it enables **O(1) lookups** in a hash table, ensuring fast and efficient retrieval.

2. Tracking Number

- **Justification**: Commonly used for tracking parcels, it also allows **O(1) lookups** and efficient parcel status retrieval.

3. Sender/Receiver Details & Delivery Date

- **Justification**: These keys are less efficient due to potential collisions or the need for complex searches, leading to **O(n)** time complexity.

Conclusion:

The Parcel ID or Tracking Number are ideal for fast and efficient parcel management with **O(1)** retrieval. Other options introduce inefficiencies.

Self-Adjusting Data Structures

One of the strengths of the HashTable is its ability to insert, remove, and retrieve parcels with an average time complexity of O(1). Additionally, it allows for efficient iteration through the stored parcels with a time complexity of O(N).

However, a limitation of the HashTable is the need for resizing when the table reaches its threshold. Since the HashTable is initialized with a fixed size, once it’s full, any new parcels inserted will trigger a

resizing process that doubles the table's capacity. While this allows for continued insertion, it can result in wasted space if not all buckets are utilized.

Data Structure

The solution utilizes a hash table as the dynamic data structure to efficiently store and retrieve parcel information. A hash table is a key-value store where the key is used to calculate an index in the underlying array, called the "package_table" in this case, where the corresponding value (parcel information) is stored.

How the Hash Table Works:

Hash Function: A hash function is applied to the key, which might be something like a parcel ID, address, or tracking number. This function computes an index based on the key, ensuring the value can be stored at a specific location in the array. The goal is to distribute the data across the array in such a way that retrieval remains fast (typically O(1) time complexity).

Handling Collisions: In case two keys hash to the same index (a "collision"), the hash table must handle this scenario. Common techniques to handle collisions include chaining (where a linked list is used to store multiple values at the same index) or open addressing (where alternative indices are probed).

Package Table Array: The "package_table" array is a dynamic array that grows as needed. The parcels (values) are inserted into this array based on their corresponding hash indices. The parcels are typically stored as objects or structs that contain all necessary information, such as sender and recipient details, parcel weight, dimensions, and status.

Efficient Lookups: With the parcel information stored in a hash table, lookups for specific parcels (using their key) can be done very efficiently. This enables rapid retrieval of parcel data without the need to search through an entire list or array.

Relationship Between Data Components:

The key-value structure of the hash table inherently organizes the parcel information by the key (e.g., tracking number). This ensures that each parcel can be uniquely identified and quickly accessed based on its specific attributes. The "package_table" array provides a convenient and efficient way to store and manage parcel information, while the hash function facilitates quick access.

By: Salvador Amaya

WGU C950 - DSA II

Explication of Data Structure

The HashTable is utilized to contain parcels and authorizes retrieval through a lookup based on the parcel ID.

Strengths of the Picked Algorithm

The algorithm is designed to scale with the number of drivers, parcels, and trucks, and can adjust to the specific characteristics of each item. For example, it can handle trucks of various sizes, each capable of carrying different numbers of parcels.

It also intelligently manages the relationship between the number of drivers and vehicles, ensuring that only the necessary resources are utilized. If there are 3 drivers and 2 vehicles, for example, the algorithm will plan shipments based on the availability of the 2 trucks, optimizing their usage.

Furthermore, the algorithm can accommodate delays, allowing a vehicle to start at a different time if a parcel has a specified delayed ETA, such as a “Postponed flight” status.

Authentication of Algorithm

All criteria have been satisfied and can be verified by generating a report through the interactive CMD interface. By selecting “1” from the menu options and entering “2:00 PM” as the designated time, a summary of the parcels will be generated once they have been delivered.

Alternative Feasible Algorithms

Dijkstra's algorithm & Depth-First Search (DFS) for finding the shortest route are alternative methods that could also be applied to this approach.

Algorithm Distinctions

Dijkstra’s & DFS shortest route algorithms are designed to determine the shortest distance between two vertices in a graph structure. In contrast, the greedy algorithm uses a different strategy, identifying the shortest route among all addresses to select the next closest parcel to assign to a vehicle.

Alternative Proposition

By: Salvador Amaya

WGU C950 - DSA II

If I were to revisit this task, I would break down the different components of the parcel assignment function into separate tasks to reduce both the space and time complexity of the assign_parcels function. Currently, it is inefficient because each loop iteration requires the algorithm to compile a list of parcels that can be assigned to the specific truck.

By prioritizing the assignment of high-priority parcels first, and then using a method to assign the remaining "regular" parcels (those with no special instructions or ETAs), the space and time complexity could be significantly reduced.

Operating Costs

As the quantity of parcels increases, the HashTable's memory usage will also expand.

Entanglements

Increasing the number of vehicles could potentially involve more vehicles in the distribution process, provided there are enough operators to manage them. Each additional vehicle would result in a linear increase in memory usage, but the time complexity would remain unchanged. Since vehicles are stored in a list, lookup time would still be O(N).

Expanding the number of urbanizations and distances will not impact the lookup time, as the data is stored in a 2D array, which can be accessed in O(1) time. However, the space complexity would grow at O(N^2) since each city would require distance data to be calculated against every other city.

Alternative Data Structures

A replacement data structure for the suggested plan could include a network of nodes or a linked list.

Data Structure Differences

Using a network of nodes, each node can represent a city address, and the weight of each edge can represent the distance between nodes. This model will provide a good representation of the car’s delivery method, with the edge weight reflecting the mileage of the address. The nodes and edges provide a simple way to calculate the total number of trips each vehicle takes, depending on the route it takes. In contrast, the HashTable model used in this solution does not have data to represent the distance to the delivery point of each package, and does not track the vehicle’s delivery route.

By: Salvador Amaya

WGU C950 - DSA II

Linked lists offer a versatile data structure for storing parcels and would likely use less space compared to the HashTable's package_table, particularly when the HashTable needs resizing. The arrangement of linked elements in the list could reflect the order of deliveries for the parcels.

Sources

The primary resource used for this project's data structure implementation was the Zybooks online textbook. The section "**Linear probing**" is cited below.

Zybooks. (n.d.). Retrieved March 20, 2022, from https://learn.zybooks.com/zybook/WGUC950AY20182019/chapter/7/section/8