Scientific Computation Final Project: Amazon Delivery Truck Scheduling
Professors: Dr. Victor Eijkhout & Professor Susan Lindsey

Students: Arushi Sadam and Nicole Olvera
UT EID: ars7724 & no4342
TACC usernames: rx9933 & nicoleolv
Email: arushi.sadam@utexas.edu & nicoleolv@utexas.edu

Amazon Delivery Truck Scheduling

Arushi Sadam, Nicole Olvera

December 5, 2023

Contents

1 Introduction

In the realm of e-commerce, efficient delivery route optimization is crucial for prompt package transportation from warehouses to customers. For the world's largest delivery company, Amazon [1], drivers, who may handle up to 350 deliveries daily [2], face increased pressure during high-demand periods like Thanksgiving and Christmas[3]. With Amazon employing over a quarter-million drivers worldwide [1], safe and swift product delivery through optimized routes is essential. This paper examines the operational and ethical aspects of potential Amazon delivery route optimizations, which could increase customer satisfaction and profit while also improving resource management and, of course, driver welfare.

Scheduling optimized routes for delivery trucks is a well-studied problem based upon a non-deterministic polynomial-time hardness (NP-Hard) problem of a traveling salesman [4]. This project seeks to minimize the delivery cost for a set of packages; either one or two salesmen can be employed, where the salesmen must begin and return to a fixed depot. For Amazon Prime deliveries, a new constraint arises: prime customers are prioritized since their packages must be delivered within a day. Additionally, an arbitrary number of new orders can be added to the pre-existing delivery list, which requires a dynamic model of delivery truck(s) scheduling.

All in all, this project explores various optimization methods and determines the best algorithm (out of three methods), which results in up to a 60% cost reduction for Amazon when compared with a random delivery route.

2 Implementation

Rather than minimizing the route distance that the drivers/salesmen must travel to deliver all packages, this model seeks to minimize Amazon's delivery/shipping cost, thus ultimately improving Amazon's profit. When two drivers are available as opposed to just one, minimizing route distance will always lead to only one driver being employed; the distance when both drivers are working will always be greater than or equal to the distance when only one driver works. Since the time until package delivery must also be considered, a cost parameter is generated based on the minimum wage of a driver, a time cost (greater cost for delayed prime deliveries versus delayed regular deliveries), and the fuel cost for traveling the route.

For the dynamic model, new deliveries are added to the route after the drivers have departed from the depot. The drivers must return to the depot to pick-up the package before setting out again to deliver to these new addresses. Given that the new deliveries can also be Amazon Prime deliveries, the model must optimize both the cost of delaying Amazon Prime deliveries and the cost of returning to the depot from a potentially far current location (versus picking up and delivering the new prime packages later, at a more convenient time).

This project uses C++ programming code to model and optimize routes, and Python (matplotlib) to generate the

data visualization of various routes.

2.1 Program Layout

Four key classes are implemented to handle individual addresses, vectors of addresses, and the manipulations of these vectors. Overall, they provide a set of functionalities for route optimization based on the calculated cost for a given delivery system.

2.1.1 Address

Each delivery address is assumed to be part of a 2D grid; addresses are stored as pairs of integer coordinates, `(i,j)`. Each address is tied with a `bool primeAddress` that marks an address as requiring Amazon Prime (true) or regular (false) delivery. The distance method calculates the Euclidean distance, not the Manhattan distance. Rather than the structured, perpendicular roads (like a grid) that are assumed by the `manhattan_distance`, the salesmen are to travel in straight lines between each address. The `Address` class holds a few other functions that allow for efficient testing/output, such as the `is_prime`, `is_depot`, and `supplies` functions.

2.1.2 AddressList

The `class AddressList` manages addresses. It has two vectors (of type `Address`) to keep track of all orders (`addresses`) and delivered packages (`delivered`). It also has methods for adding (known and random addresses) or removing already-visited addresses, and, most importantly, optimizing delivery routes. The `add_address` method uses the vector `push_back()` function while the `remove_address` method uses the vector `erase` function. Additionally, this class contains one of the optimizing methods, `greedy_route`. Supporting functions for `greedy_route` like `index_closest_to` are also located in `class AddressList`. Another important function is the `insertShuffle` function which introduces randomness and variability in a route subvector; this allows for new routes to be created efficiently. The `AddressList` class provides different methods for adding/removing addresses and is the foundation for the more complex optimization functions found in `class Route` and `class Amazon`.

2.1.3 Route

The `Route` class is derived from the `AddressList` class, allowing the inheritance (access) of all methods and properties of `AddressList`. However, it is a specialization that is used to model a particular route. Correspondingly, it holds two fixed depots at arbitrary (0,0), one at the beginning and the other at the end of the route, and ensures that addresses are added before the last element (the depot). This class holds an extra parameter, `fullySupplied`, which ensures that the driver has all the packages for the addresses within the route; this function is crucial for dynamicism.

For optimization methods, the `reverse_route` function assists in the implementation of the `opt2` heuristic, another optimization strategy. Additionally, this class introduces methods for calculating fuel, driver, and time costs: `fuelCost`, `driverCost`, `time`, and `timeCost`. All of these allow the program to calculate `totalCost`, which quantitatively measures the optimization of a route.

2.1.4 Amazon

The `Amazon` class is derived from the `Route` class. It adds two new methods, `combineSegments` and `multiPaths`, which allow for the optimization of multiple (a maximum of two) salesmen. This is the last class required for a basic model of Amazon deliveries.

2.2 Amazon's Cost

In this program, cost is used to quantitatively measure and compare different optimization methods. Cost encompasses various factors, including fuel price, driver's wage, and time-related costs (time/distance taken before delivery).

The method `fuelCost` calculates the total cost for fuel, which is based on the total distance traveled; the variable `fuelPrice` is assigned to an arbitrary amount of fuel required per unit distance traveled (1.0 dollar/mile). The product of the length of the given route(s) (number of miles) with the current constant `fuelPrice` (dollars per mile) is returned by method `fuelCost`.

The method `driverCost` calculates the minimum wage of a salesman; if the driver drives more than 10^{-6} units of distance, he/she is employed and must be paid the constant value of the `driverBasePrice` (0.5 dollars):

```
if (l1 > 1e-6) //truck has moved, l1 is input distance
  cost += driverBasePrice;// returns a price, dollars,
return cost;
```

The method `timeCost` finds the cost Amazon pays to ensure quick deliveries. For this, Amazon prime deliveries are more heavily weighted than regular deliveries. The function `time` returns the delivery time cost. When called on a route object, `time` considers the number of prime/non-prime customers who have not received their package yet against each step the salesman takes. Two vectors, `primes` and `nonPrimes`, check the number of prime/non-prime deliveries left at each index of the route. First, the number of already-visited primes is calculated for each of the route's addresses:

```
for (int adr = 1; adr < this->size(); adr++) // per address
  {
    bool isPrime = this->addresses[this->size() - adr - 1].is_prime(); //ignore last depot
    if (isPrime) { totPrime++; } // find total number of prime, nonprime addresses
    else { totNonPrime++; }
    primes.push_back(totPrime); // append the total number of primes already visited
    nonPrimes.push_back(totNonPrime); // total number of primes not visited
```

```
    }
```

Then, the order of the `prime` and `nonPrime` vectors is reversed; now, the `prime` and `nonPrime` represent the number of packages (of each type) waiting for delivery at any given stop/address. Finally, for each step or distance the salesman travels between delivery stops, a customer wait time cost, `custTimeCost`, is incremented:

```
for (int adr = 1; adr < this->size() - 1; adr++) // not counting depot / return time from last address
 {
     step = this->addresses[adr - 1].distance(this->addresses[adr]); // previous step matters to future
  + present customers
     custTimeCost += (primes[adr] * primeTimeCost + nonPrimes[adr] * normalTimeCost) * step; //
  cost of each step is based on the cost each prime or nonprime delivery has to wait
 }
 return custTimeCost;
```

Variables `primeTimeCost` and `normalTimeCost` are constant parameters of unit money per mile driven before delivery:

```
 float primeTimeCost = 2; // dollars per mile driven before prime delivery
 float normalTimeCost = 1; //dollars per mile driven before regular delivery
```

The sum of all of these functions/factors builds the `totalCost` function:

```
float totalCost(Route newPath1)
{
 float l1 = newPath1.length(); // route length for a given salesman
 return fuelCost(l1) + driverCost(l1) + timeCost(newPath1); // sums up all net costs
}
```

2.3 Optimization Strategies

The three strategies that are considered for optimizing cost, in order from least to most effective, are:

1. The greedy search strategy
2. The opt2 heuristic
3. A combination of the greedy and opt2 method

Note that since the greedy and opt2 strategies are based on local optimality; none of these three methods will necessarily lead to the best-possible route, which can only be found by searching through all possible routes [4].

2.3.1 Greedy Route

The `greedy_route` search method uses a strategy that optimizes each new address (next address) individually. From a given address, it finds the closest unvisited customer address (Figure 1). When all product deliveries are completed, the depot becomes the next address of the route:

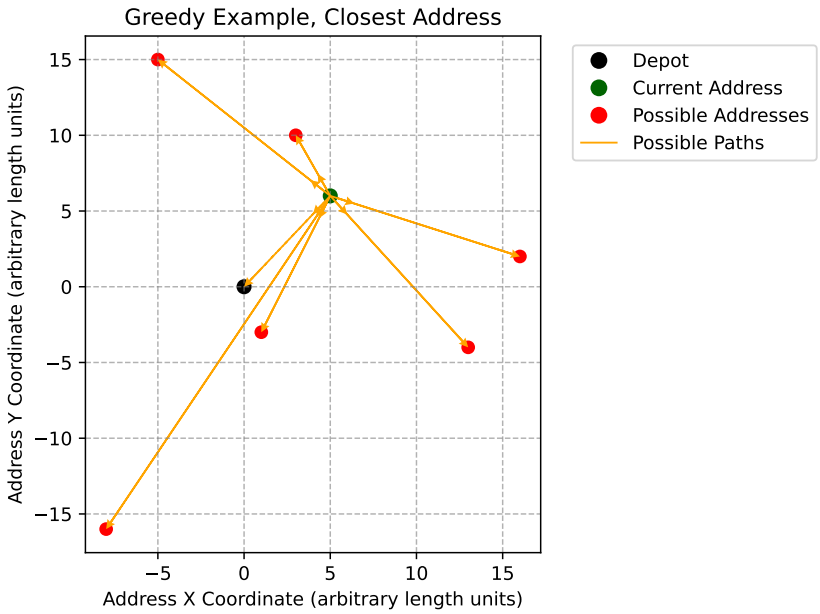

Figure 1: Finding Closest Address

The `greedy_route` method is found in `class AddressList`. First, an `AddressList` of non-visited/remaining addresses is initialized, as a copy of the original addresses: `AddressList remainingAddresses(*this)`. `greedy_route` loops through each element of `remainingAddresses` and finds the closest address to the present address, using the `index_closest_to` method, and sets the closest address as the new current address. The method `index_closest_to` (in `class AddressList`) loops through each element of the `remainingAddresses` and checks the distance from the current address to each of the remaining customer addresses (Figure 1). The closest element is stored and returned to the `greedy_route` method and is removed from the `remainingAddresses` vector. These elements/addresses are stored in an addresses vector of object `AddressList route`, which is initialized with the depot at the start. Key steps of the `greedy_route` method are shown below:

```
AddressList greedy_route() const
{
    AddressList remainingAddresses(*this);
    route.add_address(addresses[0]);
    while (remainingAddresses.size() != 0) // all deliveries are not completed
    {
```

```
        Address closest_address = remainingAddresses.index_closest_to(current);
        //.... ensures no addition of depot
        current = closest_address;
        remainingAddresses.remove_address(current);
      }
      route.add_address(addresses[0]); // add final address to return to depot
      return route;
  }
```

The `index_closest_to` method takes a reference to the current address (`const Address& other_point`), and returns the `closest_address`). The `closest_distance` is initialized to be the max floating point value:

```
float closest_distance = std::numeric_limits<float>::max();
```

Then, excluding the beginning and end address (start and end depot addresses), the method searches for and saves the address with the least Euclidean distance to the input address (a `distance` method in `class Address` finds the distance between two given points.)

The following code snippet shows the main steps of the `index_closest_to` method:

```
  for (int i = 1; i < addresses.size() - 1; ++i)
  {
      float distance = addresses[i].distance(other_point);
      if (distance < closest_distance)
      {
          closest_address = addresses[i];
          closest_distance = distance;
      }
  }
```

2.3.2 Opt2 Route

The 2-opt or opt2 heuristic is a search algorithm that suggests a route that crosses over itself can be shortened by reversing a portion of the points/addresses within it. The `opt2` method loops through different subvectors (based on pairs of starting and ending addresses), and reverses the addresses order of the particular subvector. If an improvement in the route cost is found, then this new route becomes the `Route optimizedPath`. The algorithm continues to loop through different subvectors of the `Route optimizedPath` until there are no new subvector indices or no more improvement can be found. Note that the opt2 heuristic does not necessarily find the best route. The algorithm only explores local improvements that might converge to a suboptimal solution, since it does not consider the entire solution space.

The `opt2` method in `class Route` calls a `reverse_route` method to create new permutations of addresses. The

method `reverse_route` takes two integer indices as inputs, and reverses the addresses subvector between the two indices, using the vector `reverse` function:

```
Route reversedList(*this);
auto it_begin = reversedList.addresses.begin() + m;
auto it_end = reversedList.addresses.begin() + n + 1;
reverse(it_begin, it_end);
return reversedList;
```

First the new route, `Route reversedList` is initialized to the object the function is being called upon. Then, given the indices `m` and `n`, the `reverse` method is called on the subvector of the `reversedList` (spliced `reversedList` between `m` and `n`). Finally, the entire `reversedList` is returned. Consider an example of addresses where $m = 1$ and $n = 4$ (Table 1). Given a list of addresses with their indices:

Indices	0	1	2	3	4	5	6
Addresses	A	B	C	D	E	F	A

Table 1: Reverse Route Example, Initial Addresses

where A represents the depot (drivers start and end the route at the depot), the following statements must always be true:

$$1 \leq m < addresses.size() - 2 \tag{1}$$

and

$$m + 1 \leq n < addresses.size() - 1 \tag{2}$$

This way, the depots are never swapped/reversed in order; they are fixed at the ends of the addresses vector. Also, $m < n$ allows the `reverse_route` method to always have valid inputs. After the `reverse_route` method (Table 2), the order of addresses of route `reversedList` will be:

Indices	0	1	2	3	4	5	6
Addresses	A	E	D	C	B	F	A

Table 2: Reverse Route Example, Reversed Addresses

The method `opt2` is located in `class Route`. The `opt2` method creates combinations for valid `m` and `n` values, calls the `reverse_route` with each set of `m` and `n`, and checks for improvement in cost. Two booleans check for change and improvement in the route(s)'s cost: `change` and `isBeingOptimized`. `change` checks if the new cost, after route reversal, is less than the initial cost:

```
bool change = newCost < minPathCost;
```

The difference between `change` and `isBeingOptimized` is apparent when a dynamic route is employed–new addresses are added when the trucks have already started their routes (view Dynamic Addresses section). `change` ensures drivers have the required packages before delivering to any given address.

While the route is still being optimized (`isBeingOptimized` is true), a double `for loop` goes through all possible valid combinations of `m` and `n` satisfying equations 1 and 2:

```
isBeingOptimized = false;
for (int m = 1; m < addresses.size() - 2; m++)
{
   for (int n = m + 1; n < addresses.size() - 1; n++)
   {
```

`isBeingOptimized` is reassigned to false during each iteration, and changed when there is an improvement in the route. Within these for loops, if `change` is true, then the `optimizedPath` is reassigned to the new path, the minimum path length is reassigned to the new cost, and `isBeingOptimized` is reset to true. After all combinations of `m,n` are finished, the final optimized path is returned.

2.3.3 Method Combinations: Greedy-Opt2 Route

The greedy-opt2 route is a combination of the greedy and the opt2 strategies. While the greedy search attempts to quickly construct an improved route, based on locally optimized steps, the opt2 search is more thorough, iteratively improving the solution through reversals of route sections. With this combination, the greedy algorithm could set a solid near-solution route, which the opt2 algorithm could polish by eliminating crossings and resolving local inefficiencies.

2.4 Two Salesmen Optimization

With two salesmen, a new option is added: deliveries can be swapped between salesmen, as long as they are non-prime orders. By iterating through various address swaps between salesmen, applying the greedy or opt2 methods on individual salesmen, and comparing cost changes, path improvements can be made.

2.4.1 Swapping Between Salesmen

The main function that isolates non-primes and swaps them between salesmen is `combineSegments`, located in `class Route`. Function `combineSegments` takes two routes, `path1` and `path2` as input, along with two sets of integers: `m1`, `n1` and `m2`, `n2`. Like in the `reverse_route` function, `m` and `n` define indices to be swapped between salesmen. Addresses between `m1` and `n1` in `path1` are swapped with the addresses between `m2` and `n2` in `path2`, with exception of prime addresses in each of the subvectors. `combineSegments` returns a `tuple` of routes (one per each salesman).

First, two new routes are initialized: `newPath1` and `newPath2`. To eliminate any initial depots (per the `class Route` constructor), a method, `clear` is used to remove all initial addresses in the route. Now, `newPath1` and `newPath2` are completely empty. This step is important for the dynamic model, where the first address of the salesmen's daily routes may not necessarily be the depot but is instead the previous day's last delivered address. Each of the new paths can be split into 3 sections: the first part until `m`, which remains the same, the second part between `m` and `n` which is swapped between the drivers (with the exception of prime addresses), and the third part from `n` to the end of the input paths.

As an example (Table 3), let salesman 1 and salesman 2 each have six arbitrary addresses. Let `m1 = 2, n1 = 5, m2 = 1, n2 = 3`. Also, let prime addresses in `path1` be B and E and prime addresses in `path2` be I and J.

Indices	0	1	2	3	4	5	6
`path1`	A	B	C	D	E	F	G
Indices	0	1	2	3	4	5	6
`path2`	H	I	J	K	L	M	N

Table 3: Swapping Between Two Salesmen, Before

The first part of `newPath1` and `newPath2` remains the same as `path1` and `path2`: `newPath1` = A-B-C and `newPath2` = H-I. The section of `path1` between indices 2 and 5 (D-E-F) is swapped with the section from `path2` between indices 1 and 3 (J-K); however, address E and address J are prime addresses. So, D-F (`nonPrimes1`) from `path1` is swapped with just K (`nonPrimes2`) from `path2`. Address E remains in the middle section of `path1` just as address J remains in the middle section of `path2`. Now, the middle section for `newPath1` will be a randomly shuffled combination of `primes1` and `nonPrimes2` (E-K becomes K-E) while `newPath2` will be a combination of `primes2` and `nonPrimes1` (J-D-F becomes D-J-F). Finally, the ending sections of `newPath1` and `newPath2` remain the same: G and L-M-N, respectively. Ultimately, the final paths change as shown in Table 4:

Indices	0	1	2	3	4	5		
`path1`	A	B	C	K	E	G		
Indices	0	1	2	3	4	5	6	7
`path2`	H	I	D	J	F	L	M	N

Table 4: Swapping Between Two Salesmen, After

A more detailed step-by-step process is described below.

In place of the initializing depot, the first addresses for either path remains the same as the input `path1` and `path2`:

```
newPath1.addresses.insert(newPath1.addresses.begin(), path1.addresses.begin(),
 path1.addresses.begin()+m1);
newPath2.addresses.insert(newPath2.addresses.begin(), path2.addresses.begin(),
 path2.addresses.begin()+m2);
```

Then, the prime and non-prime addresses for `path1` and `path2` are sorted by calling the `getPrimesAndNonPrimes` function from the function `combineSegments`. Given two indexes, `start` and `end`, and a reference to a route `path`, `getPrimesAndNonPrimes` loops through each of the elements of `path` within the range from `start` to `end` and checks if that particular address is prime, using the `is_prime` function in `class Address`. If it is a prime address, the address is appended to a vector of primes, otherwise, it is appended to a vector of non-primes:

```
for (int i = start; i < end; i++)
{
 if (path.addresses[i].is_prime())
 {
   primes.push_back(path.addresses[i]);
 }
 else
 {
   nonPrimes.push_back(path.addresses[i]);;
 }
}
```

After looping through each element of the subvector from input `start` and `end` indices, `getPrimesAndNonPrimes` returns a tuple of the `primes` and `nonPrimes` vectors.

The second/middle part of `newPath1` contains the prime addresses in `path1` from between indices `m1` and `n1` (`primes1`) and the non-prime addresses in `path2` from indices `m2` and `n2` (`nonPrimes1`). The middle section of `newPath2` follows the same pattern: it contains the prime addresses in `path2` from between indices `m2` and `n2` (`primes2`) and the non-prime addresses in `path1` from indices `m1` and `n1` (`nonPrimes2`):

```
auto [primes1, nonPrimes1] = getPrimesAndNonPrimes(path1, m1, n1);
auto [primes2, nonPrimes2] = getPrimesAndNonPrimes(path2, m2, n2);
```

When inserting this middle section, since the optimal/preferred order of prime and non-prime addresses is unknown, this section is randomly shuffled, and then inserted into the respective new paths. Given as input a vector of prime addresses `primes`, a vector of non-prime addresses `nonprimes`, and an `AddressList` of addresses `AddressList path`, the function `insertShuffle` inserts a shuffled set of the input `primes` and `nonprimes` at the end of the `path`:

```
primes.insert(primes.end(), nonprimes.begin(), nonprimes.end()); //combines primes and non-primes
std::shuffle(primes.begin(), primes.end(), rng); //shuffles addresses
```

```
path.addresses.insert(path.addresses.end(), primes.begin(), primes.end()); //inserts addresses at
 end of path
```

Finally, the last part of newPath1 and newPath2 remains the same as in path1 and path2. The addresses from n1 to the end of path1 are inserted at the end of newPath1, and a corresponding procedure is done to newPath2:

```
newPath1.addresses.insert(newPath1.addresses.end(), path1.addresses.begin()+n1,
 path1.addresses.end()); //does not add additional depot/last address at end
newPath2.addresses.insert(newPath2.addresses.end(), path2.addresses.begin()+n2,
 path2.addresses.end());
```

2.4.2 Optimizing Each Set of Routes

With the function combineSegments putting together new routes based on preliminary conditions (m and n values) and the basic single salesman route optimizing functions (like greedy_route and opt2), a function that can combine both these components is the last component to building a basic model for optimizing two salesmen routes. Function multiPaths does this. multiPaths takes two initial routes, path1 and path2 as input, along with a couple of booleans that set if the routes are dynamic and if the greedy search strategy and/or opt2 strategy should be used:

```
std::tuple<Route, Route> multiPaths(Route path1, Route path2, bool dynamic = false, bool greedy =
 false, bool opt = true)
```

The default setting for multiPaths is a non-dynamic opt2 optimization. multiPaths has a similar structure to opt2: a while loop checks for optimization (bool isBeingOptimized = true). For more iterations, in hopes of getting better-optimized solutions, the while loop is repeated five times. Within the while loop, four nested for loops go through each possible combination of m1, n1, m2, and n2:

```
for (int m1 = 1; m1 <= optimizedPath1.addresses.size() - 1 && !restart; m1++)
  {
    for (int n1 = m1; n1 <= optimizedPath1.addresses.size() - 1 && !restart; n1++)
    {
      for (int m2 = 1; m2 <= optimizedPath2.addresses.size() - 1 && !restart; m2++)
      {
        for (int n2 = m2; n2 <= optimizedPath2.addresses.size() - 1 && !restart; n2++)
        {
```

When m1=n1 or m2=n2, addresses are added, rather than swapped, between the two salesmen's routes. There is one extra case, where m1=n1 and m2=n2, where there is no change to the routes. Since both the greedy and opt2 methods do not necessarily lead to the most optimized routes, more optimization iterations (attempts) are added; an additional nested for loop is used. For each set of m1, n1, m2, and n2 ten iterations or attempts through the optimization functions (greedy_route, opt2, or the greedy-opt combination) are used, for more attempts at a better solution.

With the five nested `for loops`, first, the `combineSegments` function is called, returning two Routes: `newPath1` and `newPath2`. Then, based on the input booleans (`opt` and `greedy`), the `newPaths` are optimized with either the `greedy_route` method, the `opt2` method, or both the `greedy_route` and `opt2` method:

```
if (greedy)
{
  //calls greedy_route on both newPath1 and newPath2
}
if (opt)
{
  //calls opt2 on both newPath1 and newPath2, also takes as input bool dynamic
}
```

The total cost of both new paths is calculated. Then, to compare the new cost with the previous minimum cost, a process similar to the `opt2` function is used: a boolean `change` checks for improved cost, and if a dynamic model is being used, `change` also ensures the salesmen have all the packages before visiting the delivery addresses:

```
bool change = newLength < minPathLength;  // checks for improvement
if (dynamic)  // and applies to check for packages
  {
    change = change && newPath1.fullySupplied() && newPath2.fullySupplied();
    // checks for depot before all new/added addresses
  }
```

Finally, if an improvement is found, if `change = true`, the `optimizedPaths`, `optimizedPath1` and `optimizedPath2` are reassigned to `newPath1` and `newPath2`, respectively. The minimum cost, `minCost` is also reassigned to the new cost, `newCost`:

```
if (change)
{
  isBeingOptimized = true;
  optimizedPath1 = newPath1;
  optimizedPath2 = newPath2;
  minCost = newCost;
}
```

At the end of the `while loop`, as a check for the dynamic model, if both the `optimizedPath1` and `optimizedPath2` are not `fullySupplied` (have packages before traveling to delivery addresses), boolean `isBeingOptimized` is reset to `true`; the `while loop` repeats until the optimized paths have supplies and are not being optimized anymore:

```
while (isBeingOptimized...)
{ ...
    for (int m1 = 1; m1 <= optimizedPath1.addresses.size() - 1 && !restart; m1++)
```

```
    { ....
    }
    if (dynamic) //checks for packages once we found the most optimized route
    {
       if (!(optimizedPath1.fullySupplied() && optimizedPath2.fullySupplied()))
        {
            isBeingOptimized = true; // continue checking for optimial solution, conditions not satisfied
        }
    }
}
```

Finally, a tuple of the two routes: `optimizedPath1` and `optimizedPath2` are returned.

2.4.3 Dynamic Addresses

The `opt2` method is initialized to not be dynamic:

```
Route opt2(bool dynamic = false)
```

When `bool dynamic = true`, `bool change` checks that there is an improvement in cost and that the new route's addresses all have supplies.

```
if (dynamic)
{
    change = change && newAddressOrder.fullySupplied();
}
```

The method `bool fullySupplied` in the `class Route` checks whether all addresses in the route, excluding the depot, have their supplies/packages available. For the new addresses, an extra depot stop must be added. `fullySupplied` finds the middle depot iterator in the addresses vector using the `std::find` function:

```
std::vector<Address>::const_iterator depotIndex = std::find(addresses.begin() + 1,
 addresses.end(), Address(0,0));
```

Then, it iterates through each address (excepting the initial depot) until the middle depot. For each address, it checks if its supplies are available using the `supplies()` method, which returns a simple `bool haveSupplies`:

```
for (auto i = addresses.begin()+1; i < depotIndex; i++)
{
    if (!i->supplies()) {return false; } // does not have package
}
```

For new addresses, the `haveSupplies` is initialized to false in the new address constructor. If any address does not have supplies, the function `fullySupplied` returns false. If all addresses are supplied, it returns true.

3 Results

3.1 Single Salesman

When optimizing the cost of addresses with a single driver or salesman, the following plots are generated.

3.1.1 Runtime/Speed Performance

When the greedy, opt2, and greedy-opt2 search are done on a single salesman (Figure 2), the runtime seems to have an order of n^4 where n represents the number of addresses input to be optimized (size):

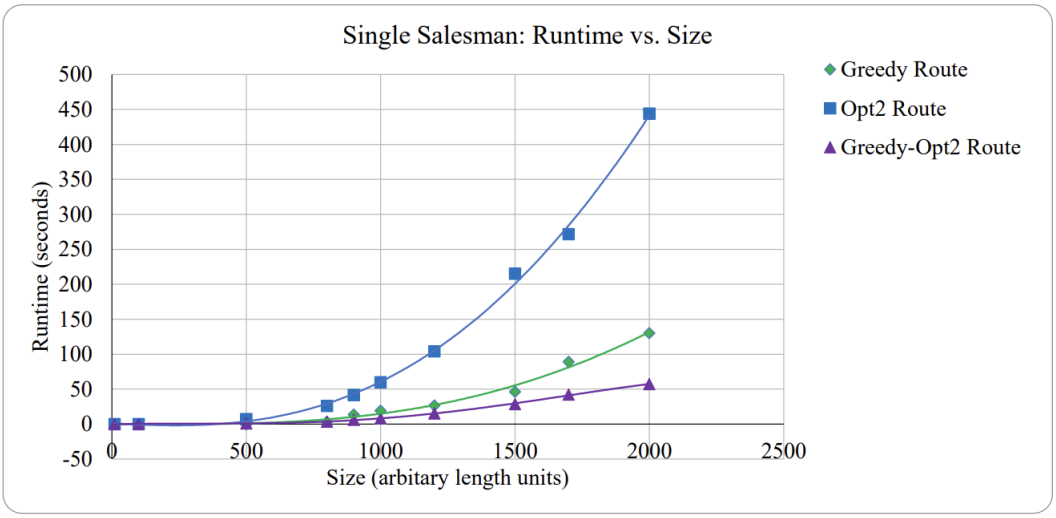

Figure 2: Runtime of Greedy, Opt2, and Greedy-Opt2 Search

Note that the greedy-opt2 search leads to significantly reduced runtime than the simple greedy search. Given that a more comprehensive algorithm, which tries out all possible combinations, grows faster than a polynomial [4] (but less than exponentially [5]), a quartic equation, which is much smaller, aligns with our model; all three optimization techniques do not provide a comprehensive, absolute solution.

3.1.2 Accuracy/Improvement Performance

Data for an example route optimized through the greedy, opt2, and greedy-opt2 routes is presented below.

The initial route (Figure 3) is a set of random input addresses, with no order/optimization in place:

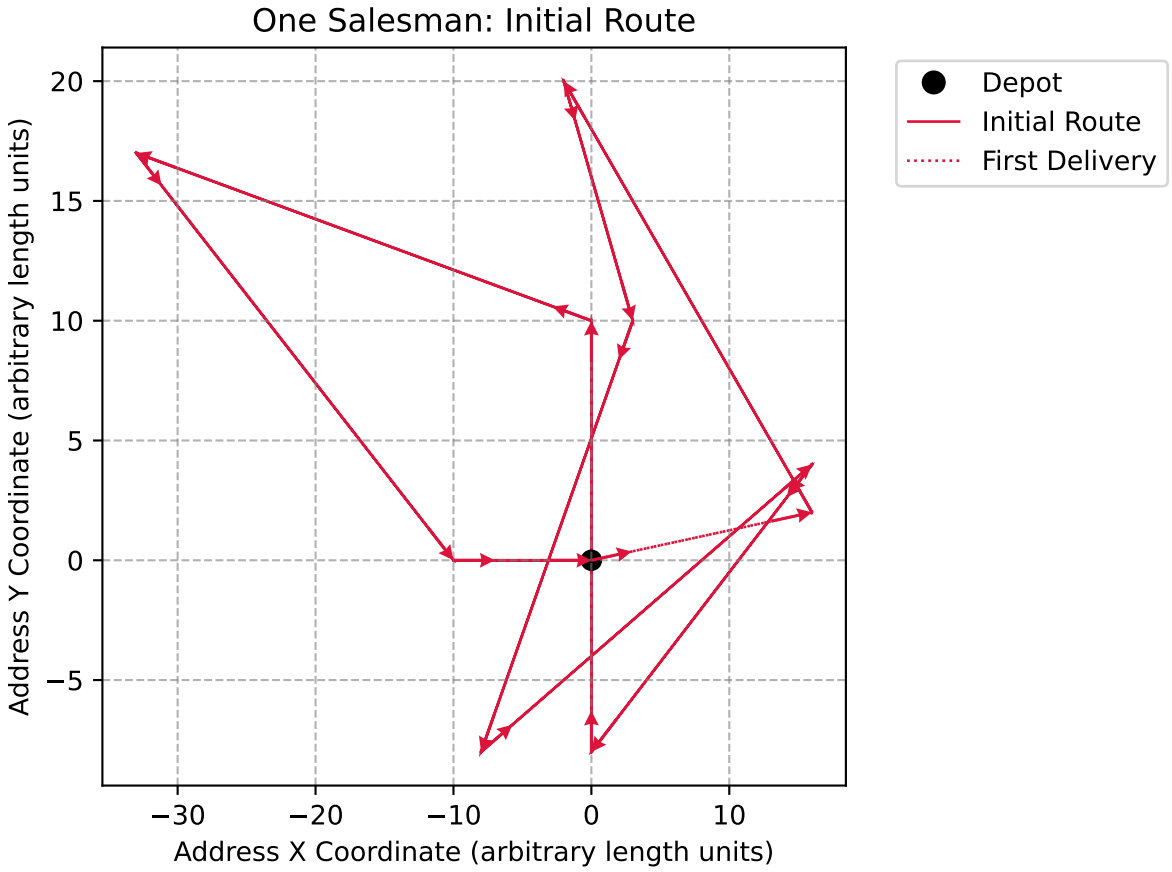

Figure 3: Initial Route

This route crosses itself seven times and has a total route cost of $1615.24.

By employing the greedy search optimization (Figure 4) on a single driver, the following route is generated:

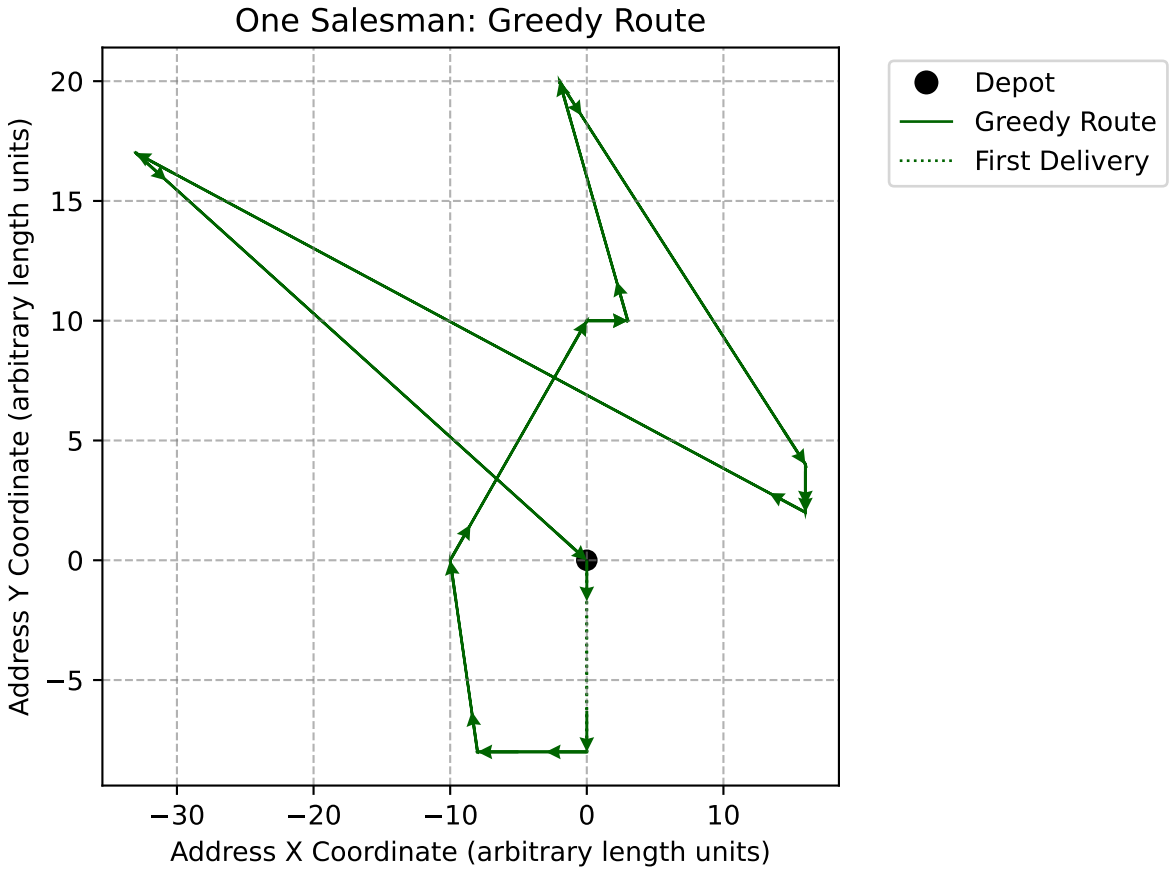

Figure 4: Greedy Route

The greedy route crosses itself only two times and has a cost of $687.698, a more than 50% drop in cost than the cost of the initial random route.

By employing the opt2 search optimization (Figure 5) on a single driver, the following route is generated:

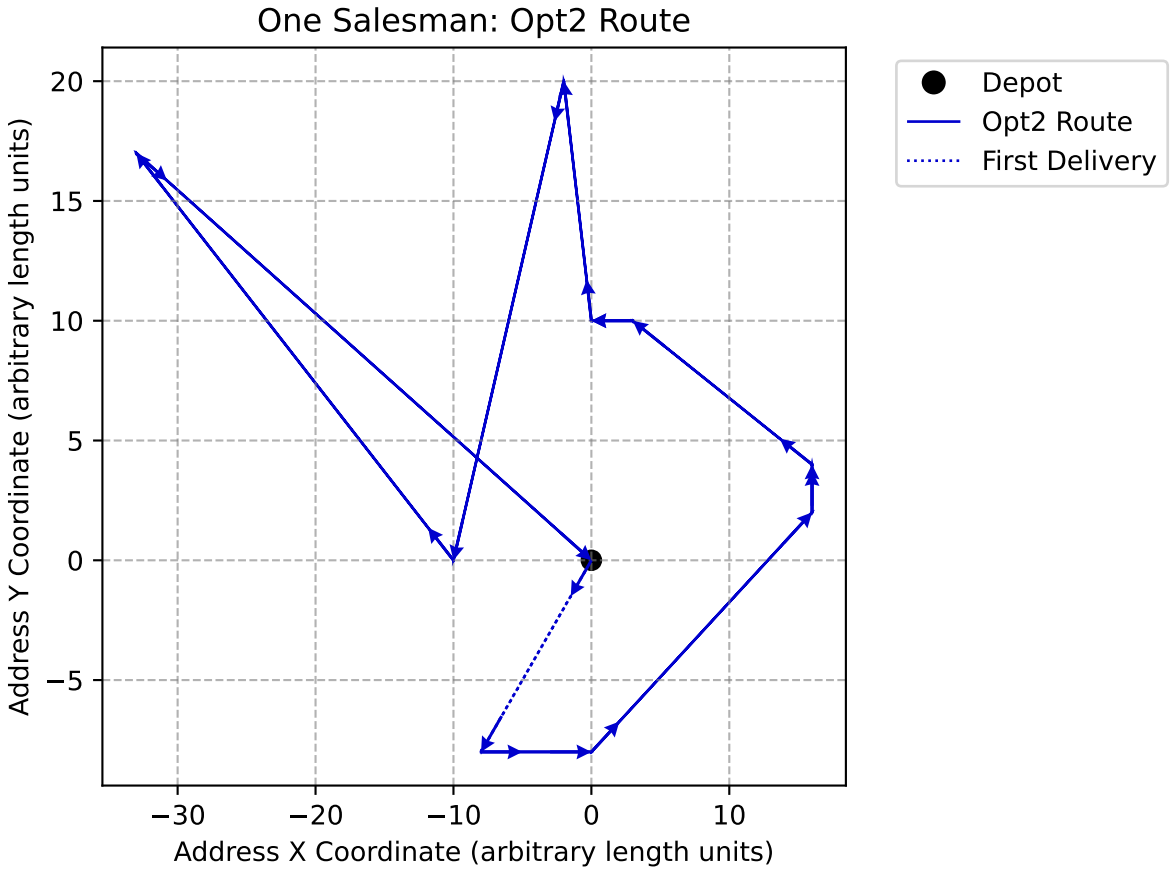

Figure 5: Opt2 Route

The opt2 route crosses itself once and has a cost of $720.038. Generally, in the test trials, opt2 leads to a more optimal solution than the greedy search, however, given that both are based on local optimality, neither route will necessarily be better than the other in terms of optimization performance. It is important to note that fewer crossings does not necessarily lead to the most optimized route.

By employing the greedy search optimization and then applying the opt2 heuristic (Figure 6) on a single driver, the following route is generated:

Figure 6: Greedy-Opt2 Route

The greedy-opt2 crosses itself twice and has a route cost of $645.221, the most optimal cost out of the three methods. The greedy-opt2 search leads to about a 60% reduction in the cost of the initial random route.

3.2 Double Salesmen

When optimizing the cost of addresses with two salesmen, the following plots are generated.

3.2.1 Runtime/Speed Performance

When the greedy, opt2, and greedy-opt2 search are done on two salesmen (Figure 7), the following runtime graph is generated:

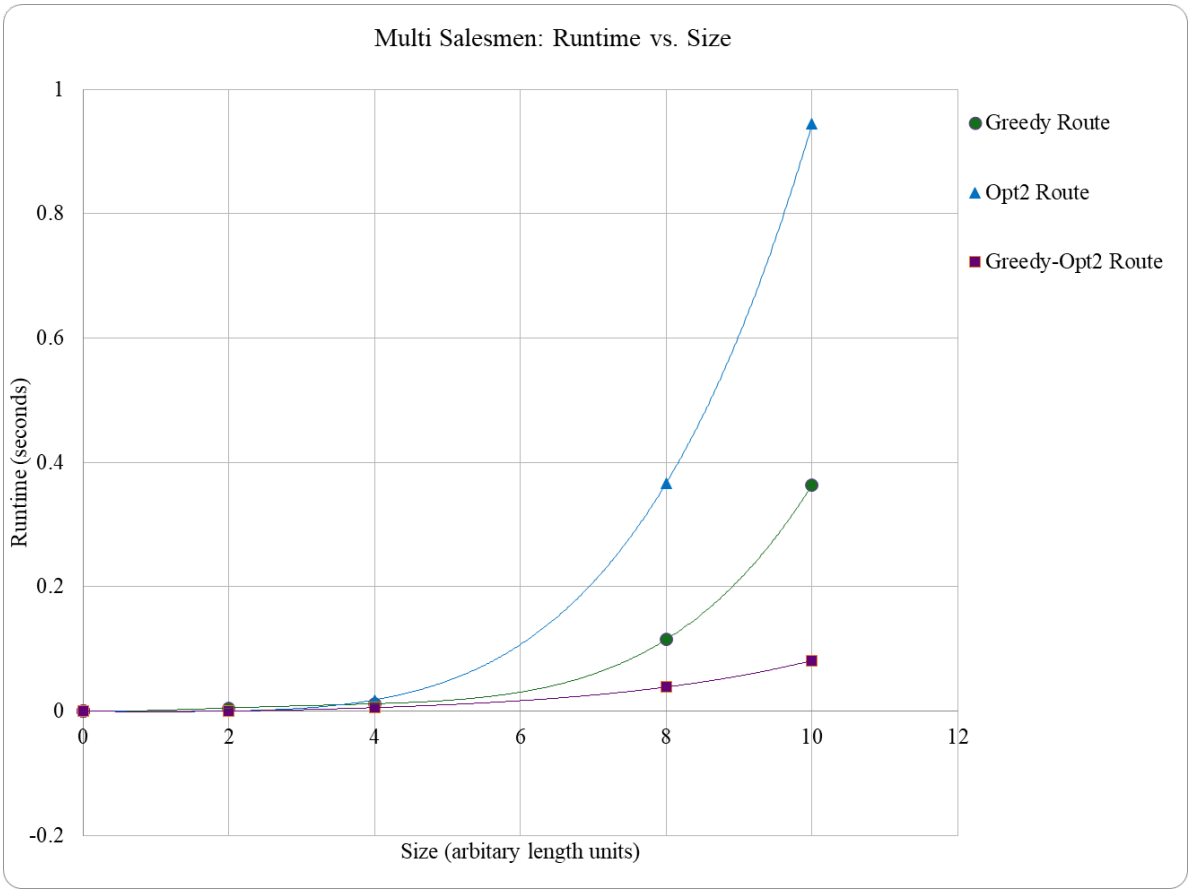

Figure 7: Runtime of Greedy, Opt2, and Greedy-Opt2 Search

Like with the single salesman, the runtime for all three search strategies seems to have an order of n^4 where n represents the number of addresses input to be optimized (size). Note that larger data sets were tested, but are not shown here due to scaling issues: the opt2 runtime grows much quicker than the other two search strategies (greeedy and greedy-opt have almost the same runtime or speed to convergence). As with the single salesman, given that a more comprehensive algorithm, which tries out all possible combinations, grows exponentially, a quartic equation, which is much smaller, aligns with our model; all three optimization techniques do not provide a comprehensive, absolute solution. Additionally, consider that the `multipath` function has four `for loops`, which could compound to create a quartic equation. It is important to note that due to the few data points in both the single salesman and the double salesmen optimization and the limited best-fit lines attempted (linear, polynomial, and exponential), no final/comprehensive conclusions about the growth rate can be made.

3.2.2 Accuracy/Improvement Performance

Data for an example route optimized through the greedy, opt2, and greedy-opt2 routes is presented below.

The initial route (Figure 8) is a set of random input addresses, with no order/optimization in place:

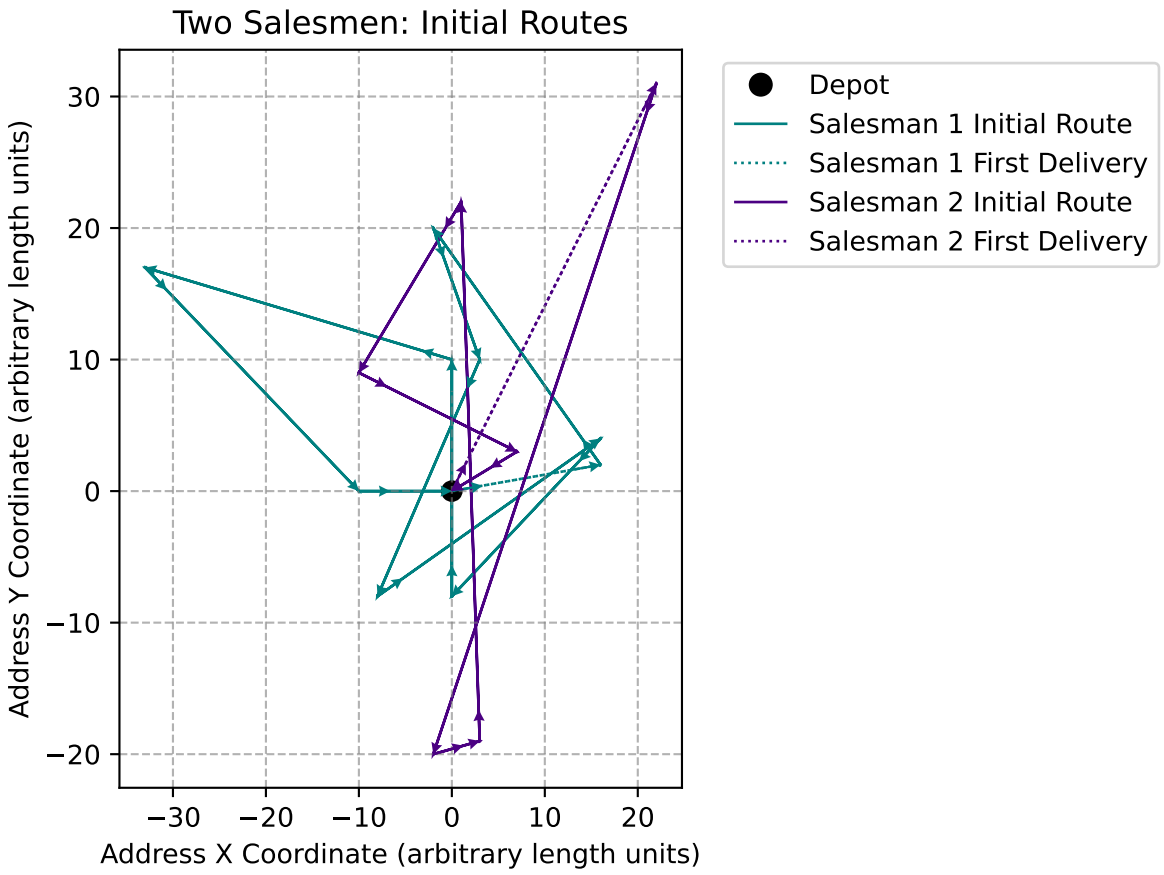

Figure 8: Initial Routes

The cost of the two salesmen for the entire random route is $2506.19.

Implementing the greedy search for multiple paths (including the swaps between salesmen) (Figure 9), results in the following plot:

Figure 9: Greedy Routes

The cost of the two salesmen for the entire greedy route is $1119.033, a more than 50% reduction in cost than the random routes.

Figure 10 shows the route after using the opt2 heuristic:

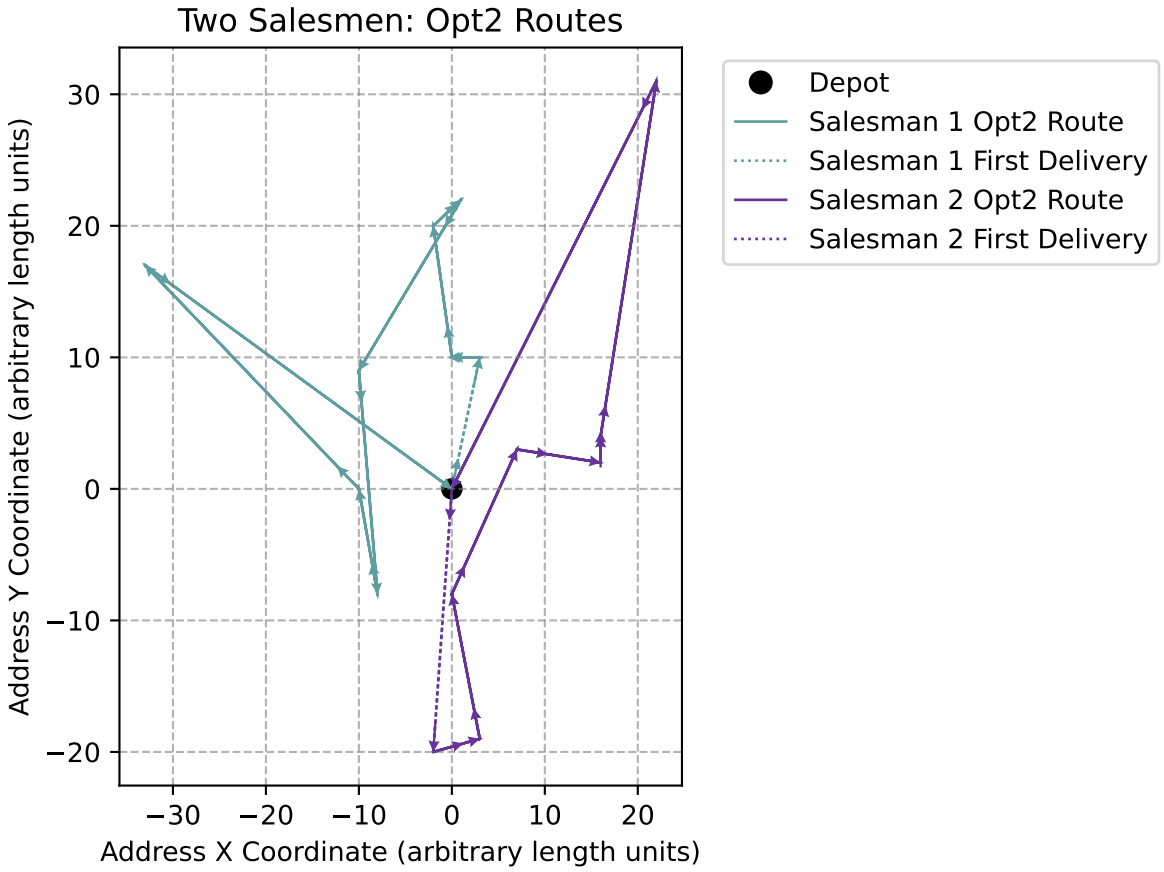

Figure 10: Opt2 Routes

The cost of the two salesmen for the entire opt2 route is $1099.97, which is an improvement from the greedy route's cost of $1119.033.

Figure 11 shows the route with the greedy-opt2 search:

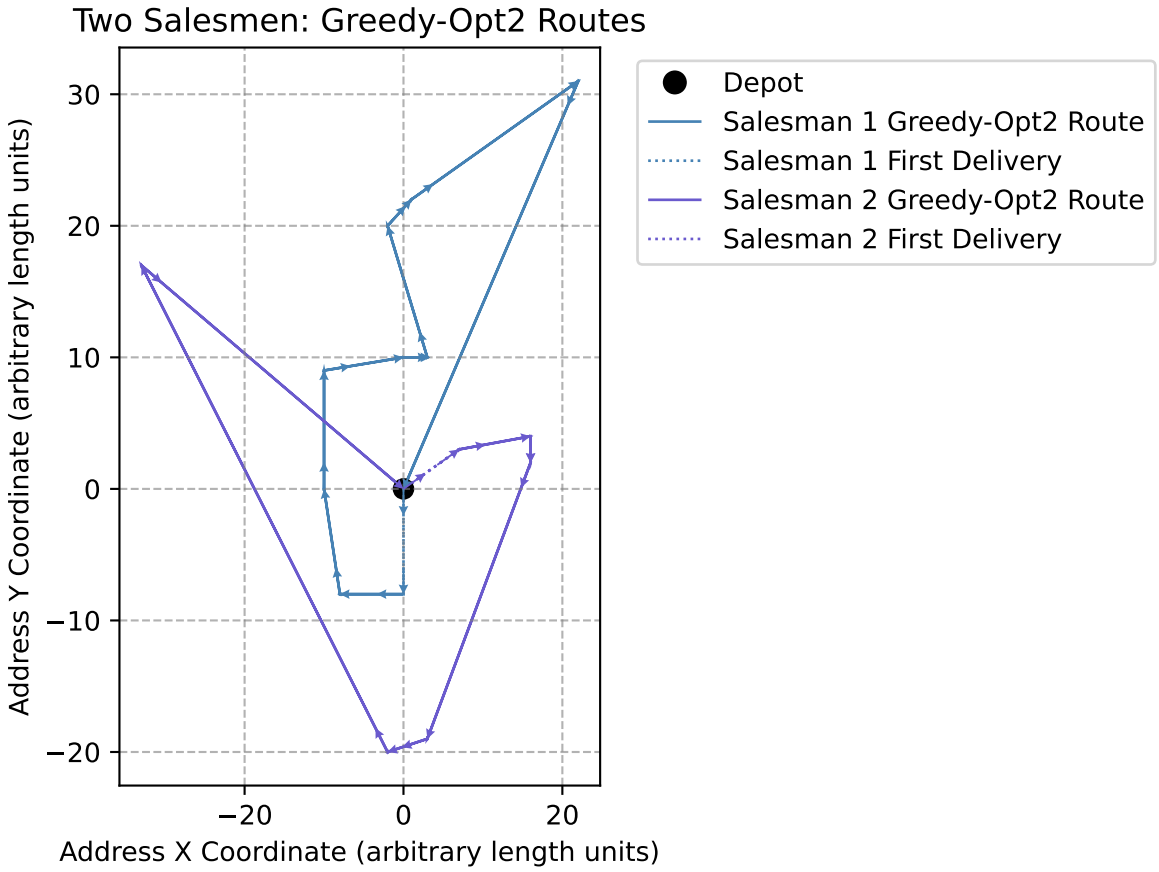

Figure 11: Greedy-Opt2 Routes

The cost of the two salesmen for the entire greedy-opt2 route is $972.19, which is the least cost out of the three optimizations. The greedy-opt2 search leads to about a 62% reduction in the cost of initial routes.

3.2.3 Dynamic Performance

As expected, an optimization formed for a dynamic set of addresses is not as good as an optimization formed on the same set addresses without dynamicism.

Table 5 compares the route length of a dynamic versus a non-dynamic model (both using greedy-opt2 search strategies):

Number of Addresses	Non-Dynamic Distance	Dynamic Distances	Random Addresses Addition to Delivery Rate
7	71.13	98.44	1:6
15	81.61	129.55	1:3
17	92.38	162.81	1:1

Table 5: Dynamic vs. Non-Dynamic Performance

4 Conclusion

Ultimately, each optimization strategy comes with its advantages and disadvantages. The `greedy_route` method focuses on optimizing each new address individually and while this strategy may be less computationally intensive, it definitely does not return the best possible route. On the other hand, the `opt2` method implements a heuristic process that reverses portions of the route, possibly leading to a less costly route than the greedy search. However, it only explores local improvements and not the entire space, so, it may also converge to an un-optimal route. Finally, the `greedy-opt2` route strategy combines the methods described above. This approach strikes a balance between efficiency and optimization, yet it still falls short of finding the most optimized route sometimes, since it also does not consider all possible routes.

For the single, non-dynamic salesman, all three methods can converge and provide reasonable improvement sufficiently quickly–for 1000 addresses, the max runtime is 60 seconds with the opt2 heuristic. As expected, the greedy-opt2 search converges most quickly and accurately. For two non-dynamic salesmen (Figure 7), however, only the greedy and greedy-opt2 can function reasonably; at only 20 addresses, while the greedy runtime is 5 seconds and the greedy-opt2 runtime is 4 seconds, the opt2 runtime is 920 seconds. Only the greedy and greedy-opt2 heuristics can withstand larger address data sets. Like for the single salesman, the greedy-opt2 search performs most quickly and optimally for the two salesmen.

4.1 Extensions

A major constraint for the current models of this project are their inability to handle large datasets of addresses quickly, especially for the dynamic multiple salesmen. A possible improvement would be to investigate and implement more advanced optimization algorithms beyond the greedy search or 2-opt. K-means, a clustering method in machine learning, could potentially divide sections of addresses into separate groups faster than the three investigated algorithms. Each salesmen could then visit one group of addresses.

Parallel processing could also be considered to speed up the route optimization methods. For example, for multiple salesmen, different processors could apply the greedy methods to each salesman; or, for non-dynamic addresses, the greedy method could be applied in the middle of the route, instead of starting at the depot, to generate a new, perhaps improved method.

Future extensions could also be made to account for more than two salesmen.

4.2 Ethics

This model suggests that drivers cannot return home until they finish all deliveries; this is a harsh, albeit exaggerated, reflection of reality, where drivers are pressured to complete all deliveries before returning to the depot and going home [3]. Per Table 5, for dynamic routes, delivery distance compounds quickly; for the same extra two addresses (rows 2 and 3), the dynamic distances have significantly larger increments than the non-dynamic addresses. The lonely delivery hours [3] quickly accumulate, especially with dynamic addresses and/or an unsatisfactory optimizing search strategy.

Works Cited

1. "Amazon now delivers more packages than FedEx, UPS." *TechSpot*, https://www.techspot.com/news/100970-amazon-now-delivers-more-packages-than-fedex-ups.html#:~:text=The%20Wall%20Street%20Journal%2C%20Amazon%20delivered%20more%20packages,FedEx%20in%202020%20and%20surpassed%20UPS%20in%202022.

2. "What it's really like to be an Amazon delivery driver." *Business Insider*, https://www.businessinsider.com/amazon-delivery-driver#:~:text=America's%20doorsteps%20would%20look%20different,the%20company%20they%20work%20for.

3. "How I Get By: A Week in the Life of an Amazon Delivery Driver." *Vice*, https://www.vice.com/en/article/bvgzwa/how-i-get-by-a-week-in-the-life-of-an-amazon-delivery-driver.

4. Eijkhout, Victor. *Introduction to High Performance Scientific Computing*.

5. "Travelling Salesman Problem." *Wikipedia*, https://en.wikipedia.org/wiki/Travelling_salesman_problem.