Brandon Robinson
Student # 006868740

C950: Data Structures & Algorithms II

Core Algorithm Overview

Stated Problem:

This project aims to develop an efficient routing and delivery optimization system for the Western Governors University Parcel Service (WGUPS) using a common high-level programming language (Python 3.12). Forty packages must be split up across two trucks with three different drivers. Some packages have delivery constraints, and some have been delayed. Additionally, there is a package that initially has the wrong address and packages that must go out on the second truck. To solve this problem, we must first use a series of conditional statements to load the trucks based on the provided data. We then must implement a greedy algorithm to optimize the delivery of each package along the truck route. This algorithm is greedy because it determines the shortest available path from its current location and continues to do this until no additional packages remain. This article will analyze the use of this algorithm and provide a descriptive overview of the application methods and components.

Algorithm Overview:

The greedy algorithm operates as follows:

1. **Initialization**: The input includes the list of packages on a truck, the truck number, and the starting location (the default is the hub).
2. **Finding the Nearest Package**:
    - Compare the current location with all package destinations to identify the nearest location.
    - Update the shortest distance and destination dynamically.
3. **Delivery Process**:
    - Remove the selected package from the truck's list and move the truck to the new location.
    - Append the delivered package to an optimized delivery list.
    - Update the current location and recursively process the remaining packages.
4. **Termination**: The recursion ends when the truck's package list is empty.

The space-time complexity of this self-adjusting greedy algorithm has a worst-case runtime of O(N^2) and a best-case runtime of O(1). The worst case is almost always guaranteed, as the best case is only possible when the list of packages being loaded onto a truck is empty. A pseudocode representation is provided below to prove this…

Greedy Algorithm

A. Take the following parameters

1. **truck_list (represents a nested list of packages on a given truck)**
2. **truck_number (represents the truck you are working with)**
3. **current_location (recursive variable that is used to show where the truck is)**

B. Create the base case to break the recursion in the algorithm (see to see recursive step)

```
    if length of truck_list is 0 then return the empty list
```

C. Enter the recursive sections if the truck_list is not empty,

```
    set lowest_value to 50.0

    (represents the closest deliverable location from the current location in miles, ex: 3.2)

    set new_location to 0

    (represents where the truck is moving next to deliver a package)

    if length of truck_list is not 0:

        for index in truck_list:

            if check_current_distance of current_location and the next index in the truck
            package list <= lowest_value:

            (Searches all reachable locations and updates the lowest_value if a smaller mileage
            path is found).

                new lowest_value exists then update lowest_value and current_path
```

The time complexity of the algorithm so far is as follows:

A.) O(1) or constant time B.) O(1) or constant time C.) O(N)

O(1) + O(1) + O(N) = O(N)

D. Enter the second for loop in the recursive section of the algorithm

```
    for index in truck_list:

        if check_current_distance of current_location and the next index in the truck package list is
        equal to lowest_value:

        (search through the list of packages again to find the package address value that is equal to
        lowest_value (see section c). This allows us to deliver multiple packages to the same address
        if they are in the same truck)

            if truck_number is equal to 1, 2, or 3

            (determines which truck object was originally passed in so it can create an
            optimized list for that truck)

                optimized_truck.append(current package)

                (Append the selected package from truck_list to the optimized_truck list)

                optimized_truck_index.append(current package index)
                (Append the selected package address index from truck_list to the
                optimized_truck index list)
                pop_value = truck_list.index(current package)
                truck_list.pop(pop_value)
                (Remove the selected package from truck_list)
        current_location = new_location
        (Update the current location to show that the truck has moved to a new location)
        calculate_shortest_distance(truck_list, truck_number, current_location)
        calls algorithm function again to loop through the shorter truck_list with an
        updated location
```

The total time complexity of the algorithm is as follows:

A.) O(1) or constant time B.) O(1) or constant time C.) O(N) D.) O(N^2)

O(1) + O(1) + O(N) + O(N^2) = O(N^2)

Below is a table breakdown of the worst-case space-time complexity of each file in the Python application:

HashTable.py

Method	**Line Number**	**Space Complexity**	**Time Complexity**
__init__	12	O(1)	O(1)
_get_hash	20	O(1)	O(1)
insert	26	O(N)	O(N)
update	42	O(1)	O(N)
get	55	O(N)	O(N)
delete	64	O(N)	O(N)
Total		3N + 3 = O(N)	4N + 2 = O(N)

ReadCSV.py

Method	**Line Number**	**Space Complexity**	**Time Complexity**
None	15	O(N)	O(N)
get_hash_map	60	O(1)	O(1)
check_first_truck_trip	65	O(1)	O(1)
check_second_truck_trip	70	O(1)	O(1)
check_third_truck_trip	75	O(1)	O(1)
Total		N + 4 = O(N)	N + 4 = O(N)

Packages.py

Method	**Line Number**	**Space Complexity**	**Time Complexity**
None	43	O(1)	O(N)
None	51	O(N^2)	O(N^2)
None	66	O(N)	O(N)
None	80	O(1)	O(N)
None	88	O(N^2)	O(N^2)
None	102	O(N)	O(N)
None	117	O(1)	O(N)
None	125	O(N^2)	O(N^2)
None	139	O(N)	O(N)
total_distance	153	O(1)	O(1)
Total		3N^2 + 3N + 3 = O(N^2)	3N^2 + 6N + 1 = O(N^2)

Main.py

Method	**Line Number**	**Space Complexity**	**Time Complexity**
None	15	O(1)	O(N)
None	24	O(N)	O(N^2)
Total		N + 1 = O(N)	2N = O(N^2)

Distances.py

Method	**Line Number**	**Space Complexity**	**Time Complexity**
check_distance	17	O(1)	O(1)
check_current_distance	27	O(1)	O(1)
check_time_first_truck	41	O(N)	O(N)
check_time_second_truck	53	O(N)	O(N)
check_time_third_truck	65	O(N)	O(N)
check_address	79	O(1)	O(1)
calculate_shortest_distance	110	O(N^2)	O(N^2)
first_optimized_truck_index	150	O(1)	O(1)
first_optimized_truck_list	155	O(1)	O(1)
second_optimized_truck_index	160	O(1)	O(1)
second_optimized_truck_list	164	O(1)	O(1)
third_optimized_truck_index	170	O(1)	O(1)
third_optimized_truck_list	174	O(1)	O(1)
Total		N^2 + 3N + 9 = O(N^2)	N^2 + 3N + 9 = O(N^2)

Memory and computational time remain nearly linear throughout the entire application. This allows the available set of inputs to scale without being overburdened by memory availability constraints. Bandwidth is not a factor in the current implementation as the application is run and managed on a local machine that does not require network resources.

Efficiency Analysis

The algorithm efficiently handles the project's constraints and scales well for the defined dataset. While the complexity could be a limitation for larger datasets, it is acceptable given the project's scope. Additionally, the algorithm ensures that all packages are delivered within a 100-mile range.

Another algorithmic approach I could have used to optimize the packages was dynamic programming. "Dynamic programming is a problem-solving technique that splits a problem into smaller subproblems" ("ZyBooks"). This approach could minimize redundant calculations by breaking the problem into smaller subproblems and storing intermediate results. However, it may increase space complexity.

A second algorithm I could have implemented for the optimal route is a self-adjusting heuristic. This method would dynamically allocate packages to trucks based on proximity to the hub and subsequent locations, potentially improving scalability and efficiency. The chosen approach and the self-adjusting heuristic share similarities in that the shortest available path is always chosen.

Algorithm's Strengths and Weaknesses

The greedy algorithm has a few strengths: simplicity, efficiency for small datasets, scalability, and real-time adaptation. It is straightforward to implement and understand. It performs well with limited packages, meeting the project's requirements. The algorithm can handle varying numbers of packages and locations with minimal adjustments. The dynamic selection of the shortest path allows for efficient routing under changing conditions.

On the other hand, some weaknesses include suboptimization for large datasets, rigidity, and manual package assignment. The complexity becomes a bottleneck with larger datasets, reducing efficiency. The greedy approach may not always find the global optimal solution, as it focuses on immediate gains. Programiz.com (Programiz) states, "This algorithm may not produce the best result for all the problems. It is because it always goes for the local best choice to produce the global best result." The current implementation requires manual allocation of packages to trucks, which could be automated for better scalability.

Programming Models:

The application is implemented in Python 3.12 and executed locally using the PyCharm IDE. Data is sourced from CSV files, and the application does not require network connectivity. This localized setup ensures efficient data handling and minimal external dependencies.

Ability to Adapt

The system is designed to accommodate changes in package numbers, truck capacities, and delivery locations. While package loading is currently manual, automating this process with a heuristic approach could enhance scalability and efficiency. The core algorithm's adaptability allows it to handle diverse datasets with minimal modifications.

Efficiency and Maintainability

Overall, the software is very efficient, with two comparisons with a time efficiency of O(N^2). While this may not be the best time complexity, it scales well with the 16-package limit per truck. It is also very maintainable, as much of the software is the same core functions modified for the use case. Debugging is easier because you can always refer to another instance of the function to determine where potential errors might be.

Data Structures

The primary data structure is a list chosen for its flexibility and compatibility with hash tables. The data key for each package is its package ID, ensuring efficient access and manipulation. I chose this key because it allows for easier tracking for individual packages than the truck, which benefits customers and WGUPS. This structure facilitates efficient data retrieval and manipulation. Alternative structures like binary search trees or graphs could be explored for future iterations to improve scalability and organization.

The greedy algorithm provides a robust and efficient solution for optimizing WGUPS's delivery operations. Brilliant.org (Ross et al.) states, “The algorithm makes the optimal choice at each step as it attempts to find the overall optimal way to solve the entire problem.” While alternative approaches could offer marginal improvements, the current implementation effectively balances simplicity, performance, and scalability. Future enhancements could focus on automating package allocation and exploring advanced data structures to further optimize performance.

Sources and In-Text Citations

“ZyBooks.” *Zybooks.com*, 2025, learn.zybooks.com/zybook/WGUC950Template2023/chapter/2/section/4?content_resource_id=61872168. Accessed 15 Jan. 2025.

Programiz. “Greedy Algorithm.” *Www.programiz.com*, www.programiz.com/dsa/greedy-algorithm. Accessed 24 Jan. 2025.

Ross, Eli, et al. "Greedy Algorithms | Brilliant Math & Science Wiki." *Brilliant.org*, 2016, brilliant.org/wiki/greedy-algorithm/. Accessed 24 Jan. 2025.