Optimal Delivery Scheduling for Amazon Trucks

Lauren Rodriguez and Katelyn La Joie

November 2022

1 Author Contact Information

EIDs	Emails	TACC Username
lnr687	laurennrodriguez@utexas.edu	lnr687
kol242	katelynlajoie@utexas.edu	klajoie

2 Accessing the Project

The repository used for this project can be found at the following Github link: `https://github.com/lajoiekatelyn/coe322finalproject`

3 Introduction

Amazon ships approximately 1.6 million packages per day [1] and offers over 12 million different products [2]. To achieve such a high delivery rate, optimizing delivery routes is necessary to enable timely deliveries to consumers within healthy working hours for the drivers. From 2020 to 2021, the company saw a 6.4% decrease in its number of late packages [3], indicating a commitment to continuously improving in route optimization.

The Amazon optimization problem is functionally very similar to the Traveling Salesman Problem (TSP), which attempts to identify the shortest route for visiting every city on a list one time before returning to the origin. This project seeks to do the same, with some additional Amazon-specific perks: given a list of addresses, calculate the optimal path to stop by each one, ensuring that all Amazon prime customers get their delivery by the next day, but understanding that normal customers have a range of days where their product may arrive. Given a list of addresses for, say, a span of 3 days, the code should schedule an optimal route to reach all Prime deliveries on the first day and as many normal deliveries as ethically possible, and then optimize the remaining two deliveries on the last two days, across a given number of trucks.

In this project, a potential solution to the route optimization problem is explored through first a greedy construction of the route, then approximating

the optimal route using a heuristic method. The greedy solution is creating a route by finding the closest address to the current address, and making that the next stop. As shown in this project, while easy to calculate, it will not provide the shortest route, and never compare itself to the original route or other solutions to check for improvement. Meanwhile, a heuristic method uses more of a 'trial and error' approach to find a better solution.

4 Method

4.1 Generating the Address List

The original list of coordinates used to construct a route was created in MATLAB. The script takes user inputs for the number of addresses to generate coordinates for, and the route number. The script then uses the MATLAB function ***randi*** to generate the input amount of values for both x and y coordinates individually, across a given range. These points are then converted to type double.

```
%select points and convert to double
addressesx = randi([-10 10], 1, numAddresses);
addressesx = double(addressesx);

addressesy = randi([-10 10], 1, numAddresses);
addressesy = double(addressesy);
```

These points are written to a .txt file in the style used by the main functions of the optimizing program, and then to another .txt file as just coordinates, for easier reference.

```
%write points to text file
fileID = fopen('addressgenRoute2.txt','w');
fileID2 = fopen('addresscoordinatesRoute2.txt','w');

for i = 1:numAddresses
writetofile = fprintf(fileID, '\nroute%d.add_address(
Address(%.1f, %.1f) );', routenumber, addressesx(i),
addressesy(i));
end

for i = 1:numAddresses
writetofile = fprintf(fileID2, '(%.1f, %.1f)\n',
addressesx(i), addressesy(i));
end
```

This address list function also plots the original route as generated by plotting all coordinates as well as [0,0] as the start and end of the plot, to construct a full possible route.

```
plotx = [0, addressesx];
plotx = [plotx, 0];
ploty = [0, addressesy];
ploty = [ploty, 0];

figure
plot(plotx,ploty,"o-")
grid on
title('Original Route')
```

The produced lists can then be copy and pasted into main functions as the route(s) to be optimized. Files "addresscoordinatesRoute1.txt" and "addresscoordinatesRoute2.txt" have the list of coordinates used by the program to obtain the results discussed below.

4.2 Constructing the Route

To construct a route, the project utilizes 3 classes: the Address class, the Address List class, and the Route class.

4.2.1 Address Class

The Address class defines an address as an arbitrary x, y pair on a city grid. Each address also indicates whether or not it is a prime destination, which defaults to false unless the constructor indicates otherwise.

```
Address() {};
 Address( double i, double j )
   : x(i), y(j) {};
 // add constructor for an address that indicates whether prime.
 Address( double i, double j, bool isprime )
   :x(i), y(j), prime(isprime) {};
```

The Address Class also calculates the distance between two addresses according to the distance equation.

```
double distance( Address other ) {
   double dx = x - other.x;
   double dy = y - other.y;
   return sqrt( dx * dx + dy * dy );
 }
```

4.2.2 Address List Class

The Address List class utilizes the aforementioned Address class to create an arbitrary list of addresses, measure the total distance traveled for the points in

the list, and find the closest address to any given address. Addresses are added to an Address object using the following function:

```
void add_address(Address address) {
    list_of_addresses.push_back(address);
  }

```

The total distance in a list of addresses is measured by using the distance function created in the Address class for every element in the list, and is written as follows:

```
double length() {
    double sum = 0;
    for ( int i = 1; i<list_of_addresses.size(); i++ ) {
      Address address = list_of_addresses.at(i);
      sum += address.distance(list_of_addresses.at(i-1));
    }
    return sum;
  }
```

The index of the nearest address to any given address in the list is identified using the following function, which also utilizes the distance method from the Address class:

```
int index_closest_to(Address other) {
    Address closest = other;
    double closest_distance = 100000;
    int index = 0;
    for ( int i = 0; i<list_of_addresses.size(); i++ ) {
      Address address = list_of_addresses.at(i);
      double distance_between = address.distance(other);
      if (distance_between < closest_distance &&
        address.getdone() == false) {
        closest_distance = distance_between;
        index = i;
      }
    }
    return index;
  }
```

4.3 Optimizing a Single Route Using the Route Class

The Route class is a child of the Address class, and it is used to optimize a list of addresses that need to receive deliveries, starting and ending with a depot. The route class explores a handful of route optimization methods. The first

method is the "greedy" route, which sorts addresses such that the next address in a route is the closest one. The next implementation is the opt2 method, a method that "checks" whether or not a path crosses over itself and updates the route accordingly.

4.3.1 Greedy Route

Greedy scheduling is a method that does not seek to find the overall, or "globally," optimal route, but rather identifies the "locally" optimal solution. The function implemented accomplishes this by finding the nearest address to the current point and going there next.

The depot, or starting point, is added to the start of the route, then until the route is one address longer than the original address list (to account for the addition of the depot), the index closest to function from the Address List class is utilized to find the index of the nearest point at each address, then add the address of that index to the route. Once this has been performed for every address in the original address list, the new route is finished out by adding the depot to the end of the route, ensuring that the driver returns back to their starting point.

```
1  void greedy_route() {
2     Address we_are_here = depot;
3     route.add_address(we_are_here);
4
5     while ( route.getsize() < list_of_addresses.size() + 1  ) {
6       int min_index = index_closest_to( we_are_here );
7       list_of_addresses[min_index].complete();
8       we_are_here = list_of_addresses[min_index];
9       route.add_address(we_are_here);
10    }
11
12    route.add_address( depot );
13
14  }
```

4.3.2 Opt2

The opt2 method operates under the premise that if a path crosses itself, a shorter route may result from reversing a length of that route. [4] To employ the opt2 method, one must first use the greedy method to get a working route. Then, the opt2 method will iterate through each index of route, flipping that index and the next index. If the resulting length of the route is shorter, the route is updated and the opt2 process starts over again.

```
void opt2() {
  int m = 1;
  while ( m<route.getsize()-2 ) {
    //std::cout << "\nm = " << m << std::endl;
    AddressList new_route;
    int n = m+1;
    for (int i=0; i<route.getsize(); i++) {
      if ( i == m ) {
        new_route.add_address( route.getlistindex(n) );
      } else if ( i == n ) {
        new_route.add_address( route.getlistindex(m) );
      } else {
        new_route.add_address( route.getlistindex(i) );
      }
    }
```

4.4 Scheduling Multiple Trucks

The method for scheduling multiple trucks is also included in the Route class. To implement the two_trucks method, one must first utilize the greedy and opt2 methods to produce the optimal routes for two individual routes before running the two_trucks method.

```
void two_trucks( Route &other_route, bool flag ){
  AddressList other = other_route.get_route();
  AddressList new_route, new_other;
  double smallest_total = route.length() + other.length();
  double current_total = smallest_total;

  // for each index of one, iterate through two and see if
↪ swapping those two values will make one or the other shorter
  int i = 1;
  while ( i<route.getsize()-1 ) {
    //std::cout << "\nPASS " << i-1 << std::endl;
    int j = 1;
    while( j<other.getsize()-1 ) {

      // reassign new_one and new_two st they match one and
↪ two and the previous swaps are "erased"
      new_route = route;
      new_other = other;

      Address other_add = other.getlistindex(j);
      Address route_add = route.getlistindex(i);

```

```
        // flip index i in new_one and index j in new_two
        if (flag) {
          if ( !other_add.getprime() && !route_add.getprime() )
↪  {
            new_route.changeaddress(i, other.getlistindex(j));
            new_other.changeaddress(j, route.getlistindex(i));
          }
        } else {
          new_route.changeaddress(i, other.getlistindex(j));
          new_other.changeaddress(j, route.getlistindex(i));
        }

        // somewhere in here, a generator with a chance of
↪  adding a new address

        current_total = new_route.length() + new_other.length();

        if (new_other.length() < other.length() ||
↪  new_route.length() < route.length() ) {
          if (current_total < smallest_total) {
            smallest_total = current_total;
            route = new_route;
            other = new_other;
            i = 1;
            j = 1;
          }
        }

        j++;
      }
      i++;
    }

    std::cout << "final distance of two_trucks(): " <<
↪  route.length() + other.length() << std::endl;

    // update the input other_route by pass through reference
    other_route.set_route( other );

  }
```

Essentially, the two_trucks method is a re-implementation of the opt2 method for two routes. The method takes input of another route and a flag that indicates whether or not the process should adhere to packages scheduled using Amazon prime, which will be discussed in a later section. The two_trucks func-

tion iterates through each index of the first route and flips it with every index of the second route before testing for a shorter distance and updating the routes accordingly.

4.5 Implementing Amazon Prime and Dynamicism

The Route class also has a function for adding random addresses to the list, with the option to include the Amazon Prime method where some packages cannot be moved to a later truck.

4.5.1 Amazon Prime

As seen below, the Address Class has a prime Boolean, which is used to flag whether or not the two_trucks method will flip a pair of addresses and calculate the new pair of routes' total distance.

```
Address( double i, double j, bool isprime )
  :x(i), y(j), prime(isprime) {};
```

The two_trucks method will then check whether or not a package is prime, and it will either pass over the pair of addresses or flip them, as seen below. The method would then go on to calculate the change in distance resulting from the switched addresses.

```
        if (flag) {
          if ( !other_add.getprime() && !route_add.getprime() ) {
            new_route.changeaddress(i, other.getlistindex(j));
            new_other.changeaddress(j, route.getlistindex(i));
          }
        } else {
          new_route.changeaddress(i, other.getlistindex(j));
          new_other.changeaddress(j, route.getlistindex(i));
        }
```

4.5.2 Dynamicism

To test the real-world scenario in which packages are added to routes after the route has been set and before the truck goes out, a add_random_address method was added to the Route class.

```
void add_random_addresses(bool with_prime) {
  //generate a number for how many addresses
  std::random_device r;
  std::default_random_engine generator{ r() };
  std::uniform_int_distribution<int> distribution(0,25);
  int num_addresses = distribution(generator);
```

```
7
8       for (int i=0; i<num_addresses; i++) {
9         //generate an x
10        std::uniform_real_distribution<double>
    ↪   distribution(-10.,10.);
11        double x = distribution(generator);
12        //generate a y
13        double y = distribution(generator);
14        if (with_prime) {
15          //give it a prime value
16          std::uniform_int_distribution<int> distribution(0,1);
17          int prime_int = distribution(generator);
18          if (prime_int == 0) {
19            add_address( Address(x, y, false) );
20          } else {
21            add_address( Address(x, y, true) );
22          }
23        } else {
24          add_address( Address(x, y) );
25        }
26      }
27    }
```

This method uses the STL library to generate two random doubles for the Address' x and y coordinate, and, given a true with_prime flag, it will generate a 1 or 0 to assign true or false to the Address' prime Boolean.

In theory, this method would be implemented within the two_trucks method, which would continue to iterate through the two routes and all of their indices to find the shortest possible combined distance for the two routes. However, upon implementation, the code would copy addresses and refuse to pass the two_trucks method's other_route input as a reference. Please see the Dynamicism branch of the project's GitHub for more.

4.6 Results Analysis

Lastly, the Route class includes 2 print functions - the first directly prints the new route and its length in the power shell, while the other writes the resulting optimized list of addresses to .txt files that can be read in by the MATLAB Script routeplot.m. This script is used to read in the x and y coordinate files for the new route, then plot the full route, which can then be used for comparison between different optimization methods.

5 Discussion and Summary

5.1 Run Set-up

For the purposes of analyzing results, the following parameters were used:

1. Excluding the depot, 250 addresses will be generated per route
2. There will be no more than 2 routes
3. Coordinates can range in value from -10 to 10, and will always be a whole number.

5.2 Generating the Address List

Using the function discussed in section 3.1, two sets of coordinates were generated for use in the discussed results. The original plot for the points as generated for Routes 1 and Route 2 are as shown below in Figures 1 and 2, respectively.

As can be seen in the figures, the original order of addresses in the routes are extraordinarily disorderly. These routes are not suitable for a delivery route. Thus, other methods must be explored.

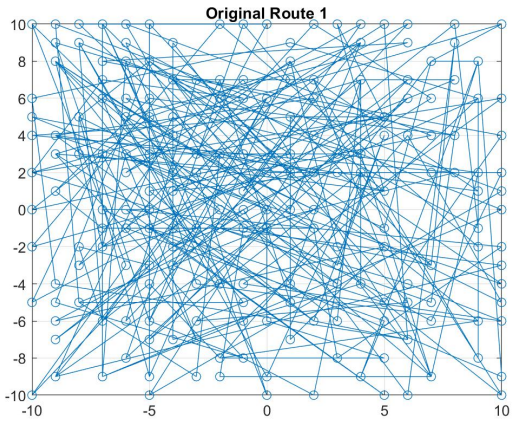

Figure 1: Plot of the generated points for Route 1

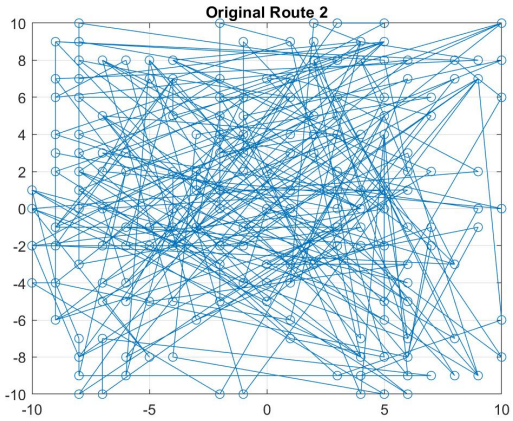

Figure 2: Plot of the generated points for Route 2

With respect to the generation of the points and plots for this section, where no optimization is attempted, all routes had a very low run time because no computation is being done on the points. As per the lack of computation, the routes are highly disorganized, sending the driver back and forth the plot several times. Once again, this is not an optimal route for a delivery service.

5.3 Optimizing a Single Route

5.3.1 Greedy Route

The first attempt at optimizing a route was the greedy_route method in which the closest point to a given address in the address list concerned is appended

to the end of a new route, directly following the address that the distance calculations concerned.

New plots for Routes 1 and 2 using the greedy_route method are as shown in Figure 3 and 4 below, respectively. The new length of Route 1 is 313.242 units, and the new length of Route 2 is 319.786 units.

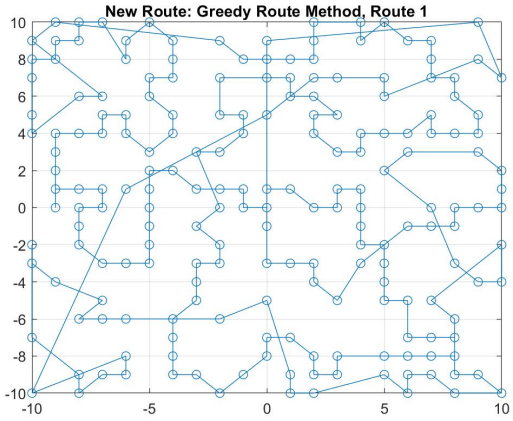

Figure 3: Plot of the Greedy Route generated points for Route 1

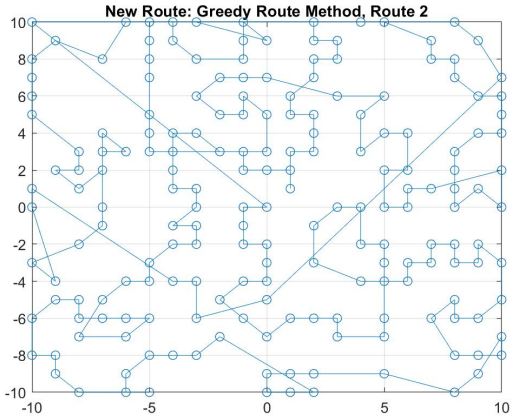

Figure 4: Plot of the Greedy Route generated points for Route 2

These routes are much easier to look at than Figures 1 and 2. There is a clear path and course of action, and thus a driver would not be driving back and forth according to the order in which packages were ordered. Clearly, these paths are much shorter, so drivers would spend much less time on the road delivering packages, which would be better both for the company and the driver.

However, there may be a way to further optimize the delivery paths.

5.3.2 Opt2

As discussed in methods section, opt2 checks for points at which the route "crosses" over itself, flips them, and effectively shortens the route as a whole. [4]

After having undergone opt2 optimization, the length of Route 1 amounted to 312.529 units, and the length of Route 2 was 316.793 units, amounting to a total of 629.322 units. Numerically, these improvements are marginal.

Visually, the results are marginal as well. However, there are instances in which the definition of opt2 can be observed. See for example in Figure 6, the opt2 plot for Route 2 around the (-8, 10) and (-7, 10) marks. When cross-referenced with Figure 4, the greedy_method plot for Route 2, one can see that those two addresses flip!

So, although the opt2 method doesn't provide much more optimization than greedy_route at this scale, it does still improve the route, producing lower route times and distances traveled for drivers.

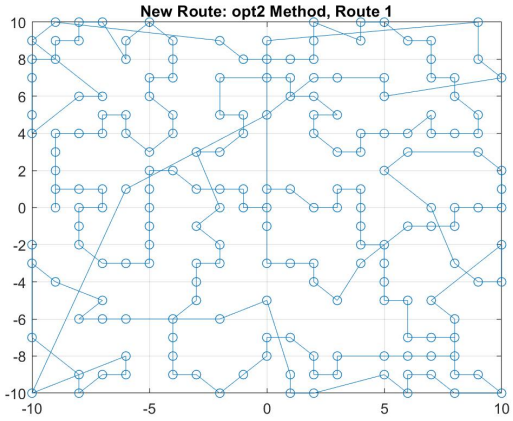

Figure 5: Plot of the opt2 Route generated points for Route 1

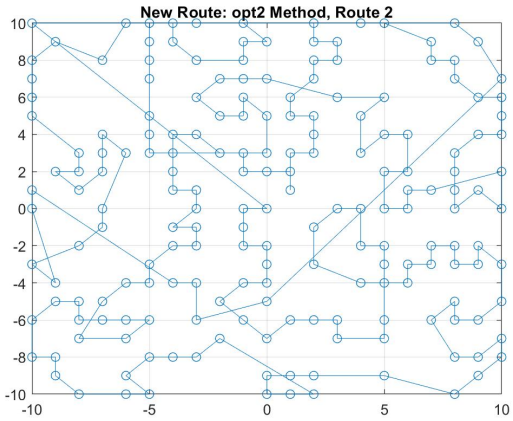

Figure 6: Plot of the opt2 Route generated points for Route 2

Up to this point, run times for all single route optimizations have been instantaneous. To see if the number addresses increase run time, a 3rd route with 1000 instead of 250 addresses was run using the opt2 method. The original route is shown in Figure 7 below, and is followed by Figure 8, which is a plot of the optimized route using opt2. The conclusion of this portion of the experiment was that there was no impact on run time from just 4x more addresses, but it is hypothesized that with orders of magnitude more points, the program will exhibit higher run times.

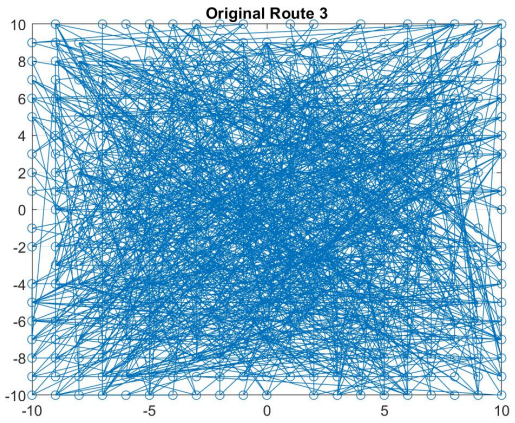

Figure 7: Plot of Route 3, a route of 1000 randomly generated points

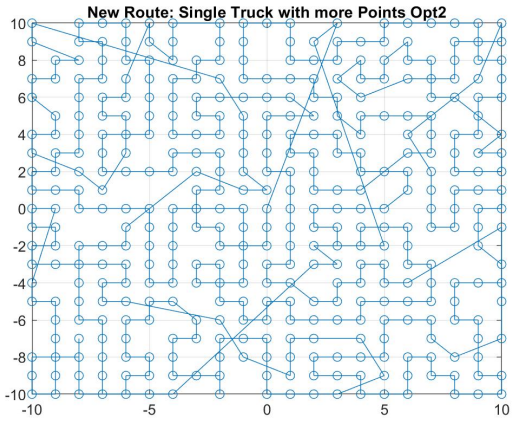

Figure 8: Optimized plot of Route 3 using opt2

5.4 Scheduling Multiple Trucks

Using the optimized Routes 1 and 2 from the previous section, two trucks with no prime packages were scheduled. The total route length across the two trucks was 546.614 units, which is much lower than the combined total route length of 629.322 units obtained from the opt2 method. Additionally, a stop watch collected run time of the program for scheduling two trucks averaged between 10-20 seconds. The plots of Route 1 and Route 2 after applying the two_trucks method is as shown in Figure 9 and 10 below, respectively.

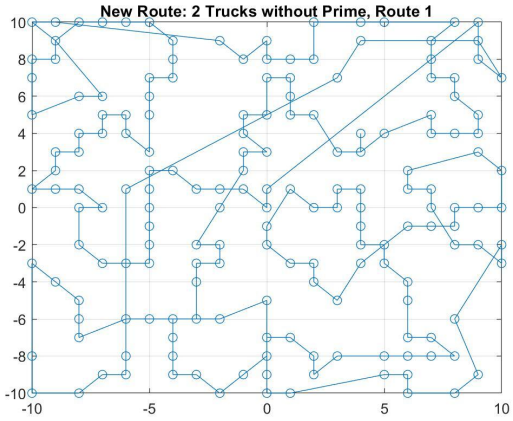

Figure 9: Plot of the Opt2 Route generated points for Route 1

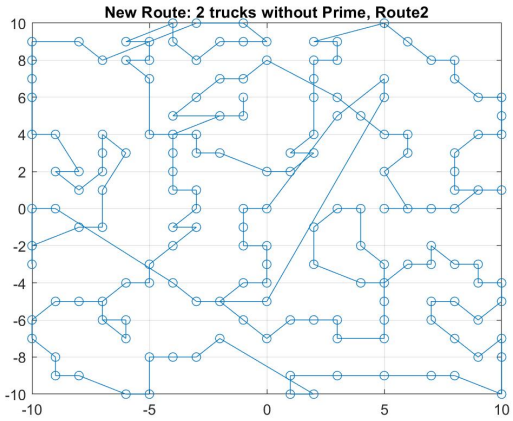

Figure 10: Plot of the Opt2 Route generated points for Route 2

When Figures 9 and 10 are cross-referenced with Figures 7 and 8, one can see the exchange of delivery addresses occurring. For example, in the bottom right quarter of Figure 10, specifically on the edges, several points are exchanged between the two routes, tending towards creating straight lines and minimizing turns.

5.5 Implementing Amazon Prime

Using the same routes as above, two_trucks with a randomly generated number of Prime packages are scheduled. Under this implementation, the total route length across the two trucks increased to 600.305 units. This is about 54 units greater than without Prime, proving that it is much more difficult to optimize a

route when there are constraints such as packages that must go out on one day or the other. Furthermore, the points exchanged between the routes don't always tend to minimizing as many turns as in Figures 9 and 10. Instead, Figures 11 and 12 below are a little more jagged, resembling Figures 7 and 8 more.

The new plots for Route 1 and Route 2 after applying the two_trucks method with Prime functionality are seen below in Figures 11 and 12, respectively.

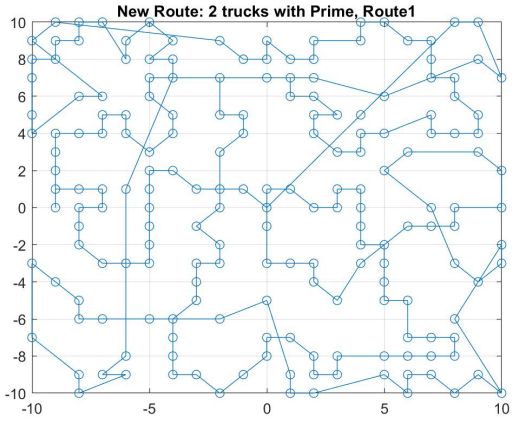

Figure 11: Plot of the Opt2 Route generated points for Route 1

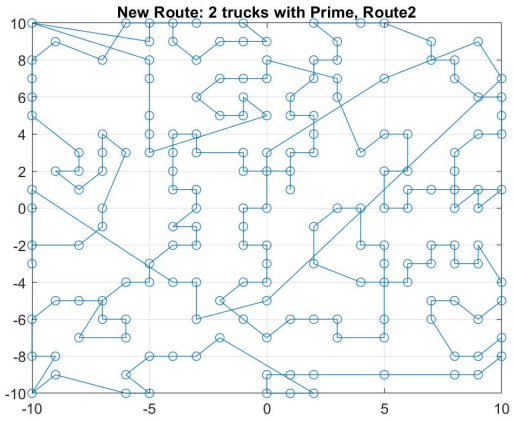

Figure 12: Plot of the Opt2 Route generated points for Route 2

For these deliveries, program run time also averaged between 10 and 20 seconds when using Amazon Prime, which is around the same amount of time as the program that did not adhere to Prime deliveries. It may be concluded that adding the functionality of Prime is not computationally heavy, but trying to optimize across many routes is.

5.6 Dynamicism

As per the Methods section, Dynamism was not achieved in this analysis. However, hypothesis for a successful implementation of a dynamic address allocation and optimization include that it would be more computationally expensive, as new packages would have to work their way through one route and then possibly be switched to another.

5.7 Ethics

Per an Amazon employee [4], most employees deliver 200-300 packages per day. With the optimizations that this project can make, this is within the lower range where there is *some* optimization, but nothing that would cut down the 20-30 packages/hour rate for a 10 hour day significantly.

Thus, ethically, the best way for Amazon to reduce hours for drivers would be to either create a better optimization algorithm, hire more drivers and coordinate across more trucks, or extend delivery package deadline ranges. The optimization route is likely the least expensive route and should be explored first, but with so few packages per truck, single route optimization is likely not as worthwhile, and most effort should be focused on coordinating across many trucks, and to accommodate Prime deliveries as well as randomly added orders.

6 Conclusion

6.0.1 General Conclusions

In this project, two methods, a greedy_ and a heuristic method called opt2 were used to optimize the routes for Amazon deliveries, then translated these algorithms over to optimizing routes across two trucks, which can have packages swapped between them. Lastly, a function for applying dynamicism, or the possibility of random packages being added to the fleet and needing to be included in the route optimization, was designed.

From testing this program for 250 addresses, it was proven that both the greedy and the opt2 methods were successful in shortening the original route, and patterns in the graphs of these routes show a much cleaner, geometric result than the scattered lines from a randomly generated path. However, with low address counts, it was proven that opt2, which is theoretically more computationally expensive, is not that much better at optimizing than the greedy route. The tests concluded that greedy routing can be used for short address lists less than 300 points, but above that, opt2 becomes more worthwhile.

The run time for all of the single route optimizations was practically instantaneous, for anywhere from 0 to 1000 packages. Run time only began to increase when more than one truck was being routed at a time. There was not a clear difference between running with Prime enabled or without. It can be concluded that the more routes to optimize across, the longer run time will be, by almost 10x.

6.0.2 Future Work

To build upon this project in the future, the team would implement far more trucks (10, for a small city) and then run a more realistic approximation of how many packages there are, and how many might get added randomly. It would also be interesting to implement scheduling across a standard week and work hours, considering shifts and stops back at the depot to add more packages, and this functionality would build a software much more realistic to what Amazon has to use to coordinate on a daily basis.

7 Bibliography

1. "Amazon Statistics and Facts," Market.us. https://market.us/statistics/e-commerce-websites/amazon/

2. DID YOU KNOW?" [Online]. Available: `https://0ca36445185fb449d582-f6ffa6baf5dd4144ff990b4132ba0c4d.ssl.cf1.rackcdn.com/IG_360piAmazon_9.13.16.pdf`

3. "6 charts show how a pandemic upended retail supply chains," SupplyChainDive. `https://www.supplychaindive.com/news/retail-pandemic-ship-store-curbside-pickup-amazon-charts-\newlineomnichannel-delivery/596296/`

4. Nevin, "Would a ups,usps,fedex, or a amazon driver know what this is?," Amazon Available: `https://www.amazon.com/ask/questions/`

5. V. Eijkout. "Introduction to Scientific Programming book," COE322. [Online].