Prime Paths: Amazon's Delivery Puzzle

Sanchit Singhal (ss99628, ssinghal37)
Muhammad Rasheed (myr385, muhammadrasheed)
Yash Jain (yj6683, yashj1579)

COE 322
Final Project

1 Introduction

Scheduling delivery routes for trucks is a challenging problem that directly impacts time, cost, and efficiency. Finding the best delivery route involves finding the shortest route from each address that needs a package so that we can minimize the total distance traveled. For Amazon, this problem is even more complicated due to having policies such as multiple trucks and prime delivery. This project aims to explore heuristic solutions for optimizing Amazon's delivery routes. Furthermore, an analysis of the restriction of Prime delivery when determining the shortest route has also been considered. Finally this paper ends with a discussion of the real world implications and looks at the ethical considerations of implementing such an algorithm.

2 Background

This problem falls under a larger set of problems known as the Traveling Salesman Problem (TSP). The Traveling Salesperson Problem is a classic optimization problem where the goal is to determine the shortest route that visits a set of locations and returns to the starting point. The Traveling Salesman Problem is difficult to solve since it is classified as an NP-complete problem, meaning there's no solution that can solve this problem in a polynomial time. This means that as the amount of data increases, the time required to compute the optimal solution grows exponentially.
Example of TSP: Take a look at Austin's public bus system where buses have to take routes to drop off passengers at various hubs. The goal for the bus company is to minimize the total distance traveled while visiting every hub they need to. This would help reduce fuel cost and improve efficiency. This is an example of the Traveling Salesman Problem (TSP), where the challenge is to design a route that minimizes the distance traveled while visiting all the necessary hubs.
The TSP is relevant in numerous applications across diverse fields, such as the problem we are addressing to optimize delivery routes and minimize operation costs. Similarly, in manufacturing, tool paths in machining processes are optimized utilizing TSP, reducing wear and production time. In computational biology, TSP is crucial for shotgun DNA sequencing, where a strand must be assembled in the shortest possible order to accurately reconstruct the original sequence. Despite its computational complexity, the solutions derived from studying TSP have transformed industries and scientific research, driving innovations in optimization and algorithm development.

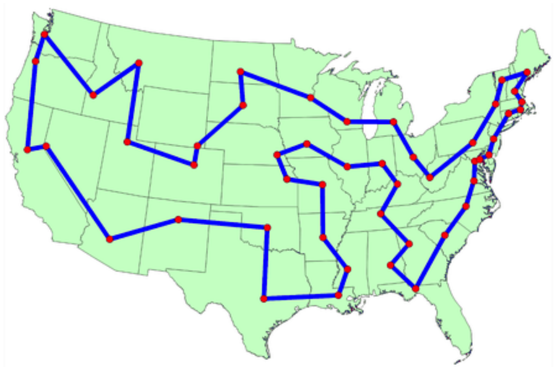

Figure 1: Solution to 48 States Traveling Salesman Problem.

3 Methodology

In order to solve the problem, it was important to consider what information was important and how information would be represented. The standard approach began with building basic tools to represent and analyze delivery routes. Subsequently, additional features and various heuristics were incorporated to enhance route efficiency. To simplify the problem, each delivery address was represented as a point with 2D coordinates, using the depot location as the origin and calculating the relative distance of each house from the depot. It was also crucial to consider the route, which was modeled as an ordered list of these points. Two consecutive points in the list formed an edge, with the length of the edge representing the travel distance. This approach transformed the delivery problem by abstracting it from its real-world context of houses and trucks, simplifying it into a graph of nodes and edges.

3.1 Step 1: Representing Addresses

Firstly, an Address class was created to store the coordinates of delivery locations. This class also allowed us to calculate the distance between two addresses using the Euclidean distance. This method became the foundation for all our route analysis.

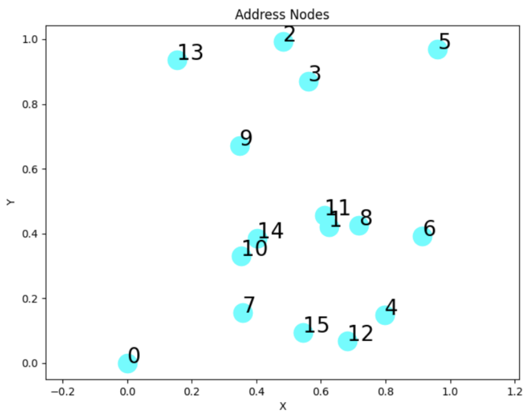

Figure 2: Set of addresses represented as points on a grid.

3.2 Step 2: Representing Routes

Next, representing a delivery route (a collection of addresses) was needed. This led to the creation of an AddressList class, which stores a list of addresses and includes a method to calculate the total distance between all addresses.
The Route class was then introduced to represent the entire truck route, including the start and end points at the depot (the origin). Due to the similarity between the Route class and the AddressList class, it made sense for the Route class to inherit the AddressList class. The following method highlights the similarity between the programs, where Route utilizes AddressList's method length() in order to run its code while still remaining distinct.
For the Address Class:

```
    double length() {
        if (list.empty()) return 0;
        double totLength = 0;
        for (int i = 1; i < list.size(); i++) {
            totLength += list[i-1].distance_euc(list[i]);
        }
        return totLength;
    }
```

For the Route Class:

```
    double length() {
       double totLength = AddressList::length();

       double new_length =  totLength + get_address(num_of_address
    () - 1).distance_euc(Address(0, 0));

       new_length += get_address(0).distance_euc(Address(0, 0));
       return new_length;
   }
```

The Route class builds on the AddressList class by inheriting the length() method to calculate the total distance between addresses while also making sure to account for the return journey to the depot. This distinction is important to properly model the delivery routes that start and end at a specific location. In consideration for possible improvements, an indexClosestTo class was added to determine which address in our addressList would be closest to the address passed in. This was crucial when implementing our greedy heuristic since connecting nodes intelligently is a crucial aspect of the Traveling Salesman Problem (TSP).

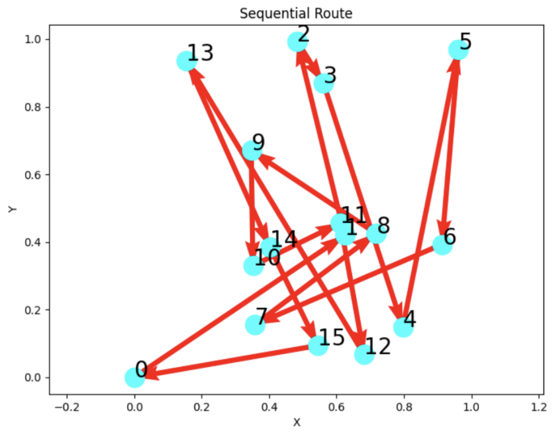

Figure 3: Simple route traversing through each address in order. Length of route: 8.67616

3.3 Step 3: Finding an Initial Route (Greedy Algorithm)

Initially, a greedy algorithm was considered to reduce distance. This method works by starting at the depot and repeatedly chooses the nearest unvisited address until all addresses are visited. This method performs better than randomly selecting routes because it takes into consideration the next best path for a truck to take, decreasing distance over small intervals. While fast, this approach often produces routes that are far from optimal due to not considering how local decisions affect the overall route.

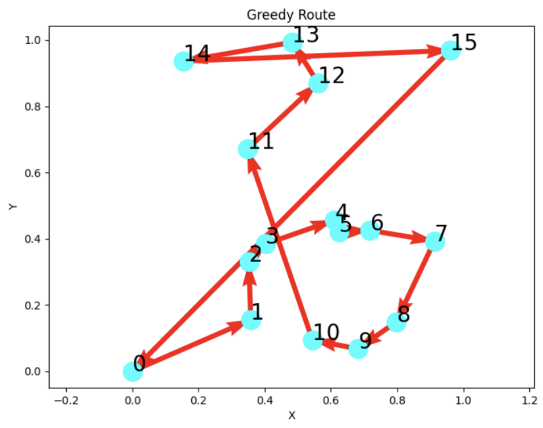

Figure 4: Greedy Route traversing through each address. Length of Greedy route: 5.29442

3.4 Step 4: Improving the Route (Opt2 Optimization)

After creating an initial route, a second more efficient algorithm was considered: the opt2 algorithm. The opt2 algorithm improves the route by taking pairs of edges to determine if the total distance can be shortened. If a swap improves the route, the algorithm keeps it and continues improving. This process continues until no more improvements can be made.

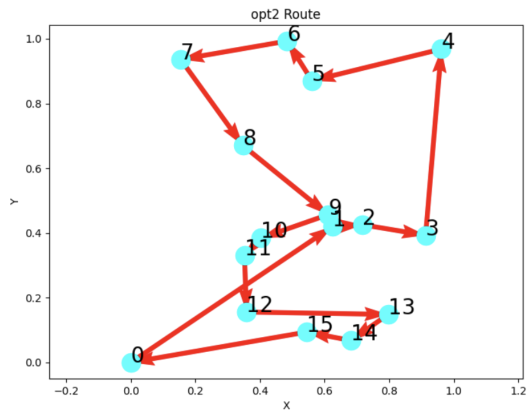

Figure 5: opt2 Route traversing through each address. Length of opt2 route: 4.92007

3.5 Step 5: Handling Multiple Trucks

In real life, Amazon uses multiple trucks for deliveries with each truck assigned their own list of addresses to visit. To simulate this scenario, the list of addresses was divided among multiple trucks, and each truck's route was optimized using a variation of the opt2 algorithm to ensure the delivery paths were more efficient. Each truck's route was optimized using a variation of the opt2 algorithm to ensure efficient delivery sequences.

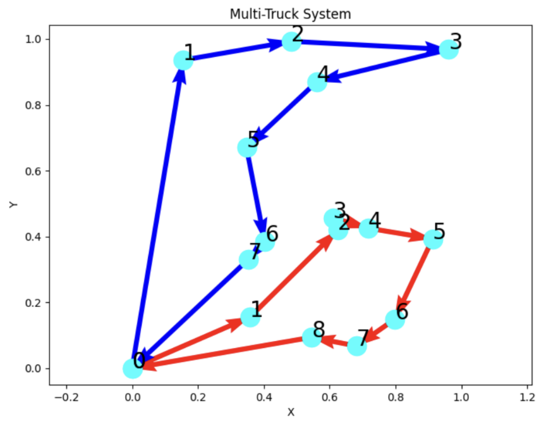

Figure 6: Multiple Trucks Route traversing through each address. Length of total route: 5.53163

When multiple trucks have overlapping routes, there is often unnecessary travel caused by the redundant pathing. This inefficiency could be reduced by optimizing the intersections. Hence the opt2 algorithm is used to exchange delivery addresses between intersecting truck routes to reduce these overlaps. The algorithm focuses on swapping two addresses within the two intersecting paths, ensuring that the trucks visit the closest possible sequence of addresses. This ensures that the delivery paths are optimized not just individually, but also in relation to each other, reducing the total distance traveled across the fleet of trucks.
This approach significantly improves the route, due to it eliminating redundant segments and minimizing the distance traveled by each truck which ultimately saves on costs and time.

3.6 Step 6: Prime Delivery Constraints

To accommodate Prime deliveries, a constraint was introduced where Prime addresses could only be assigned to specific trucks. This restriction reduced optimization flexibility by preventing routes to be shared between Prime and non-Prime trucks. This lead to longer overall travel distances making it more challenging to efficiently handle multiple deliveries while meeting the Prime requirements. To implement this solution, we applied the greedy and opt2 algorithms separately to each path, as the independence of Prime and non-Prime routes required treating them as distinct optimization problems.

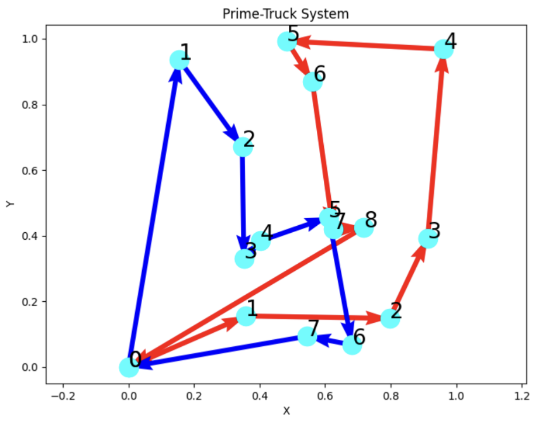

Figure 7: Prime Trucks (blue) and Non-Prime Trucks (red) Route traversing through each address. Length of total route: 6.68279

3.7 Step 7: Friendly User Interface

Finally, in order to enhance the user experience, the code allows users to choose how they want to input delivery addresses. They can manually enter addresses, generate a random set for testing purposes, or upload a file containing the addresses. There is also an option for administrators to use predefined sets of addresses for testing. By having a variety of input options, we ensure that the program can cater to different user needs.

4 Performance Analysis

In order to gauge the effectiveness of both situations, it's helpful to run a Performance Analysis to determine the run time efficiency of the Greedy Algorithm and the Opt2 Algorithm.

1. **Greedy Algorithm**: The greedy algorithm runs in $O(n^2)$ time, where n is the number of addresses in the list. This means that as the number of addresses increases, the number of distance calculations grows quadratically, meaning the time to solve the problem will increase rapidly for larger datasets. This occurs because the greedy algorithm iteratively determines the closest address for every address, leading to a double nested for loop.

2. **Opt2 Algorithm**: The opt2 algorithm runs in $O(n^3)$ time, where n is the number of addresses in the list. This means that as the number of addresses increases, the number of distance calculations grows cubically, meaning the time to solve the problem will increase rapidly for larger

datasets. This occurs because each pair of addresses needs to be compared to determine if a swap is necessary for all addresses. As a result, opt2 tends to be more computationally expensive than the greedy algorithm, especially for larger datasets.

N	Greedy Algorithm Time (ms)	Opt2 Algorithm Time (ms)
100	0	2
250	1	43
500	5	167
1000	22	522
5000	539	15018
10000	2154	98107

Table 1: Greedy and Opt2 algorithm times for various values of N

The data highlights that the execution time of the opt2 algorithm tends to increase more significantly than that of the greedy algorithm's. Also for large values of N, opt2 has a significantly greater execution time compared to the greedy algorithm. This suggests that while opt2 may produce more accurate results in the Traveling Salesman Problem (TSP), its computational inefficiency makes it less viable for large datasets. In contrast, while the greedy algorithm yields less optimal results, it does have faster execution times as the data size scales.
Implementing a multi-truck system adds more complexity to the scenario since we have to not only minimize each truck's individual path but also allow for each truck to have optimal routes compared to other trucks. This significantly raises the computational complexity to $O(n^4)$ due to the increased the number of possible solutions that need to be evaluated.
Prime deliveries further complicate this task by introducing additional constraints. These constraints are just as computationally demanding as the multi-truck systems, although with the added benefit that Prime trucks do not swap addresses with non-Prime trucks. However, it must find ways to fit both regular and Prime deliveries into the schedules, all while trying to minimize the total distance traveled. This introduces additional complexity, as the algorithm must carefully balance the needs of different deliveries to keep the routing efficient without compromising on delivery times or priorities.

5 Real World Considerations

5.1 Realistic Route Sizes

A typical Amazon delivery driver handles 250 to 300 packages per day. The simulations demonstrated that the opt2 heuristic performs well for optimizing routes with up to 200 addresses but becomes increasingly slower for larger datasets, highlighting its computational limits. As the number of addresses grows, the algorithm's runtime makes it less practical for large-scale applications. However, both the opt2 and greedy algorithms are better compared to exact solutions, making them more viable for real-world scenarios where efficiency and scalability are critical.

5.2 Ethical Considerations

While TSP optimization can reduce fuel consumption and delivery times, it may also inadvertently increase driver workload. For instance, routes with over 150 stops often leave drivers with insufficient time for breaks, causing a feeling of unrealistic delivery targets and tight deadlines, reflecting broader criticisms of Amazon's labor practices.
In multi-truck scenarios, algorithms assign addresses to different trucks, but this can result in an uneven distribution of tasks. By prioritizing proximity, one driver might receive a disproportionate number of stops in less desirable or unsafe areas, further highlighting disparities in workload. Moving forward, optimizing delivery algorithms should account not only for efficiency but also for fairness and driver well-being. This would enable equitable work distribution, enhance safety, and maintain high-quality service.

6 Conclusion

This project explored the Amazon Delivery Problem, a real-world example of the Traveling Salesman Problem (TSP), where the objective is to optimize delivery routes for multiple addresses. Two heuristic algorithms, the greedy algorithm and the Opt2 algorithm, were implemented and examined to determine their effectiveness in tackling the TSP. The greedy algorithm provided faster solutions but sacrificed optimality, while the opt2 algorithm offered better results at the cost of higher computational expense. The analysis also considered the implications of multi-truck routes, and prime deliveries, which added an extra layer of complexity. Finally, the paper ended with a discussion of the ethical implications at play when optimizing delivery times ended up compromising worker well-being.

6.1 Looking Ahead

One possible suggestion for improvement could be implementing a scenario where every day a random number of new deliveries is added to the list. At Amazon, new deliveries have requirements to be delivered at different dates, and being able to update timely orders is critical. This would cause the algorithm to need to adapt to the changing input by constantly updating itself whenever new deliveries come in, ensuring that all new deliveries are prioritized according to their deadlines. As a result, this would introduce more complexity into the routing process, as the system would need to consider both new and existing

deliveries while maintaining efficiency. One approach might be to use a rolling window model that discards old data while efficiently inputting new data. This way the algorithm considers a fixed set of deliveries within a given timeframe before re-calculating the most efficient routes.
Another possible suggestion would be to consider varying traffic times between paths. Since in the real world, travel times are not consistent, being able to change the model as a result of road congestion would help handle unexpected delays. This approach would require the system to consider weights on edges in the route graph based on travel times. By implmenting this situation, it could lead to better time predictions for deliveries which in turn would lead to improved customer satisfaction and operational efficiency.

7 References

- Amazon Logistics Statistics (2024): Number of package deliveries. Capital One Shopping. (2024, October 8). https://capitaloneshopping.com/research/amazon-logistics-statistics/#:~:text=Amazon%20DSP%20drivers%20each%20deliver,2%20minutes%20and%2015%20seconds.

- Eijkhout, V. (2017–2022). *Introduction to scientific programming in C++17/Fortran2008: The Art of HPC, volume 3* (formatted August 19, 2024). Retrieved from https://tinyurl.com/vle322course

- Overcoming the rest gap: Why drivers fail to take sufficient breaks and how to address it. Descartes SmartCompliance. (2023, November 22). Retrieved from https://smartcompliance.descartes.com/resources/news-blog/blog/overcoming-the-rest-gap-why-drivers-fail-to-take-sufficient-breaks-and-

- Traveling salesman problem. Traveling salesman problem - Cornell University Computational Optimization Open Textbook - Optimization Wiki. (n.d.). Retrieved from https://optimization.cbe.cornell.edu/index.php?title=Traveling_salesman_problem

8 Appendix

8.1 Euclidean Distance

```
    double distance_euc(Address a) {
        double delx = abs(i - a.geti());
        double dely = abs(j - a.getj());
        return sqrt(pow(delx, 2) + pow(dely, 2));
    }
```

8.2 Address List Class

```cpp
class AddressList { // vector
protected:
    vector<Address> list = {}; // Stores a list of Address objects
public:
// Constructor to initialize the list with an optional vector of addresses
    AddressList(vector<Address> field) : list(field) {};
    AddressList() {};

// Add a unique address to the list
    void add_address(Address add) {
        for (int i = 0; i < list.size(); i++) {
            if (list[i].geti() == add.geti() && list[i].getj() == add.getj()) {
                return;
            }
        }
        list.push_back(add);
    }

// Add an address to a specific position (used for reinserting)
    void add_address(Address add, int pos) {
        if (pos < 0 || pos > list.size()) throw std::out_of_range("Position is out of bounds.");
        list.insert(list.begin() + pos, add);
        // This function doesn't need a duplicacy check because at this point all addresses are unique
    }

// Remove an address from the list (done temporarily to try new routes)
    void remove_address(int k) {
        if (k < 0 || k >= list.size()) return;
        list.erase(list.begin() + k);
    }

// Get an address at a specific index, with bounds checking
    Address get_address(int index) {
        if (index < 0 || index >= list.size()) {
            throw std::out_of_range("Index is out of bounds.");
        }
        return list[index];
    }
// Return the number of addresses in the list
    int num_of_address() {
        return list.size();
    }
// Calculate the total Euclidean distance of the route
    double length() {
        if (list.empty()) return 0;
        double totLength = 0;
        for (int i = 1; i < list.size(); i++) {
            totLength += list[i-1].distance_euc(list[i]);
        }
        return totLength;
    }

//converts the address list to a string representation in format (i, j)
    string as_string() {
        string s = "";
        for (int i = 0; i < list.size(); i++) {
            s += "(" + to_string(list[i].geti()) + "," + to_string(list[i].getj()) + ") ";
        }
        return s;
    }

//converts the address list to a string representation in format i j (with line breaks)
    string as_string2() {
        string s = "";
        for (int i = 0; i < list.size(); i++) {
            s += to_string(list[i].geti()) + " " + to_string(list[i].getj()) + "\n";
        }
        return s;
    }
} // end class
```

8.3 Index Closest To Method

```cpp
// Within the AddressList class
    Address index_closest_to(Address address) {
        if (list.empty()) {
            throw runtime_error("Address list is empty.");
        }

        Address closestAddress = list[0];
        double minDistance = address.distance_euc(list[0]);

        for (Address& addr : list) {
            double currentDistance = address.distance_euc(addr);
            if (currentDistance < minDistance) {
                minDistance = currentDistance;
                closestAddress = addr;
            }
        }
        return closestAddress;
    }
```

8.4 Greedy Algorithm

```cpp
// Within the AddressList Class
vector<Address> greedy_list() {
        Address we_are_here = Address(0, 0);
        vector<Address> new_list = {};
        vector<Address> curr_list = list;

        double distance_euc(Address a) {
            double delx = abs(i - a.geti());
            double dely = abs(j - a.getj());
            return sqrt(pow(delx, 2) + pow(dely, 2));
    }

        while (!list.empty()) {
            we_are_here = index_closest_to(we_are_here);
            for (int i = 0; i < list.size(); i++) {
                if (we_are_here.equals(list[i])) {
                    list.erase(list.begin() + i);
                    break;
                }
            }
            new_list.push_back(we_are_here);
        }

        list = curr_list;
        return new_list;
    }
```

8.5 Opt2 Algorithm

```
void opt2(Route& route) {
    if (route.num_of_address() < 4) return;
    bool improvement = true;

    while (improvement) {
        improvement = false;

        for (size_t m1 = 1; m1 < route.num_of_address() - 2; m1++) {
            for (size_t n1 = m1 + 1; n1 < route.num_of_address() - 1; n1++) {
                double origDist = route.get_address(m1 - 1).distance_euc(route.get_address(m1)) +
                                  route.get_address(n1).distance_euc(route.get_address(n1 + 1));

                double newDist = route.get_address(m1 - 1).distance_euc(route.get_address(n1)) +
                                 route.get_address(m1).distance_euc(route.get_address(n1 + 1));

                if (newDist < origDist) {
                    reverse(route.get_list().begin() + m1, route.get_list().begin() + n1 + 1);
                    improvement = true;
                }
            }
        }
    }
}

// Single function to run opt2 on two different routes
void multipath_opt2(Route& r1, Route& r2) {
    opt2(r1);
    opt2(r2);
}
```

8.6 Opt2 Algorithm for Multiple Trucks

```
//performs 2-opt on two routes, but factors in optimization by transferring addresses between the routes
void multipath_opt2_switching(Route& r1, Route& r2) {
    bool improvement = true;
    int max_it = 1000;
    int it = 0;

    while (improvement && (it < max_it)) {
        improvement = false;
        multipath_opt2(r1, r2); // Optimize by reversing segments independently first
        double originalTotalLength = r1.length() + r2.length();

        Route prev_r1 = r1;
        Route prev_r2 = r2;

        // Switch addresses from r2 to r1
        for (int i = 1; i < r2.num_of_address() - 1; i++) {
            Address addr = r2.get_address(i);
            AddressList list2 = r2;
            list2.remove_address(i);

            for (int j = 1; j <= r1.num_of_address() - 1; j++) {
                AddressList list1 = r1;
                list1.add_address(addr, j);

                double newTotalLength = list1.length() + list2.length();
                if (newTotalLength < originalTotalLength) {
                    r1 = Route(list1.get_list());
                    r2 = Route(list2.get_list());
                    multipath_opt2(r1, r2);
                    improvement = true;
                    originalTotalLength = newTotalLength;
                    break;
                }
            }
            if (improvement) break;
        }

        // Switch addresses from r1 to r2
        for (int i = 1; i < r1.num_of_address() - 1; i++) {
            Address addr = r1.get_address(i);
            AddressList list1 = r1;
            list1.remove_address(i);

            for (int j = 1; j <= r2.num_of_address() - 1; j++) {
                AddressList list2 = r2;
                list2.add_address(addr, j);

                double newTotalLength = list1.length() + list2.length();
                if (newTotalLength < originalTotalLength) {
                    r1 = Route(list1.get_list());
                    r2 = Route(list2.get_list());
                    multipath_opt2(r1, r2);
                    improvement = true;
                    originalTotalLength = newTotalLength;
                    break;
                }
            }
            if (improvement) break;
        }

        // Break if no meaningful change in routes
        if (r1.as_string() == prev_r1.as_string() && r2.as_string() == prev_r2.as_string()) {
            break;
        }
        ++it;
    }
    multipath_opt2(r1, r2); //final reordering of segments for optimal performance
}
```

```
//In the AddressList Class
    double length() {
        if (list.empty()) return 0;
        double totLength = 0;
        for (int i = 1; i < list.size(); i++) {
            totLength += list[i-1].distance_euc(list[i]);
        }
        return totLength;
    }
```

8.7 Performance Analysis Code

```
vector<int> sizes = {100, 250, 500, 1000, 5000, 10000};
    for (auto N : sizes) {
        Route addrList;
        //add addresses
        for (int i = 0; i < N; i++) {
            addrList.add_address(Address((double) rand() / (RAND_MAX), (double) rand() / (RAND_MAX)));
        }
        // Timing Greedy Algorithm
        auto start = chrono::high_resolution_clock::now();
        addrList.greedy_route();
        auto end = chrono::high_resolution_clock::now();
        auto duration = chrono::duration_cast<std::chrono::milliseconds>(end - start);
        cout << "Greedy algorithm time for N=" << N << ": " << duration.count() << " ms\n";

        // Timing Opt2 Algorithm
        start = chrono::high_resolution_clock::now();
        addrList.opt2();
        end = chrono::high_resolution_clock::now();
        duration = chrono::duration_cast<chrono::milliseconds>(end - start);
        cout << "Opt2 algorithm time for N=" << N << ": " << duration.count() << " ms\n";
    }
```