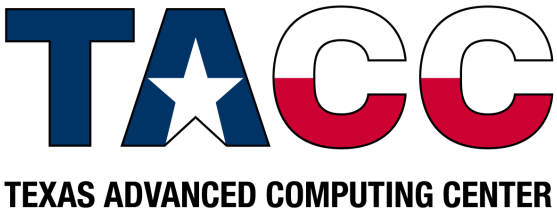

Amazon Delivery Scheduling: The Traveling Salesman Problem

Authors:

Atharva Kalamkar
EID: amk4934
atharva.m.kalamkar@gmail.com
TACC: amk4934

William Zhang
EID: wz4944
williamzhang@utexas.edu
TACC: williamzhang

Jacob Hands
EID: jmh8598
JacobHands77@gmail.com
TACC: jacobhands

Date: 12/05/2023

1 Introduction

1.1 The Problem

The Traveling Salesman Problem (TSP) is a very common problem faced by companies that provide any sort of delivery services. Although optimizing the route of delivery has always been a crucial aspect of efficient logistics, the contemporary landscape, specifically in the context of Amazon delivery truck scheduling, introduces novel considerations.

1.2 The Approach

This project revolves around addressing the unique problems posed by Amazon's delivery commitments to its customers. For example, instead of a full sweep of a delivery list, due to Amazon promising a window of days for their deliveries, these lists often need to be split up into sublists resulting in a shorter total distance than a continuous sweep through the entire list. Most importantly, Amazon Prime adds a whole new caveat to TSP since customers expect deliveries at their doorstep the next day.

This paper sheds light on innovative strategies aimed at meeting the evolving demands of modern delivery services, particularly in the context of the time-sensitive Amazon Prime delivery model. One finding of our study reveals that the implementation of a specific optimization approach resulted in a substantial reduction in the overall travel distance covered by Amazon delivery trucks. The paper not only demonstrates how we came up with different route optimization techniques, optimizing multiple routes, and adding the constraint of Prime delivery, but also addresses the ethics of the problem and future improvements to our approach.

2 Methods and Processes

2.1 Setup

To start the project, an `Address` class was defined with doubles `i` and `j` representing the coordinates of the address (i,j), a string `last_possible_delivery_date` to denote the last possible delivery date of the item at that address, and a boolean `prime_member` to represent the customer's Prime membership status. Within the address class, a `distance()` method was implemented as shown below.

```
double distance(const Address& additional_addresses, bool manhattan_distance = true) {
    if (manhattan_distance) {
        return std::abs(i - additional_addresses.i) + std::abs(j - additional_addresses.j);
    } else {
        return std::sqrt(std::pow(i - additional_addresses.i, 2) +
                         std::pow(j - additional_addresses.j, 2));
    }
}
```

This method allows us to choose and test using either the Manhattan distance or straight-line method to calculate the distance between two addresses. Next, an `AddressList` class was defined with a single vector `address_list`. This can be used to store all the addresses within a route. Within `AddressList`, there are three methods: `add_address()` to add an address to `address_list`, `length()` to calculate the total distance to visit all addresses in order, and `index_closest_to()` to find the index of the address in the list closest to `target_address`, which is an `Address` passed as a parameter.

Finally, the `Route` class inherits from `AddressList` and adds two (0,0) addresses at the beginning and end of the `address_list` vector to symbolize a depot. Within `Route` are all the algorithms we used to optimize the routes. We used brute force, greedy route, 2-Opt, and multiple path 2-Opt algorithms in this class. The 2-Opt methods were then further improved to include constraint of Prime delivery, checking the reversibility of the routes, and dynamism.

2.2 Brute Force Searching

One approach to finding the shortest route is to simply check all possible routes and calculate the length of each route. This amounts to searching all permutations of address_list[1 to n-1] for the shortest route (we have to exclude permutations involving the start/end points of the route.)

This algorithm starts by initializing the solution `best_route`, as the original route, as well as the distance of the shortest route, `best_length`, as the length of the original route. Next, for each possible permutation of the addresses between the depots, calculate the distance of this new route. If the length of the current route, `test_length`, is less than `best_length`, then the current route is the most optimal route found so far.

```
std::vector<Address> best_route = address_list; // Store the initial route as the best
double best_length = length(); // Calculate its length
// Copy the initial order of addresses for permutation
std::vector<Address> original_order = address_list;
// Find permutations and track the best route and length
sort(address_list.begin() + 1, address_list.end() - 1);
do {
    double test_length = length(); // Calculate the length for the current permutation
    if (test_length < best_length) {
        best_length = test_length; // Update the best length
        best_route = address_list; // Update the best route
    }
    // Exclude the depot addresses
} while (std::next_permutation(address_list.begin() + 1, address_list.end() - 1));
// Update the address_list to the best route found
address_list = best_route;
```

The number of possible routes to search is $(n - 2)!$, the number of permutations of addresses excluding the start and end depots. Furthermore, at each new route, calculating the length of the route takes $O(n)$ time. So the time complexity of a brute force search is approximately $O(n \times (n - 2)!)$. Clearly, this is a very inefficient method and can take large amounts of time to compute for routes with more addresses, making it unrealistic for a real world scenario. Unfortunately, the Traveling Salesman Problem has been shown to be in the class of NP-complete problems. Informally, that means it's unlikely a more efficient solution with polynomial time complexity exists.

2.3 Heuristics

While the brute force solution to the Traveling Salesman Problem is computationally expensive, oftentimes the absolute shortest path is not necessary. For many applications including Amazon deliveries, approximately short routes are acceptable and more realistic. To this end, we can use heuristics, imperfect but sufficient methods, to find optimized solutions in a reasonable amount of time.

2.3.1 Greedy

One such heuristic is to always travel to the nearest unvisited address. This method is considered “greedy” because it chooses the best case locally without any consideration to the route as a whole. The route starts at the depot and chooses the nearest address as the next address. Then from that address it chooses the nearest unvisited addresses as the next address, and so on.

In our implementation we start with a list of known addresses, `address_list`, and an empty route, `new_route`, that starts at the depot. We also initialized `current_address`, which represents the last address in `new_route` as it is constructed. Then we exclude the remaining depots from `address_list`. While there are still addresses to visit (`address_list` has more than one address), we mark `current_address` as visited by removing it from `address_list`. We search `address_list` for the address nearest to `current_address` and set it as next_address. `next_address` is added to the route and `current_address` is updated to `next_address`.

```
// initialize a new route starting at the depot
Route new_route;
Address current_address = address_list[0];
// remove the last stop (the depot)
address_list.pop_back();
// At the end address list will only contain the next address, which is added to new_route
while (address_list.size() > 1) {
    // remove current_address from address_list
    for (int i = 0; i < address_list.size(); i++){
        if (address_list[i] == current_address){
             address_list.erase(address_list.begin()+i);
                 break;
             }
        }
    // the next address in the route is the address closest to current
    Address next_address = index_closest_to(current_address);
    new_route.add_address(next_address);
    current_address = next_address;
    }
}
```

2.3.2 2-Opt

The next approach we took in optimizing the TSP was developing a 2-Opt heuristic method. This method is a local search algorithm that iteratively improves a given route by swapping edges within a current route, and checking if the total length traveled is shorter than

the original. The intuition behind swapping edges in a route is that when a route ‘crosses’ itself, the route can be shortened by ‘uncrossing’ it as shown in Figure 1.

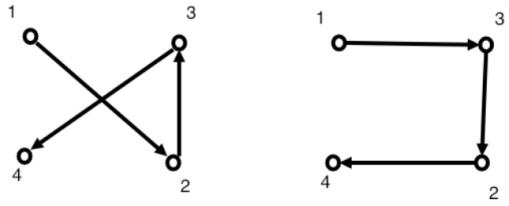

Figure 1. 2-Opt Example

In the apply_2_opt() method, we iterate throughout the list while not accounting for the first and last positions because they are constant predetermined locations in all routes. We then check if by switching the next destination to a position 2 destinations ahead of the original, gives us a shorter distance. If the switch decreases total length traveled using the manhattan method, we switch the 2 positions.

```
bool improvement = true;
while (improvement) {
    improvement = false;
    for (int i = 1; i < address_list.size() - 2; ++i) {
        for (int j = i + 1; j < address_list.size() - 1; ++j) {
            // Check if the new connection is shorter
            double delta_distance = address_list[i - 1].distance(address_list[j]) +
                                    address_list[i].distance(address_list[j + 1]) -
                                    address_list[i - 1].distance(address_list[i]) -
                                    address_list[j].distance(address_list[j + 1]);
            if (delta_distance < 0) {
                // If the new connection is shorter, reverse the portion of the tour
                reverse(address_list.begin() + i, address_list.begin() + j + 1);
                improvement = true;

            }
        }
    }
}
```

The nested for loops that selects a pair of edges in the route each run roughly $O(n)$ iterations. Additionally the reverse operation and distance calculations both have constant time complexity (we are using the lengths of the edges instead of the length of the whole route). For a single pass of the 2-Opt heuristic, the time complexity is approximately $O(n^2)$. In our implementation, the 2-Opt heuristic is re-run continuously until no further improvement is found.

However, this heuristic is often applied a pre-set number of times, which can be especially useful for large numbers of addresses.

2.4 Two Trucks

The largest dilemma with TSP is when you have multiple salesmen navigating around similar points. If you were to use a greedy or opt2 method alone, there would be multiple segments crossing. These crossings are inefficient to the business, and so by swapping segments of multiple opt2 optimized routes, we are able to optimize the paths even more. Through this designed method, we iteratively check through all potential segment swaps and choose the best segment to maintain in the route. In addition, due to Amazon's guarantee of next day shipping, we designed a boolean case that checks addresses for Prime membership, and if they are Prime members, their address cannot be exchanged with another route to fulfill the guarantee.

```
// Similar to 2-Opt...
while (improvement) {
    improvement = false;
    // select any two segments both routes
    for (int i = 1; i < rt1.address_list.size() - 2; ++i) {
        for (int j = 1; j < rt2.address_list.size() - 2; ++j){
            // Don't swap if any of the addresses are amazon prime (have to stay on the same route)
            if (either address in segment is prime){
                continue;
            }
            double initial_length = rt1.length() + rt2.length();
            // try swapping segments - flip neither segment
            std::swap(rt1.address_list[i], rt2.address_list[j]);
            std::swap(rt1.address_list[i+1], rt2.address_list[j+1]);
            // keep the swap and flag improvement if swap decreases length
            if ((rt1.length() + rt2.length()) < initial_length){
                improvement = true;
                continue;
            }
            // otherwise swap back
            else{
                std::swap(rt1.address_list[i], rt2.address_list[j]);
                std::swap(rt1.address_list[i+1], rt2.address_list[j+1]);
            }
            // same code but flip both segments...
        }
    }
}
```

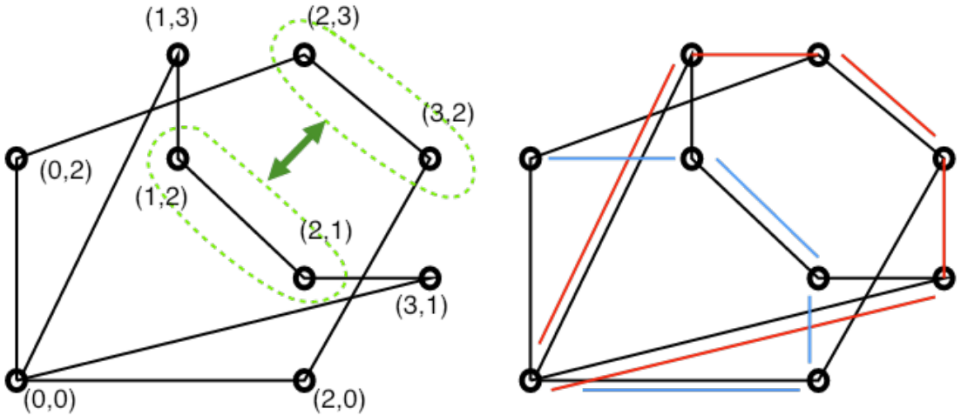

Figure 2. Multiple Paths Example

2.5 Amazon Prime

One service that Amazon offers is Amazon Prime, this guarantees customers next day or 2-day shipping. With the Amazon guarantee, we would need to check if an address on a route has Prime membership or not to determine if their address can be exchanged with another route. This stipulation leads to a boolean being added in as a part of an individual's address. We then would implement a check using method `is_prime_member()` to determine if an address can be swapped in the `multip_path_apply_2_opt()`. Due to the Prime guarantee, any address found to be a Prime member cannot be swapped.

```
std::vector<Route> multi_path_apply_2_opt(Route &rt1, Route &rt2) {
    //previous code
    // If any member in the address swap is a prime member do not exchange
    if (rt1.address_list[i].is_prime_member() or
        rt1.address_list[i+1].is_prime_member() or
        rt2.address_list[j].is_prime_member() or
        rt2.address_list[j+1].is_prime_member()) {
            Continue;
    }
}
```

2.6 Generalizing to Multiple Trucks

The `multip_path_apply_2_opt()` is designed to optimize the routes of 2 trucks, but for a business like Amazon, there are tens of drivers around the same area. In order to generalize the `multip_path_apply_2_opt()` case of n trucks, we developed an algorithm to check every surrounding driver and their opt2 routes. We would then apply `multip_path_apply_2_opt()` to the two routes and save the best outcome for that segment swap between the routes. This

process would go on for n number of drivers and afterwards, we check which `multip_path_apply_2_opt()` segment swap had the best results for both drivers and make the finalized swap there. It is a iterative algorithm that generalizes the issue of having n number of drivers.

```
class MultiRoute{
      // constructor: if vector of routes is known
      MultiRoute(vector<Route> routes):
          num_routes(routes.size()), all_routes(routes) {}
      // constructor if only addresses are known
      MultiRoute(int n, vector<Address> addresses){}
void optimize_routes(){
    // keep optimizing until no more improvement
    double initial_length;
    bool improvement = true;
    int n = 0;
    while (improvement == true){
        n++;
        improvement = false;
        initial_length = total_length();
        // apply 2 opt individually
         for (int i = 0; i < num_routes ; i++){
              all_routes[i].apply_2_opt();
         }
         // apply 2 opt between all routes
         for (int i = 0; i < num_routes - 1; i++){
             for (int j = i+1 ; j < num_routes; j++){
                  Route rt1 = all_routes[i];
                  Route rt2 = all_routes[j];
                  vector<Route> new_routes =
                  multi_path_apply_2_opt(rt1,rt2);
                  all_routes[i] = new_routes[0];
                  all_routes[j] = new_routes[1];
              }
         }
    }
}
```

3 Results and Experiments

3.1 Single Route Algorithms

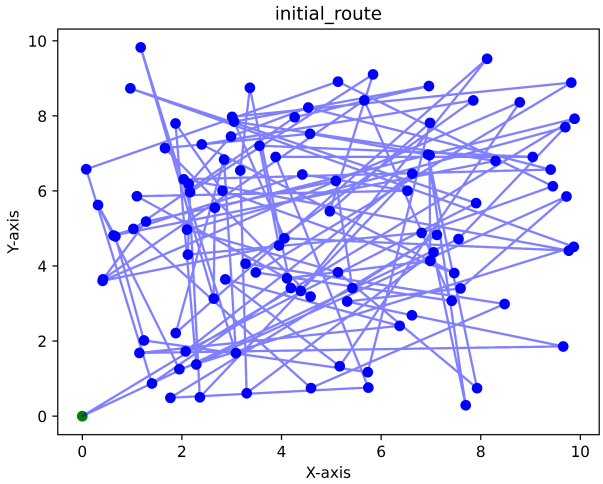

Figure 3. Initial Route

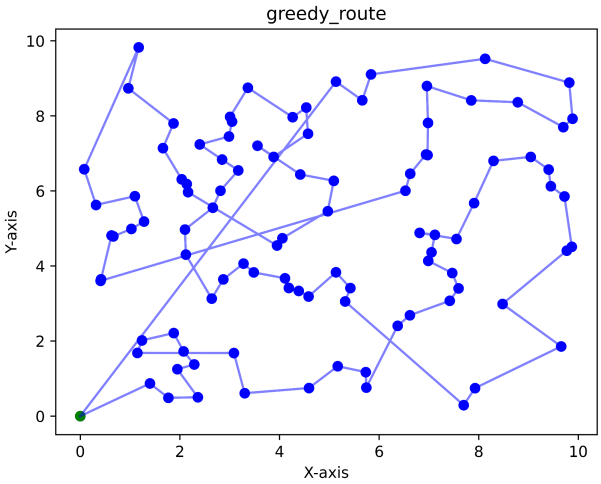

Figure 4. Greedy Route

In Figures 3 and 4, we see the visualization of a test trial run on one delivery driver taking the greedy route with 50 addresses to visit with random coordinates between 1 and 10 per address. For the initial route, the distance traveled was 512.277, and the greedy-optimized route reduced the distance to 97.2559, which is a significant improvement in the distance traveled by the driver. We can see how starting at (0,0), which represents the depot, the truck continuously chooses the nearest address to the one that it is currently at. This algorithm works quite well in decreasing the distance traveled, however we can observe that the last commute on the way back to the depot is relatively long. Also, there are several instances where the driver crosses paths with itself, and as discussed in the 2-Opt section, this “crossing of paths” is not ideal and can be further optimized. This particular limitation of the greedy route is addressed by the 2-Opt algorithm.

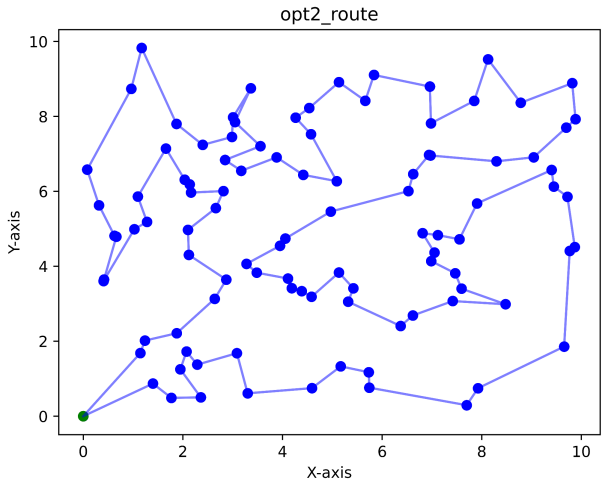

Figure 5. 2-Opt Route

Figure 5 shows a test trial run on one delivery driver visiting the same 50 random addresses from the initial route shown in Figure 3, however in this case, the driver uses the 2-Opt technique to visit each address. Using the 2-opt route brought the total distance traveled by the truck down to 83.4305, which is even more of an improvement of the initial route than the greedy method. In this route, we can notice how there are no edges that cross each other and hence create a more optimal route than the initial.

Overall, we see that taking the greedy approach to the TSP leads to an undesirable routing with the amount of crossing and total distance traveled. When applying the opt2 method, our route is far easier to navigate for a driver and leads to a decrease in total distance traveled. In Table 1, a comparative analysis of route lengths is presented, showcasing the results obtained using the greedy and 2-Opt optimization techniques. Also, the table includes the optimal route length determined using the brute force method. We generated 10 random addresses each of the five trials, and we can notice how for each trial the length of the route using the 2-Opt technique gets very close to if not is equal to the length of the best possible route, while the greedy route is consistently less optimal than 2-Opt.

	Initial Route	Greedy	2-Opt	Best (Brute Force)
Trial 1	64.521	36.0774	29.4218	28.6255
Trial 2	67.5995	32.7834	30.9488	30.9488
Trial 3	76.1604	42.9661	33.3699	33.3699
Trial 4	81.7545	39.854	35.1681	35.1681
Trial 5	50.8843	32.1105	31.7267	31.7267

Table 1: Distance Traveled For Each Optimization Method

3.2 Multiple Route Algorithms

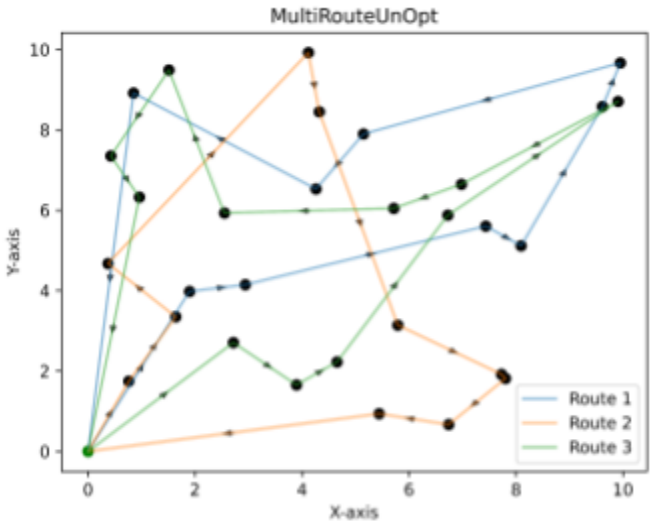

Figure 6. Initial Routes

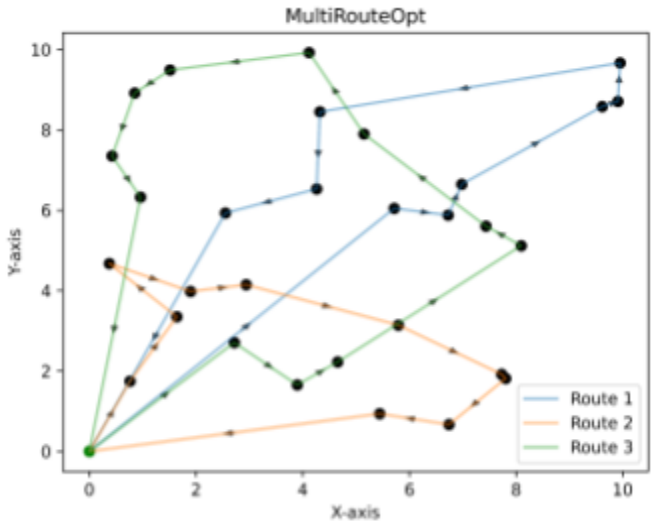

Figure 7. Multipath 2-Opt Routes

The 2-Opt technique discussed in the previous section can be generalized to a scenario with multiple trucks. Right now, we can pretend these are either three different drivers traveling on the same day or the same driver going through three different routes on separate days. Regardless, we can see in Figure 6 how although each of the three routes are already optimized themselves with the opt2 method, checking if swapping segments with other routes as shown in Figure 7 does lead to a more optimized route for all drivers. The lengths of the routes shown in

Figure 6 are 35.7966, 29.841, 36.6243 respectively for a total length of 172.096, and the lengths shown in Figure 7 are 30.6496, 22.151, 29.8823 respectively for a total length of 82.683.

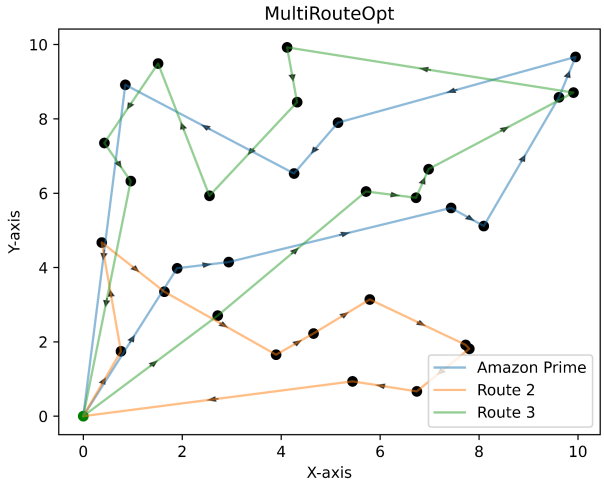

Figure 8. Multipath 2-Opt Routes With Prime

Lastly, we have the implementation of the desired Amazon Prime member constraint. In this case, we have three different delivery days; one for the next-day delivery promised to Amazon Prime members and two for the next two promised delivery dates. The same initial, individual 2-Opt optimized routes from Figure 6 were used for this test case. However, as shown in Figure 8, the blue line — represented by Route 1 in Figure 6 — is a route in which all the addresses are Amazon Prime members, while Route 2 and Route 3 have no Prime members. The lengths of the routes shown in Figure 8 are 35.7966, 22.7558, 37.8787 respectively for a total length of 96.4311.

We can see that the implementation is working as the initial route for the blue line crosses through the same points in the same order it was initially assigned. Also, as expected, the total length of the routes increased from the multipath 2-Opt routes without Prime due to the entirety of the first route having a strict next-day delivery date. However, with this method, the Prime members were able to receive their packages in a timely manner, and the next two days of routes were optimized so that the combined length of the routes of those days decreased from 66.4653 to 60.6345, or by 8.8%.

3.3 Comparison of Brute Force and 2-Opt Runtimes

In this experiment we compare the average run times of the 2-Opt heuristic as well as the Brute Force method for a range of values of n, the number of addresses in a route. For each value of $n$tested, ten routes with n addresses ($n - 2$ randomly generated addresses plus the two depots) were constructed. Then each method is tested on the route and its run time is recorded.

For the 2-Opt heuristic we ran this experiment with values of n ranging from 10 to 400, incrementing n by 10.

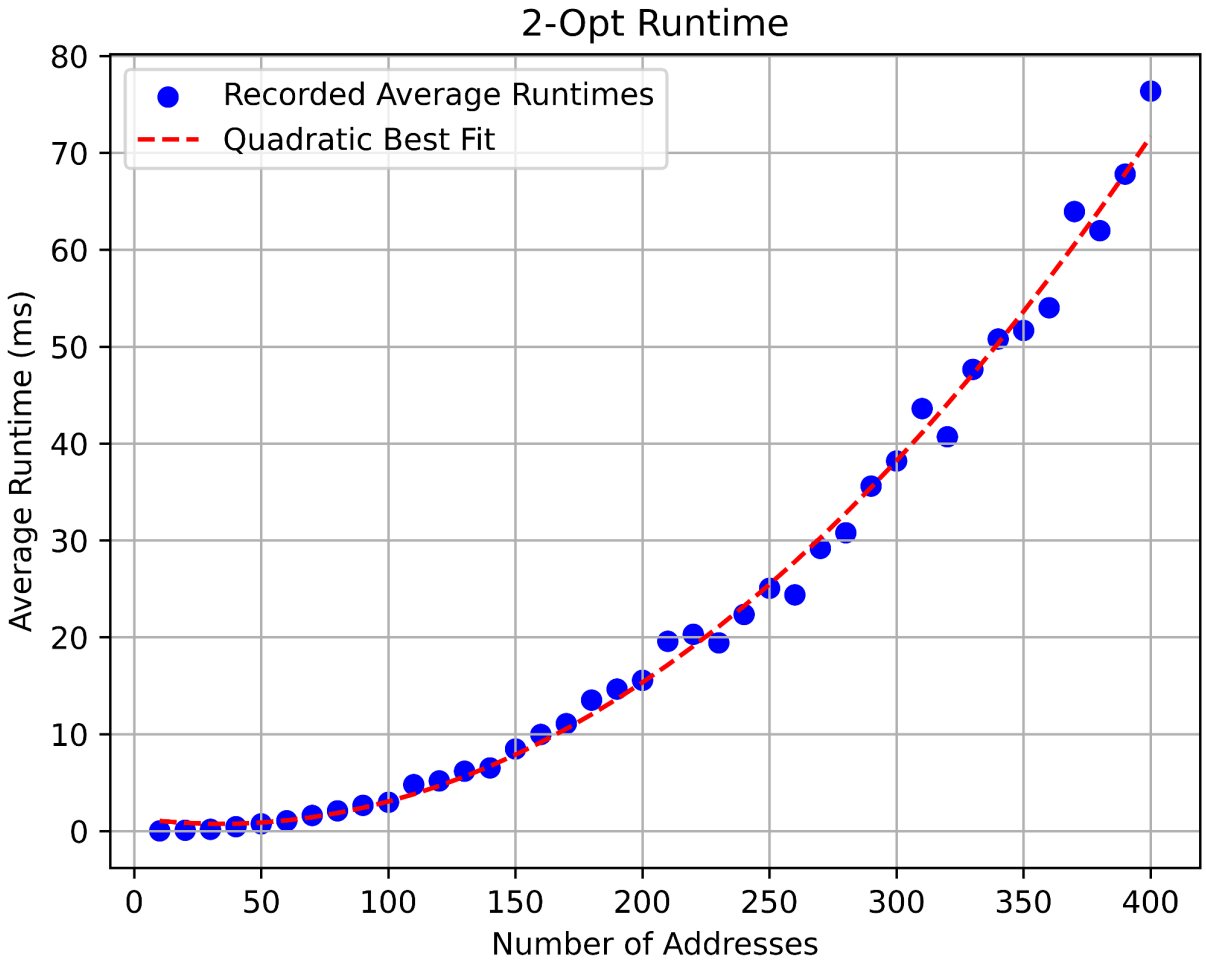

Figure 9: 2-Opt Runtimes

From Figure 9, it seems that the relation between the number of addresses and the average runtime is approximately quadratic. This supports our analysis in section 2.3.2 that the time complexity of the 2-Opt heuristic is $O(n^2)$.

Due to resource constraints, we only recorded the average runtimes of the Brute Force method for n ranging from 4 to 13.

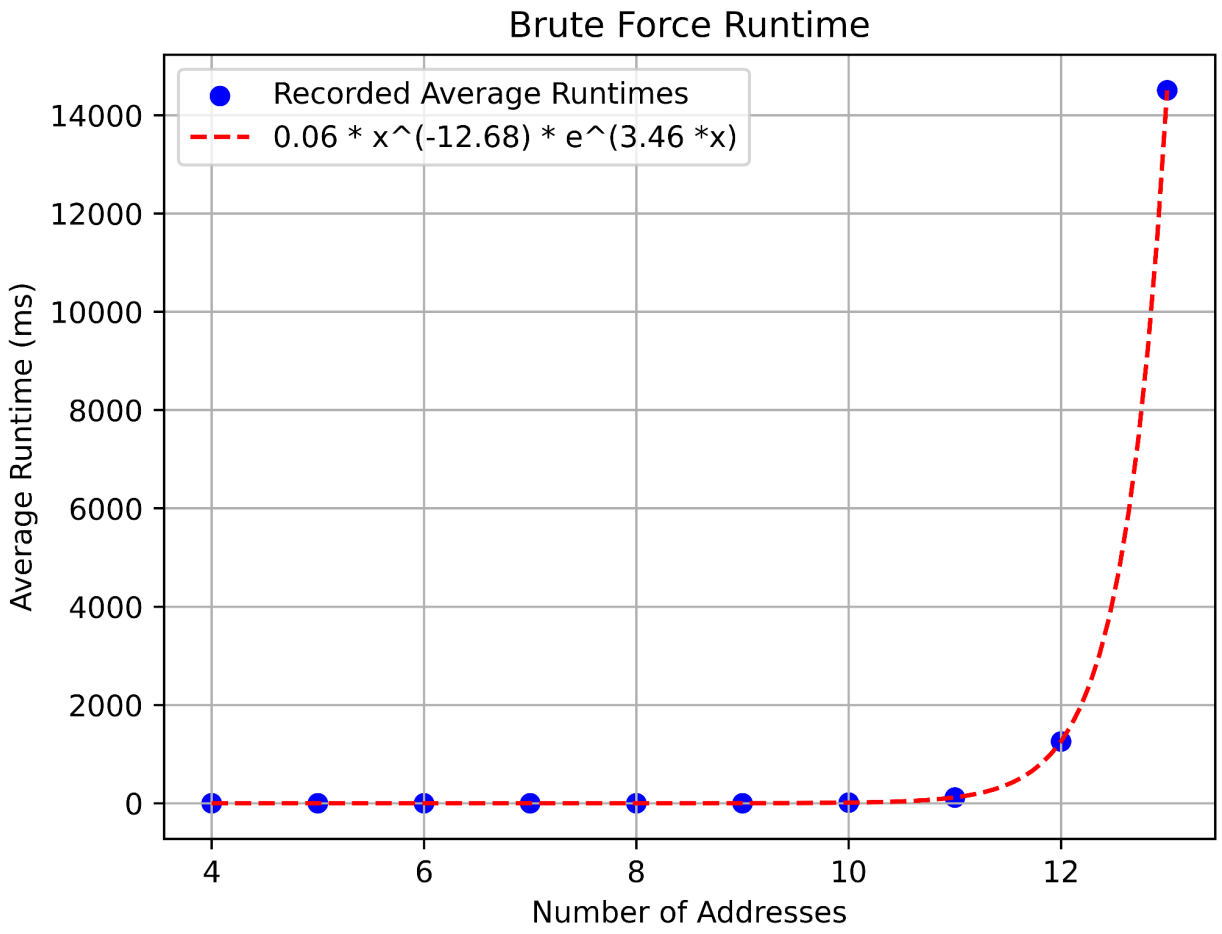

Figure 10: Brute Force Runtimes

Figure 10 shows that the runtime grows dramatically at $n = 11$. Using the data we were able to fit a function of the form $f(x) = ax^b e^c$. Earlier in section 2.2 we mentioned that the time complexity of Brute Force searching is factorial-like, so we'd expect that if we kept testing larger values of n , our data and the best fit line would diverge. However this does help visualize that the Brute Force method has non polynomial time complexity.

This experiment demonstrates the importance of applying heuristics like 2-Opt when it comes to scalability. For an application like Amazon deliveries, where a single warehouse might need to schedule deliveries to hundreds of nearby addresses, it isn't feasible to search all possible routes. Using the function we fitted to the Brute Force Runtime data, we calculated that searching all possible routes for just 100 addresses will take 4.8×10^{120} seconds on the same hardware. While not necessarily the best route, the 2-Opt heuristic can likely find a sufficiently short route in under 10 milliseconds.

3.4 Amazon Prime and Shipping Guarantees

Delivery services like Amazon offer premium services like Amazon Prime that guarantee shipping dates for certain products, e.g. next-day or two-day delivery. This means that these products must be shipped on a certain date and can't be exchanged with other routes that run on different days.

In this experiment we tested the impact Prime addresses had on the optimal route found by our multi-route 2-Opt heuristic. We initialized two routes with 50 addresses each. Then in each route, we randomly select a percentage of the addresses to be Prime Addresses which cannot be switched between routes. Next we recorded the total lengths after optimizing the route. We tested 11 trials, ranging from 0% to 100% Prime Addresses incrementing by 10%. In each trial the initial routes are kept identical. Furthermore, addresses selected to be Prime Addresses in one trial are guaranteed to be Prime Addresses in the next.

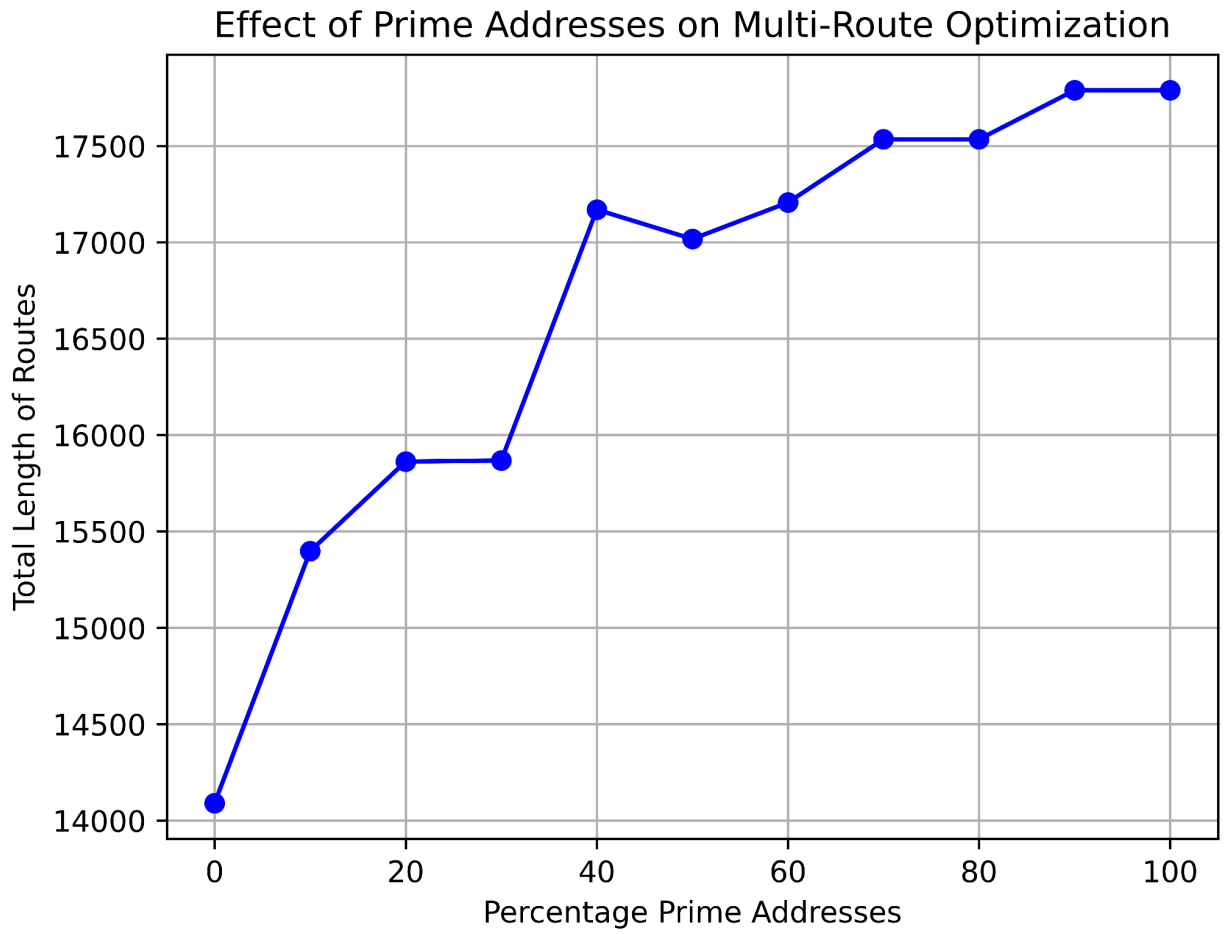

Figure 11: Percentage of Prime Addresses vs Optimized Route Length

Generally, as the number of Prime Addresses increases, the optimal routes returned by our heuristic tend to get longer. Intuitively, when 100% of the addresses are Prime Addresses, none of the addresses can be swapped between routes. In this case, our multi-route heuristic is no better than just running the 2-Opt heuristic on each route individually. However, as the number of Prime addresses decreases, it becomes more likely that a swap between routes that decreases the length of both routes is possible. This can be visualized in Figure 12. When there are no Prime Addresses, the heuristic finds routes that generally avoid crossing over each other frequently. However, as more Prime Addresses are added, the routes cross over each other more often.

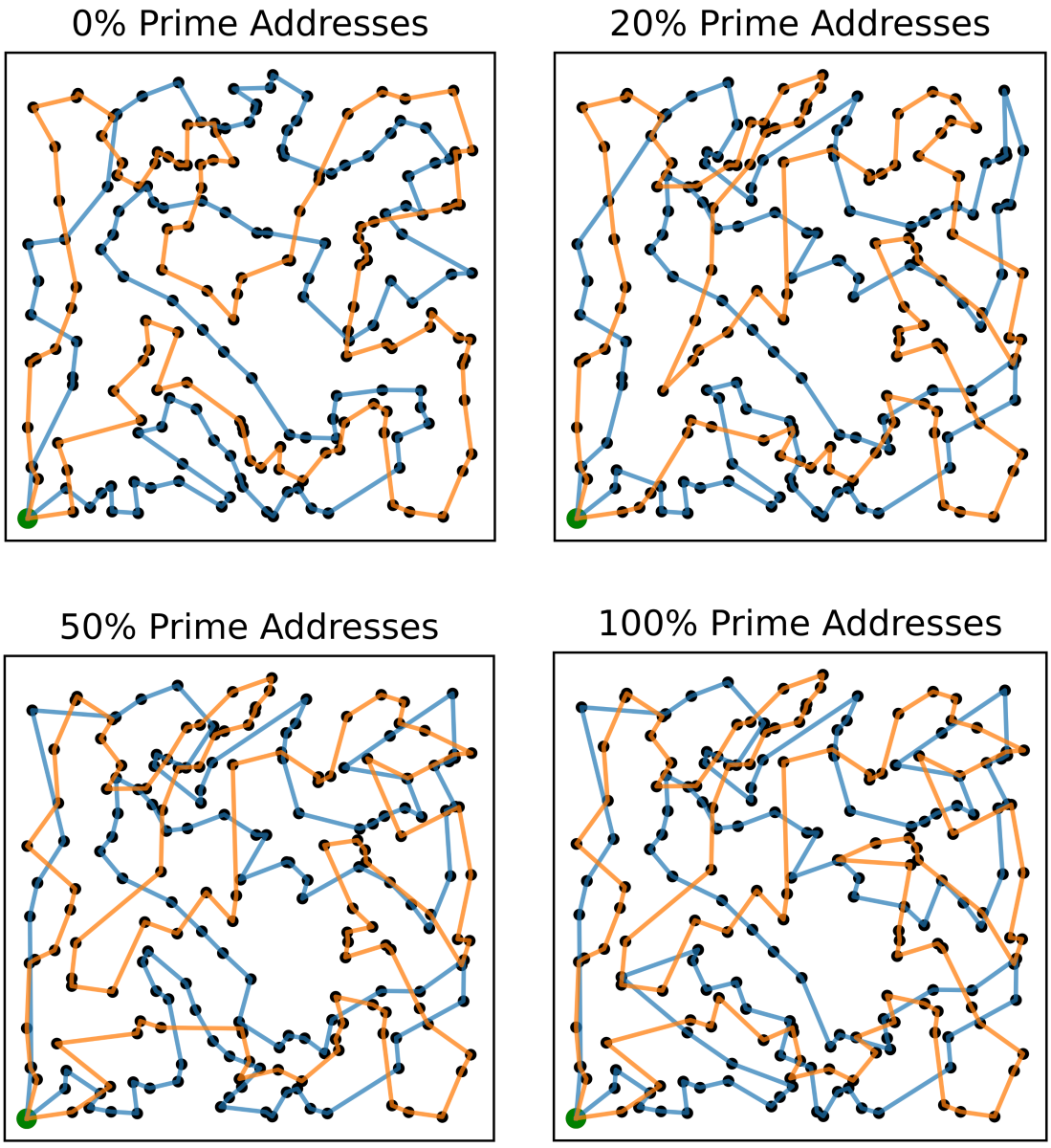

Figure 12: Samples of Multiple Routes

Interestingly, the relationship between the percentage of Prime Addresses and the total length of both routes is not necessarily monotonically increasing. Specifically in Figure 11, from 40% to 50% Prime Addresses the lengths actually decreased. A possible explanation for this is that preventing the heuristic from making a potential swap early on might preserve part of the path that allows for a future swap that is more beneficial. This highlights one of the weaknesses of our heuristic. The 2-Opt heuristic and the multi-route heuristic we implemented only consider two edges of the route at a time. They do not consider the route as a whole and how a swap might affect the optimization in the long run.

3.5 Ethics

The criticism towards Amazon's labor policies can be observed through randomly generated routes.The greedy route is a driver friendly route that sees delivery drivers have a more narrow pathway that ends with the final drop off point and the depot being the furthest distance apart. This allows the driver to destress before they arrive back at the depot. The multiple trucks route shows an inverse relationship with individual route lengths decreasing when there are added drivers, however, Amazon tries to optimize labor cost by giving drivers a challenging amount of addresses to fulfill. Overall, due to the guarantee of next day delivery with Prime membership, drivers are expected to make hard time committed deliveries when traveling great lengths.

Also, an important aspect to highlight regarding the optimization process in Figure 8 is that although the combined length of Route 2 and 3 decreased, Route 3's length increased and the difference between the two routes was greater than the initial routes shown in Figure 6. This disparity in the route lengths between the two days may not be fair for Amazon workers if we have three different drivers for each day. Amazon would need to either figure out a method to distribute the routes more evenly for the non-Prime days or provide additional compensation to the driver navigating through Route 3 accordingly.

4 Conclusion

In this study we have explored the traveling salesman problem and implemented optimization strategies such as the Greedy and 2-Opt heuristics. Additionally, we extended these strategies to handle their application in simulating Amazon delivery schedules. Specifically we considered performance, scalability, multi-route optimization, and delivery guarantees.

TSP is a difficult problem to solve computationally. Currently, finding the shortest route a delivery truck should take requires checking every single possible route, known as the Brute-Force method. However, instead of finding the shortest route, we can get by with finding relatively short routes. We have tested two heuristics that can efficiently approximate solutions to TSP. The first heuristic is called a 'Greedy heuristic' as it constructs a route based on always traveling to the nearest address. Additionally, we tested the 2-Opt heuristic, which tries to prevent a route from crossing over itself. We found that the 2-Opt heuristic consistently outperformed the Greedy heuristic, mainly because the Greedy heuristic makes decisions early on that lead to undesirable situations later on.

Furthermore we demonstrated that the 2-Opt heuristic was computationally far more efficient than a Brute Force search. Based on our implementation of both methods, we found that the 2-Opt heuristic is quadratic in time complexity, while a Brute Force search is factorial in time complexity. We experimentally verified this and showed that the runtime of the 2-Opt heuristic was roughly quadratic while the Brute Force search has non-polynomial time complexity. For Amazon deliveries where individual delivery trucks might visit hundreds of addresses, it simply isn't feasible to compute the best route. It is more practical to use heuristics to find good routes, even if they are not guaranteed to be the best.

Often for delivery services it is important to consider optimizing multiple routes simultaneously, as those routes can represent delivery schedules for different days or multiple delivery trucks running on the same day. We extended the 2-Opt heuristic to check and swap segments between two routes and considered the case of two routes running in opposite directions. We further generalized our 2-Opt heuristic to n routes by applying the 2-Opt heuristic for two routes on all pairs of routes.

Delivery services like Amazon also provide delivery guarantees like same-day or two-day delivery. In the context of our problem that means that certain addresses cannot be swapped between routes (days). Experimentally, we found that increasing the number of addresses that have a delivery guarantee generally, although not necessarily, results in less optimal routes since the heuristic is prevented from making beneficial changes to the route.

There are many areas of future work that we did not cover. In the real world, the optimal route is not necessarily the shortest route. There are additional metrics that influence the perceived cost of a route. For example, large companies like Amazon have to consider their environmental impact. In this case an optimal route might be one that is the most fuel efficient,

so we also have to consider not only the distance, but also the amount of fuel needed to get from one address to another.

Similarly, our work does not put any constraints on certain routes. For safety or policy reasons delivery drivers may not be allowed to drive a certain amount of miles or for a certain amount of time without ending their route. Additionally, our multi-route optimization algorithm might find in certain cases that making one route much longer than the other is most optimal. In reality, it might be better to find a way to balance routes amongst drivers to fairly split the workload. All of these additional constraints pose additional considerations for any optimization algorithm for the traveling salesman problem.

Our 2-Opt heuristic extended to multiple trucks has its limitations as well. Like the greedy heuristic, it's not able to consider how certain changes locally impact the route globally. Furthermore it only considers a single edge from each route and does not consider longer segments. There are plenty of other algorithms devised to tackle the multiple Traveling Salesman problem, such as genetic and clustering algorithms, and future work should be done to test their efficacy when applied to Amazon delivery scheduling.

5 References

[1] Eijkhout, Victor. *Introduction to Scientific Programming in C++17/Fortran2008: The Art of HPC, volume 3*. 2017–2022. Formatted August 18, 2023.

[2] *Traveling salesman problem*. Online Tutorials, Courses, and eBooks Library. (n.d.). https://www.tutorialspoint.com/design_and_analysis_of_algorithms/design_and_analysis_of_algorithms_travelling_salesman_problem.htm

6 Appendix

```
class Address {
private:
    double i, j;
    std::string Last_possible_delivery_date;
    bool prime_member;
public:
    // Constructor
    Address(double i, double j, std::string Last_possible_delivery_date = "No_Date"
, bool prime_member = false) : i(i), j(j),
Last_possible_delivery_date(Last_possible_delivery_date) ,
prime_member(prime_member) {}

    Address(){};

    double get_i() const {}

    double get_j() const {}

    // manhattan_distance and straight line method
    double distance(const Address& additional_addresses, bool manhattan_distance =
false) const {}


    // check if addresses are equal
    bool operator==(const Address& other) const {}

    bool operator<(const Address& other) const {}

    string as_string() const {}

    bool is_prime_member() const {}
};

class AddressList{
    protected:
        std::vector<Address> address_list;
    public:

        virtual void add_address(Address address){}

        // Calculate the total distance to visit all addresses in order
        virtual double length(){}

        Address index_closest_to(Address address) {}
```

```
        string as_string() const {}

class Route: public AddressList{
    public:
        Route(){

        void add_address(Address address) override {}

        Route greedy_route() {}

        void apply_2_opt() {}

        double swap_improvement(Route &rt2, int i_1, int i_2, int j_1, int j_2){}
        bool do_best_swap(Route &rt2, int i_1, int i_2, int j_1, int j_2){}

       // search through all possible routes
        void apply_total_search(){}

        void save_routes(string file_name, Route &rt2) {}
        // Check if any address is amazon prime
        bool has_prime_member(Address &rt1) const {}

std::vector<Route> multi_path_apply_2_opt(Route &rt1, Route &rt2) {}
}
```

```
class MultiRoute{
    private:
        int num_routes;
        vector<Route> all_routes;

    public:
        // constructor: if vector of routes is known
        MultiRoute(vector<Route> routes):
            num_routes(routes.size()), all_routes(routes) {}

        // constructor if only addresses are known
        MultiRoute(int n, vector<Address> addresses){}
```

```
void optimize_routes(){

    // if case route = 1 handle separately

    // keep optimizing until no more improvement
    double initial_length;
    bool improvement = true;
    int n = 0;
    while (improvement == true){
        n++;
        improvement = false;
        initial_length = total_length();
        // apply 2 opt individually
        for (int i = 0; i < num_routes ; i++){
            all_routes[i].apply_2_opt();
        }

        // apply 2 opt between all routes
        for (int i = 0; i < num_routes - 1; i++){
            for (int j = i+1 ; j < num_routes; j++){
                Route rt1 = all_routes[i];
                Route rt2 = all_routes[j];
                vector<Route> new_routes =
                multi_path_apply_2_opt(rt1,rt2);
                all_routes[i] = new_routes[0];
                all_routes[j] = new_routes[1];
            }
        }
```