Final Project: Amazon Delivery Truck Scheduling

Braden Pecora *(BSP825)*, Harrison Jin *(HQJ59)*
COE322

December 7th, 2020

Abstract

This paper explores some basic approaches to the classic "Traveling Salesman Problem" (TSP) with some additional constraints in the context of Amazon delivery trucks. Using a model based around a two-dimensional Cartesian coordinate system, simulations are run of a variety of different algorithms. Starting with the original TSP, the first approach is the "greedy route," which iteratively searches for the nearest location without accounting for the route as a whole. To further optimize the route, the researchers employed the 'opt2' heuristic to prevent the truck from doubling back to previously travelled areas. The paper then discusses the effect of certain variables on the computational complexity of this algorithm, and compares the improvements made over the "greedy route" approach. The next condition introduced to the model is a second route. With a second route, additional optimization options become available, built around the same ideas as the 'opt2' route. Optimizations that involve swapping route segments between the two paths are explored, and the resulting increase in runtime and decrease in length is discussed. In general, the paper finds there is often a point at which increasing the complexity has minimal payoffs for these methods. Next, the model considers Amazon Prime customers, who are guaranteed their delivery on a certain day. Representing a separate route as a separate day, this behavior can be simulated by preventing certain deliveries from being swapped to a different route. The implications of this restriction are explored for various proportions of Prime customers. It is found that increasing this proportion increases the length of the route, but only to a limit determined by the optimizations that can be made on a single path. Finally, same-day delivery orders are considered. This condition introduces the additional challenge of adding delivery locations to routes that have already been planned. The paper explores two possible ways of tackling this challenge. The first method involves locating a section of the existing route that is closest to the new address and inserting the address there. The second method is to completely reconstruct the route with the new addresses and the original addresses. The paper finds that in general, both methods yield comparable results, although method 1 is has lower computational cost. The paper concludes by exploring the limitations of this model, and how the model could be improved upon by considering more real-world factors such as gas station locations, expected traffic, and truck capacity.

1 Introduction

This paper explores the advantages and limitations of some basic route planning methods by simulating Amazon delivery trucks. Like the well-known "Traveling Salesman Problem" (TSP), the primary goal of this model is to optimize the route through a given list of addresses such that the total distance, and thus time and cost, are minimized. However, this particular model introduces some additional constraints and parameters to the construction of the route. These conditions are as follows:

- Amazon, a trillion-dollar company, likely has the ability to afford more than one truck to fulfill its deliveries. In this simulation, we assume two trucks could be used for a given list of addresses.
- Amazon Prime customers are guaranteed delivery on a certain day, whereas regular Amazon customers are only given an estimated delivery day.
- Some addresses might need to be added to a truck's route after the route has already been planned (same-day delivery).

This paper begins with the original TSP, then adds one or more of these conditions at a time. The simulation explores various strategies for dealing with each of these conditions.

1.1 Model Construction and Assumptions

To construct a model for route planning, we must first establish a method to describe locations. Our model uses a two-dimensional Cartesian coordinate system, where all locations are described by a coordinate (x, y) such that x and y are integers. Furthermore, each location is designated as either a Prime customer or not a Prime customer. Each location will be an instance of the `Address` class, which stores the properties described above.

```
class Address{
    private:
        int x, y;
        bool prime;
}
```

Listing 1: Address class

Next, we need a way to calculate the distance between two locations. For our model, we provide two methods to calculate distance:

1. The Pythagorean distance formula: $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$
2. The Manhattan distance formula: $|x_2 - x_1| + |y_2 - y_1|$

While calculating distance is a necessity to determine the optimal route, the route-planning algorithms described in this paper still work properly regardless of the actual implementation method. The simulations in this paper use Method 1.

The next step is to store a list of delivery locations. For this simulation, the `AddressList` class was used. `AddressList` contains some helper methods, as well as an attribute `addresses` that stores `Address` instances in a `vector`. This model assumes that no `Address` will need to be visited more than once. That is, each instance of `AddressList` will not contain any duplicate locations. Finally, we created a class `Route`, derived from the `AddressList` class. A `Route` instance is the actual path that would be followed by a delivery truck, stored in order in the `addresses` attribute. `Route` defines the location of the truck depot, which will be used as the starting point for each delivery truck. For this paper's simulations, the depot is arbitrarily placed at the point (0, 0). `Route` also contains the optimization algorithms described throughout the remainder of this paper.

```
class Route : public AddressList{
    private:
        Address depot;
    public:
        Route(): depot(0,0) {
            addresses.push_back(depot);
        }
}
```

Listing 2: Route class

2 Greedy Route Scheduling

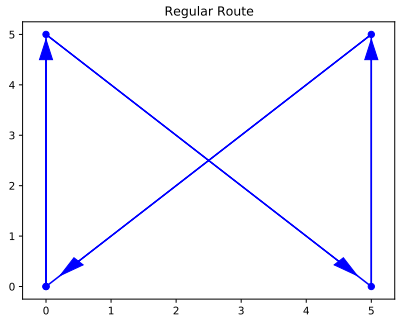
To begin to solve the TSP with a given list of addresses, it is preferable to make a quick approximation of an optimal tour before employing further optimizations with higher computational intensities. The strategy we used to create this approximation is called the `greedyRoute()` method. Given a list of addresses and a starting point, the `greedyRoute()` method aims to find the address closest to the starting point and establish it as the first address in the route. This address is then removed from the list. From there, the method finds the address closest to the first address out of the remaining addresses in the list. This address is added to the route and removed from the list. This process is repeated until there are no remaining addresses in the address list.

```
Route greedyRoute(bool manhattan=true){
    Route copy = *this;
    Route greedy;
    Address currentLoc = depot;
    int i;
    while(copy.addressesSize() > 0){
        i = copy.indexClosestTo(currentLoc, manhattan);
        currentLoc = copy.at(i);
        greedy.addAddress(currentLoc);
        copy.removeAddress(i);
    }
    return greedy;
}
```

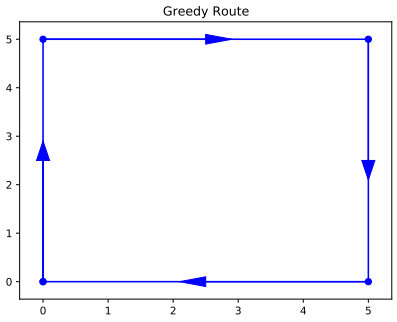
Listing 3: Greedy Route method

A simple depiction of this can be seen in Figure 1. Originally, the addresses were inputted in the order (0,5) (5,0) (5,5) with a starting point at the depot, located at (0,0). It is obvious that this is not the most optimal path to travel. For instance, the `greedyRoute()` method recognizes that (5,5) is closer to (0,5) than (5,0) is, and returns a route that accounts for this. As a result, the total distance traveled along the route decreases.

This is a fairly trivial example that ends up returning the most optimal route. However, it does not take much experimentation to see that this method will rarely return the most optimal route as the size of the address list increases. The `greedyRoute()` method does not look more than one address ahead and does not look behind in its attempt to optimize the route. Consequently, further optimizations will need to be made to create shorter routes.



(a) A route constructed from addresses as they were inputted. Length = 24.14



(b) An optimized version of the route using the Greedy Route method. Length = 20.

Figure 1: Regular Route vs. Greedy Route [1]

3 'Opt2' Optimization of the Route

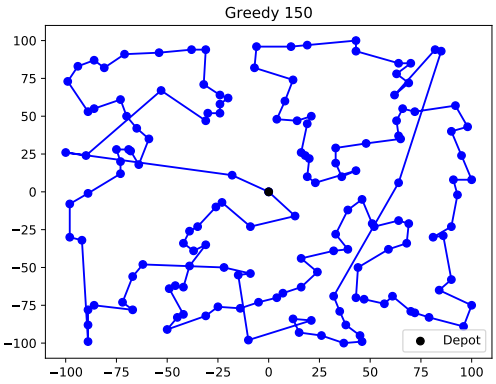
Properly solving the TSP is a difficult and time consuming task for both the coder and the computer. Consequently, we aimed find the solution heuristically. The method we employed is based on the *opt2* idea, which states that a path that crosses or intersects with itself can be made shorter by reversing a segment of the route.

In order to execute this idea, the first run the `greedyRoute()` method within our `opt2Route()` method to generate a starting route with the inputted addresses. The `opt2Route()` method then determines the maximum number of addresses to be reversed based on an inputted percent of the list (the default is 15%). Then, for each integer number of addresses to reverse (from the max number to one), the method creates a new route where that number of addresses forward from a given starting point in the list are reversed. It does this for every possible starting point (every address in the list besides those that do not have enough subsequent addresses to swap). If a newly generated route shortens the tour, it is kept. The `opt2Route()` method can be seen below in Listing 4.

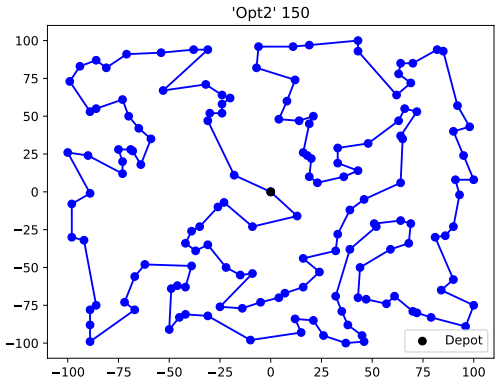
```
Route opt2Route(bool manhattan=true, float percentToReverse=0.15){
    Route opt2 = greedyRoute(manhattan);
    int maxToReverse = max((int)floor(percentToReverse*opt2.addressesSize()), 1);
    for(int numToReverse = maxToReverse; numToReverse > 0; numToReverse--){
        for(int i = 1; i < addresses.size()-numToReverse; i++){
            Route test = opt2;
            test.reverseAddresses(i, numToReverse);
            if(test.length(manhattan) < opt2.length(manhattan)){
                opt2 = test;
            }
        }
    }
    return opt2;
}
```

Listing 4: 'Opt2' heuristic method

[1] Graphics created using Python library `matplotlib`.



(a) 150 random addresses optimized with only the "greedy route" method.
Length = 2163.97



(b) The same 150 addresses further optimized with the "Opt2" method (percentToReverse = 0.75).
Length = 1967.02

Figure 2: Greedy Route vs. 'Opt2' Route

3.1 Runtime

Our current `opt2Route()` method has a runtime complexity of $\mathcal{O}(n^2)$ in its current state. However, compromises can be made to lower the runtime. For instance, an easy way to lower the runtime would be to decrease the `percentToReverse` parameter. For small and medium sized routes, the `percentToReverse` parameter can be close to one and the runtime will still be very short. However, if this method is run with a very large route, it would not make sense nor would it be practical to have a `percentToReverse` close to 1. The runtime would be way too long. Furthermore, in a practical situation with a long address list, it would be likely that addresses are clumped into groups such as neighborhoods, cities, or states (depending on the scale). It would not make sense to test if reversing the route for each starting point and each length to reverse across several groups would decrease the length of the route.

Our current method treats every possible address as a starting point with which to reverse from. Depending on the scale, it might not be worthwhile to have this method check every single starting point (especially for trials where the `numToReverse` is high). Instead, it could be optimal to check high `numToReverse` for only a few addresses in the list.

It is worth noting that, for the same address list, the runtime of the `opt2Route()` method will always be much longer than the `greedyRoute()` method. Unless the address list has very few addresses and `percentToReverse` is very low, the the runtime of `opt2Route()` will be at least 1000% longer than `greedyRoute()` (the fact that `opt2Route()` runs `greedyRoute()` is accounted for).

3.2 Improvement over the Greedy Route method

Both the computational runtime and length decrease of the 'opt2' method are dependent on `percentToReverse` and the number of addresses in the list. Below are tables that compare the effects of these values. The range of x and y values will be from the negative of the list size to the positive of the list size. For instance, if the list contains 150 addresses, $x, y \in [-150, 150]$. The addresses are to be randomly generated.

The data in Tables 1 through 3 provide interesting, but not unexpected, results. Tables 1 and 2 indicate that `percentToReverse` has an approximately linear relation with runtime. Table 3 suggests

Percent of the list reversed:	75%	50%	40%	20%	10%	5%	1%
Percent increase in runtime:	36,736%	29,932%	24,804%	13,360%	7,132%	3,688%	688%
Percent decrease in length:	11.29%	11.15%	7.90%	7.90%	7.42%	7.36%	0.45%

Table 1: The impact of the `percentToReverse` parameter of the `opt2Route()` method on runtime and decrease in length when compared to the `greedyRoute()` method. The address list contains 200 randomly generated addresses. The 'percent increase in runtime' row is the average percent increase in runtime over twenty-five trials with the same data on the same machine.

Percent of the list reversed:	50%	25%	10%	5%	1%
Percent increase in runtime:	75,580%	43,890%	18,980%	9,770%	1,920%
Percent decrease in length:	9.73%	7.57%	6.87%	5.24%	2.27%

Table 2: Various `percentToReverse` values and the resulting increase in runtime/decrease in length when comparing the results of the `opt2Route()` to the `greedyRoute()`. The address list contains 500 randomly generated addresses. Runtime is averaged over 10 trials.

Length of list:	500	250	100	50	10
Percent increase in runtime:	28,096%	13,208%	4,836%	2,106%	96%

Table 3: The effect of list length on runtime. For each list length, five random lists of addresses were generated. Both `opt2Route()` and `greedyRoute()` were performed on each list. The average percent increase in runtime was calculated over ten trials for each list. `percentToReverse` = 0.15

that the length of the list also has an approximately linear impact on runtime as well (for reasonably long lists). Tables 1 and 2 also seem to suggest that a `percentToReverse` value around $15 \pm 5\%$ returns a reasonably shorter list without expending too much runtime. Higher `percentToReverse` values seem to not yield too much shorter of a result.

The runtime of the `greedyRoute()` method is on the order of milliseconds. The percent increase in runtime for `opt2Route` seems huge, but the actual additional runtime is not really that long. For a list that contains a couple hundred addresses, the user can expect an additional runtime of around 10 seconds (if the `percentToReverse` value is reasonable). For shorter lists, the additional runtime can be less than a second.

If the list is not extremely short, the user can reasonably expect that `opt2Route()` will decrease the length of the route by 5 to 10% in comparison to the route provided by `greedyRoute`. This is a pretty decent improvement for the reasonably short runtime. Figure 2 provides a visualization of these improvements. It can be see that portions of the route that seem to 'cross' in Fig. 2a are 'untangled' in 2b. Noticeable improvements are made with this method.

4 Multiple Trucks

The next condition we introduce into the model is that of multiple trucks. To simulate this, we split a single `AddressList` into multiple `Route` instances. For the purposes of this model, we use only two instances of `Route`. This could be used to simulate either two trucks simultaneously fulfilling deliveries or as a single truck fulfilling deliveries over two days. The initial `AddressList` is therefore split randomly instead of by geometric proximity, as this simulates the addition of new delivery locations during day one for a single truck. It is also assumed that each instance of `Route` will have approximately the same number of delivery locations.

At this point, we have two separate `Route` instances, and the goal is to minimize the sum of their

lengths. It makes sense to again apply the *opt2* heuristic. We can start by simply running `opt2Route()` on each `route` individually. However, rather than simply reversing a number of addresses within a route, we now have the additional option of swapping addresses between the two routes. In order to keep each `Route` around the same length and reduce computational intensity, we only swap sections of equal length between the two routes. Thus, we have four possible variations to consider given sections of equal length in each `Route`:

1. Directly swapping the sections
2. Reversing the section in the first `Route`, then swapping it to the second `Route`
3. Reversing the section in the second `Route`, then swapping it to the first `Route`
4. Reversing both sections, then swapping them

Of course, the variation with the shortest total length is not definitively the most optimal, as there could be a swap between two different sections that result in a shorter total routes. Therefore, we store the shortest set of routes for a given combination of sections, then compare it to the shortest set from the next combination. This algorithm is shown below in Listing 5.

```
for(int lengthToSwap = maxToSwap; lengthToSwap > 0; lengthToSwap--){
  vector<Route> minPath = truckPaths; // truckPaths starts as the original Routes
  for(int path1 = 1;path1 <= truckPaths.at(0).addressesSize()-lengthToSwap; path1++){
    vector<Route> minTestPath = minPath;
    for(int path2 = 1;path2<=truckPaths.at(1).addressesSize()-lengthToSwap; path2++){
        // Find minTestPath with shortest length
        // Omitted for brevity
        ...
    }
    if(twoTruckLength(minPath, manhattan) > twoTruckLength(minTestPath, manhattan)){
      minPath = minTestPath;
    }
  }
  if(twoTruckLength(truckPaths, manhattan) > twoTruckLength(minPath, manhattan)){
    truckPaths = minPath;
  }
}
```

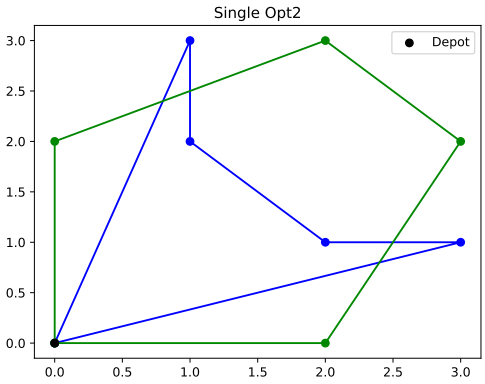
Listing 5: 'Opt2' heuristic for multiple trucks

This algorithm assumes that starting by swapping the longest sections will yield the largest gains in optimization. The algorithm then continues to optimize smaller and smaller sections of the routes. An example of the optimizations made by this algorithm are demonstrated in Figure 3. Notice how the route segments [(1,2),(2,1)] and [(2,3),(3,2)] have been swapped between the different paths.

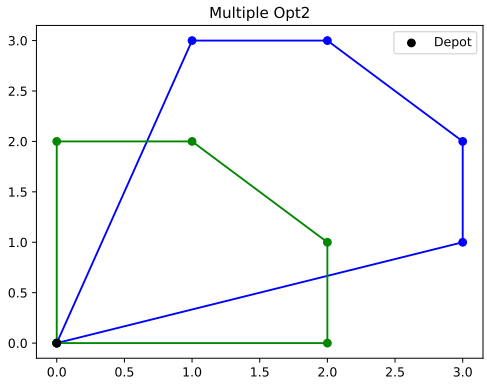
The assumption that the separate routes will have the same number of delivery locations does introduce some edge cases that are actually longer than necessary (see (-7, 8) in Figure 4a). However, this is a reasonable assumption to make given real-world factors such as fixed truck capacity or newly-placed orders. In any case, these specific instances could be accounted for in more comprehensive route-planning algorithms.

4.1 Flexibility versus runtime

The algorithm described above has a runtime complexity of $\mathcal{O}(n^3)$, so it is important to balance flexibility of the route with the runtime of the program. Like the 'opt2' method for a single route, both the computational runtime as well as the decrease in total length is dependent on the value of



(a) Two routes separately optimized with the single truck 'opt2'. Combined length of routes: 19.63



(b) Two routes optimized with the multiple truck 'opt2'. Combined length of routes: 17.15

Figure 3: An example of the optimizations made with the 'opt2' heuristic for multiple trucks.

`percentToSwap`, which serves an analogous purpose to `percentToReverse` in `opt2Route()`. Another important factor to consider is the number of delivery locations. The tables below show the effect of these variables on the runtime and decrease in total length compared to simply running `opt2Route()` individually on each route.

Percent of list swapped	1 %	5%	10%	15%	20%	50 %
Average runtime increase	2,560%	5,472%	12,960%	18,776 %	25,211%	40,776%
Average length decrease	4.76%	9.36%	15.89%	20.63%	22.32%	24.66%

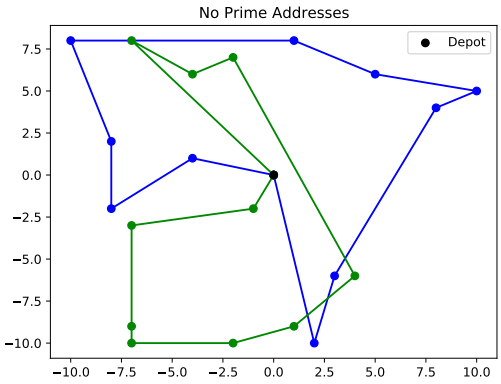
Table 4: The impact of the percent of the lists swapped between two routes on runtime and length for the two truck 'opt2' relative to running 'opt2' on the routes separately. Five randomly generated address lists containing 100 addresses ($x, y \in [-100, 100]$) were generated for this simulation. The runtime increase and length decrease were averaged over ten trials for each of the five different address lists.

Number of addresses	250	100	50	25
Average runtime increase	38,400%	15,896%	8,160%	3,004%
Average length decrease	17.4%	15.5%	12.2%	8.29%

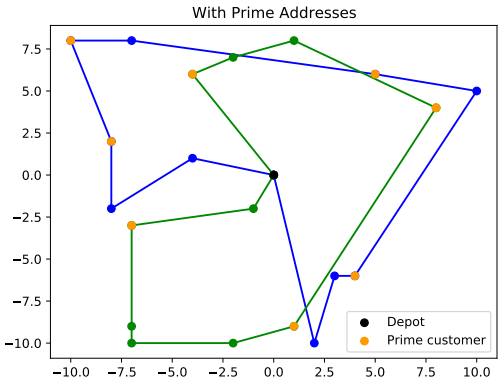
Table 5: The impact of number of addresses between two routes on runtime and length for the two truck 'opt2' relative to running 'opt2' on the routes separately. For each address list length, five trials across five randomly generated address lists were averaged to determine average runtime increase and length decrease. The percent of the list swapped was 15% for all lengths.

As expected, increasing the percent of list swapped or the length of the list increases the runtime. Table 4 indicates that, for a list of 100 addresses, swapping a maximum of 15 percent of the list yields a close-to-optimal result. Increasing the percent to swap further only marginally decreases the route lengths and costs a lot of additional runtime.

For creating Table 5, individual trials with on a list size of 250 addresses took upwards of 30 seconds on a somewhat decent computer. This is a fairly reasonable runtime for a program like this; however, the runtime could be very large if the list size increases. Furthermore, a more complicated method



(a) Without accounting for 'prime' status.



(b) 'Prime' status accounted for.

Figure 4: Accounting for the 'prime' status of an address changes the route generated with the multiple route 'opt2' method.

of determining the distance between two addresses would be used in a real-life scenario. Thus, the multiple truck 'opt2' method could be very costly in practical execution.

It is also worth noting that while the principles behind this algorithm could be applied to more than two separate routes, the computational complexity would increase exponentially for each additional route. Therefore, a different algorithm may need to be used for a scenario that requires a large number of trucks or separate routes.

5 Amazon Prime

In Section 4, we made the assumption that all addresses can be exchanged between routes. Practically, we can not always hold this assumption to be true. For instance, if the different routes represent different delivery dates, there may be a scenario in which certain packages must be delivered to their respective addresses on a specific date. Such is the case for Amazon: members of Amazon Prime are guaranteed a certain delivery date while "non-Prime" members only get an approximation.

As seen in Listing 1, our `Address` class contains the boolean instance variable `prime`. If `prime` is true for an instance of `Address`, this indicates that this specific address has 'Prime' status. Thus, this address can not be swapped between routes. There is also a method within the address class called `isPrime()` that returns `true`. (The `Address` constructor defaults `prime` to false if not specified, so all previous sections of code have ignored this parameter.)

The 'prime' status of an address is fairly simple to implement into the *'opt2'* heuristic method for two trucks. Within the portion of the method that swaps addresses between routes, we check to see if either address is 'prime'. If the address is 'prime', we do not swap the addresses:

```
1 if(!address1.isPrime() && !address2.isPrime()){
2     // Swap address1 and address2
3 }
```

Listing 6: A simple way to check for 'prime' status before swapping addresses between routes.

The impact of this can be seen in Figure 4. Some 'prime' addresses can have no impact on the route while other 'prime' addresses can cause the shape of portions of the route to change entirely.

Percent of addresses that are 'prime':	1 %	5%	10%	25%	50%	75%
Percent increase in length of both routes:	0%	5.0%	13.1%	21.2%	26.8%	32.8%

Table 6: Average increase in the combined length of the routes when the 'prime' status of an address is considered.

5.1 Proportion of Prime customers

To see the impact the `prime` parameter has on the combined length of the routes, we created five randomly generated address lists containing 100 addresses with $x, y \in [-100, 100]$. For each list, we ran the multiple truck 'opt2' heuristic method. Then, we randomly assigned a percentage of the address list a 'prime' status and ran the multiple truck 'opt2' method again. The total lengths of the routes created before and after assigning 'prime status' were compared. The percent increase in length was then averaged across the five address lists. These steps were repeated for several ratios of 'prime' to 'non-prime' addresses on the same lists. The results are displayed in Table 6. It is fairly apparent that increasing the amount of prime addresses increases the length of the routes. The percent increase in route length reaches a maximum as the percentage of 'prime' addresses nears 100%. This occurs because the multiple route 'opt2' method can no longer make many changes, and only the single route 'opt2' can create a majority of the optimizations.

6 Dynamicism

In many high-population areas, Amazon allows Prime customers to select same-day delivery. This means that some delivery trucks may need to add locations to their route after having already planned and optimized their routes. This section will explore two possible approaches to this scenario, with the assumption that there are two trucks fulfilling deliveries for the day.

6.1 Method 1: Adding new addresses to existing routes

The first approach to this scenario is to find the part of an existing route that is closest to the new address, then inserting the new address into the route at that index. We try this approach for both routes, and select the one in which the increase in route length is minimized. This process is repeated for each new address to be added. Assuming that the existing routes were well-optimized to begin with, this method yields a decently optimal route for relatively low computational cost. Finally, `opt2Route()` is run individually on each route to prevent any crossing paths. Our implementation of Method 1 is shown below in Listing 7.

```
for(int i = 0; i < newAddresses.addressesSize(); i++){
    Address house = newAddresses.at(i);
    Route route1 = paths.at(0);
    Route route2 = paths.at(1);
    double origDist1 = route1.length(manhattan);
    double origDist2 = route2.length(manhattan);

    int index1 = route1.indexClosestTo(house);
    int index2 = route2.indexClosestTo(house);
    route1.insertAddress(index1, house);
    route2.insertAddress(index2, house);

    route1 = route1.opt2Route(manhattan, percentToReverse);
    route2 = route2.opt2Route(manhattan, percentToReverse);
    double newDist1 = route1.length(manhattan);
    double newDist2 = route2.length(manhattan);
```

```
    if(newDist1 - origDist1 < newDist2 - origDist2){
        paths.at(0) = route1;
    }else{
        paths.at(1) = route2;
    }
}
```

Listing 7: C++ implementation of Method 1

6.2 Method 2: Adding new addresses and re-optimizing

The second way to approach this situation is to completely scrap the existing routes and plan new routes from scratch. We simply add the new addresses to the existing list of addresses, then construct the routes using the method described in Section 4. This method is more computationally expensive, especially given the fact that it also renders any prior computation meaningless. Our implementation of Method 2 is shown below in listing 8.

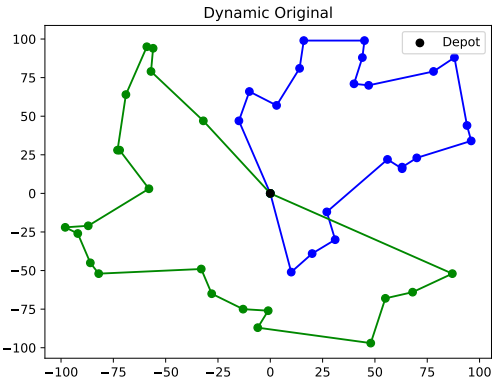
```
vector<Route> addBeforeSplittingRoutes(AddressList newAddresses, bool manhattan =
    true, double percentToSwap = 0.15){
    for(int i = 0; i < newAddresses.addressesSize(); i++){
        addresses.push_back(newAddresses.at(i));
    }
    return twoTruckOpt2(manhattan, percentToSwap);
}
```

Listing 8: C++ implementation of Method 2

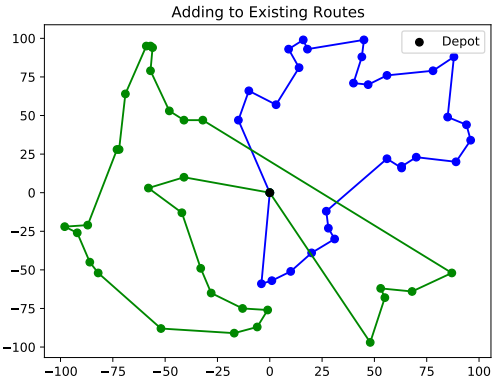
6.3 Comparison of Method 1 and Method 2

The best method to use in a given situation depends on a variety of factors, such as computational cost, number of new addresses, and other real-world factors. In general, Method 1 will produce satisfactory results if the number of new addresses is low relative to the number of existing addresses. For a high number of new addresses, it may be optimal to choose Method 2, as Method 2 is not influenced by any prior computation. Figure 5 shows an example of adding 16 new addresses to a list of 43 existing addresses using both Methods 1 and 2.

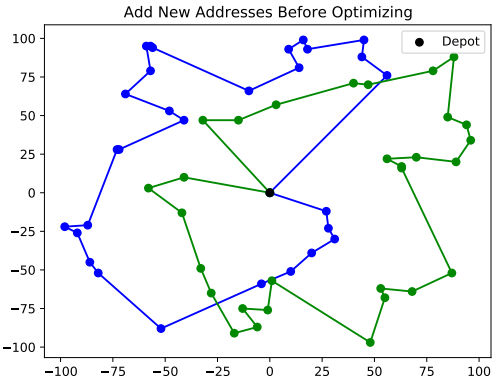
In this specific case, Method 1 yielded a slightly lower total length with routes similar in shape to the original routes. Method 2 yielded a comparable result for the total length, but the routes were completely different shapes and directions, which may be a factor when choosing between the two methods. It is impossible to name one method that will yield the shorter routes in all cases, so if computational cost is not a concern, it would be trivial to run both methods and choose the desired set of routes.



(a) Initial 43 addresses split between two trucks and optimized as described in Section 4, with `percentToSwap` = 0.5. Combined length of routes: 1115.01



(b) 16 new addresses added to the routes from Figure 5a using Method 1, with `percentToReverse` = 0.5. Combined length of routes: 1372.71



(c) 16 new addresses (the same ones as in Figure 5b) added to the routes from Figure 5a using Method 2, with `percentToSwap` = 0.5. Combined length of routes: 1398.72

Figure 5: Resultant routes when applying either Method 1 or Method 2 to the same test case.

7 Conclusion

The problem of planning a route with minimal distance, time, and cost depends on a variety of factors in the real world, yet it is an important problem with many applications. In this paper, we considered conditions specific to Amazon's case, including multiple delivery trucks, Amazon prime customers, and newly placed same-day delivery orders. Various solutions to these constraints were shown, yet these algorithms are still just a preliminary exploration into this problem. Even just within the context of Amazon, there are many more factors to consider, such as gas station locations, expected traffic, and truck capacity. It is obvious that other applications of the TSP would also have a variety of constraints and parameters specific to those situations. It would be an interesting challenge to model those conditions and fit the algorithms described in this paper to other applications. There are also many more solution strategies to the problem that can be explored, ranging from brute force to more mathematically rigorous approaches.

In addition, the simulations in this paper modeled a world where every location is directly connected to every other location. In reality, it would be important to consider the various paths between two locations and the effect of those paths on the distance and time spent. It was also assumed that distance correlates directly to time, but factors such as traffic and speed limits mean that this is not always true. For Amazon, time spent paying hourly workers could be just as valuable as distance traveled, so it is important to consider the actual time spent fulfilling deliveries as well as the distance traveled. Some companies chose to prioritize gas prices and truck longevity in many cases. Many transport companies aim to optimize their routes in manners that limit the amount of left turns and time spent on the highway.

There is a reason why the "Traveling Salesman Problem" lives in infamy among the programming community. Further optimizations can always be made, but always at the cost of processing power and the programmer's patience. As demonstrated in this paper's simulations, there is often a point at which spending more computations yields only marginal improvements. Ultimately, it is the user and their computational capabilities that decide how far they want to optimize.