Amazon Delivery Truck Simulation

Hanna Butt [*] Ashton Cole [†] Kelechi Emeruwa [‡]

December 4, 2022

Abstract

Summary of whole paper. Note that this is not an introduction or context, but a summary.

1 Introduction

Imagine that a delivery company has a series of orders that it needs to fulfill. It has a truck that can stop at each address and make the delivery. The company naturally wants to save on time and fuel costs, so it tries to find the shortest path from its warehouse to cover all of the stops. This is the basic premise of the Traveling Salesman problem. Our team chose to solve this for our final project in COE 322: Scientific Computation at the University of Texas at Austin for the fall of 2022.

In Section 2 we discuss the algorithm that we used to solve the simple Traveling Salesman Problem, and expansions upon it to construct our final program. In Section 3, we display the outcomes of several test scenarios and discuss the results. Finally, in Section 4, we offer our final thoughts and reflect on ethical considerations. However, we will first further define the problem for our purposes and apply some limitations and assumptions in the following subsections.

1.1 Perfect is the Enemy of Good

One could simply test each of the $n!$ combinations of a list of n addresses to fund the one with the shortest total distance, but with 4 stops this becomes quite tedious, and after that nearly untenable. A computer could solve this faster, but the problem still becomes very computationally expensive at a factorial rate. The preferable alternative is to use algorithms to find good and better paths, significantly cutting down on calculations. Perhaps it will neglect the perfect solution, but in a world with finite resources, we have to make the trade-off to settle for less. The algorithms which we employ, detailed in Section 2 reflect this.

[*] HFB352, hannaf2020@gmail.com
[†] AVC687, ashtonc24@utexas.edu
[‡] KEE688, kelechi@utexas.edu

1.2 Earth is Flat

We also have to restrict the definitions of an address and the distance between addresses. Since this project is intended to focus on the algorithms one would use to solve the Traveling Salesman problem, as opposed to practical considerations for constructing a turn-key solution for use in the real world, addresses are represented with Cartesian coordinates on a two-dimensional Euclidian plane. Distances are assumed to be Euclidian, i.e. "as the bird flies," and travel times are assumed to be proportional thereto. The distance could alternatively be defined as the "Manhattan Distance," i.e. the distance in a perfect rectilinear street grid. Using real-world geographic data considering road layouts, speed limits, real-time traffic data, and Earth's curvature would be more realistic, but it would also add significant complexity to our implementation while contributing little to the core principles of the solution.

1.3 TODO

MORE DEFINITIONS HERE: deliver by date, multiple days, multiple trucks

2 Methodology

To approach this problem, we wrote a library and scripts in C++. These were compiled with GNU's `g++` on our local machines and Intel's `icpc` on the Texas Advanced Computing Center's ISP supercomputer. These scripts are outlined below.

- `traveling_salesman.h`: a header file for our TravelingSalesman library, defining all of the objects and algorithms used in the project
- `traveling_salesman.cpp`: an implementation file
- `tester.cpp`: a script which tests the functionality of our TravelingSalesman library and generates TikZ code for figures displayed in this report; this script is compiled as `tester.exe`
- `deliveries_generator.cpp`: a script which generates `.dat` files containing lists of orders to be processed by `delivery_truck_simulation.exe`; this script is compiled as `deliveries_gen.exe`
- `delivery_truck_simulation.cpp`: a script which represents a hypothetical final product for use in industry, as described in Section 2.4; this script is compiled as `delivery_truck_simulation.exe`

We began our project by writing the header and implementation files, coupled with tests in our tester file. After we confirmed that all of our objects and algorithms functioned as expected, we designed a main program to best parallel a real world application of solving the Traveling Salesman Problem. The structures and algorithms that we developed are further detailed in this section.

In Section 2.1, we outline the structure of the classes that we used to represent and solve the problem. In Section 2.2, we describe the algorithms we used to solve the simple Traveling Salesman Problem. In Section 2.3, we describe the expansion of the problem to account for optimizing multiple delivery routes. Finally, in Section 2.4, we describe how we combined our algorithms into a final product for a hypothetical user.

2.1 Object-Oriented Structure

Our scripts took advantage of C++'s object-oriented capabilities to organize the problem. This section provides a brief overview; the contents of these classes are not described exhaustively.

Each delivery stop is represented by an `Address` object, which has two-dimensional integer Cartesian coordinates `i` and `j` representing the location of the address, an an integer `deliver_by` which describes the day by which the order is supposed to be delivered, or the stop passed-by. The class can also calculate the distance to other `Address`es, using either the Euclidian distance $\sqrt{i^2 + j^2}$ or Manhattan distance $|i| + |j|$. In our implementation, we use the Euclidian distance, but it could easily be replaced with another formula.

A list of `Address`es is represented by an `AddressList` object, which holds the objects in a `std::vector<Address>` instance variable called `address_list`. This class can add, remove, and rearrange `Address`es. It does not accept duplicate `Address`es, i.e. those with the same coordinates. If the user attempts to add an order to the same `Address` with different `deliver_by` due dates, then the lesser value is accepted. This parallels orders being combined in real life. Note that the preference for the earlier date is based on the assumption that at the time that the `Address`es are added to the `AddressList`, they are available to be delivered.

The `Route` class extends the `AddressList` class by including a `hub` instance of type `Address`. This represents the starting and ending point of the `Route`. This class contains several functions to solve variants of the Traveling Salesman problem.

2.2 Traveling Salesman Problem

Having developed a strong Object-Oriented skeleton we can explore algorithms to address the Traveling Salesman Problem. An intuitive approach we could adopt is called the *greedy algorithm* also known as the *nearest neighbor algorithm*. The greedy algorithm works to develop an optimal route by traversing (from a starting point) to the next closest point in a list of points until all points in the list have been visited. To achieve this optimal route, the greedy algorithm must determine which point is closest to the current point at each iteration. This can be accomplished with the help of our *index_closest_to()* method. Now we can iterate through the list of addresses and at each iteration calculate the next closest address until we have visited all the addresses in our list. To prevent visiting the same address more than once, we can make another list, and pop elements from our current list into our new (optimized) list. Since we call our *index_closest_to()* method n times (where n is the length of our address list) and the method itself has a time complexity of O(n) we arrive at a Big-O time complexity of O(n^2) for this greedy algorithm . The figure below illustrates the effect our greedy algorithm has on developing a more optimized route. 8. The

Figure 1: insert improvement from greedy alog

improvements from the greedy algorithm seem to suggest that it will play an important role in developing our Amazon route scheduling algorithm. On the other hand, additional testing demonstrates how the greedy algorithm alone can fail to provide the most accurate solutions:

Figure 2: insert deficiency of greedy algorithm

Although this deficiency may seem minimal with a few routes, at scale , the costs incurred due to longer distances traveled and longer delivery times may prove to be a great burden for Amazon, therefore we'd like to explore better methods. One approach we can adopt is based on the opt-2 heuristic. The opt-2 heuristic suggests that optimal routes are generally not "entangled" (i.e no intersections). Therefore, if we can work to "detangle" a given list of points we can find its optimal path. Both pathes can be seen in the figure below.

Figure 3: ** insert entangled vs detangled route and its total distance.

Rather than aim to algorithmically identify the intersections in a given path, we can try to "detangle" a path by reversing segments of it and checking to see if such a modification generates a more optimal path. The figure below shows how this is done visually.

To implement this in code we employ the following strategy:

```
For all possible segments in our path: Make the new path:
Original start + reversed segment + original end If new path is shorter
keep it
```

In code this equates to:

```
Route Route::opt2(){
    AddressList address_list(address_vec);
    double current_length = address_list.length();
    for (int m=1 ; m<address_list.size(); m++){
        for (int n=0; n <= m; n++){
            AddressList new_list(address_list.reverse(
    n, m+1));
            if ( new_list.length() < address_list.
    length() ){
```

Figure 4: ** insert single pass of reversage of a path

```
                address_list = new_list;
                current_length = new_list.length();
            }
        }
    }
    Route new_route(address_list, hub);
    return new_route;
}
```

In this implementation, we utilized the *reverse()* method from the C++ standard library reverses a given segment of a vector in place. Below we compare the results from our greedy algorithm with that of our opt-2 algorithm:

Figure 5: compare greedy w/opt 2 (use previous greedy deficiency example too)

The opt-2 algorithm seems to be a reasonable replacement for the greedy algorithm. However, we should also consider this algorithm's efficiency. Calling the *reverse()* and *length()* methods of an `AddressList` of size n leads to an average time complexity of O(n) for each. Moreover, because this opt-2 algorithm evaluates a total of $\frac{n^2}{2}$ combinations.[1] the overall time complexity of this algorithm is O(2n^3) which simplifies to O(n^3). Fortunately the efficiency of this algorithm can be slightly improved. Instead of reversing segments of our route we can choose to swap pairs of Addresses instead Additionally, rather than call *length()* to compare the *total distances* between the modified and original path, we can focus our comparisons on just the *change in distance* caused by each swap. In code, this results in the following modified opt2 algorithm:

Listing 1: Opt-2 Algorithm (optimized)

```
Route Route::opt2(){
    // this code needs to be modified
    AddressList address_list(address_vec);
    double current_length = address_list.length();
    for (int m=1 ; m<address_list.size(); m++){
        for (int n=0; n <= m; n++){
            AddressList new_list(address_list.reverse(
   n, m+1));
            if ( new_list.length() < address_list.
   length() ){
                address_list = new_list;
                current_length = new_list.length();
```

[1]The estimation comes from the fact that the total number of combinations is equivalent to the sum of a triangular number sequence

```
                }
            }
        }
        Route new_route(address_list, hub);
        return new_route;
}
```

Consequently, the time complexity is now reduced to O(n^2) (same as our greedy algorithm!) and yields the same results as before:

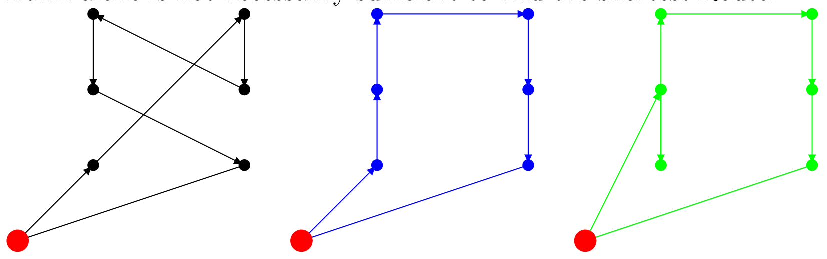
Figure 6: ** make sure it yields same results, then copy paste from previous figures **

Thus, our opt-2 algorithm seems to be a substantial improvement from the greedy algorithm. Yet, even this approach seems to not be heavily reliable. See below:

Figure 7: ** include optimized opt2 yielding poor solution results

Thus, to develop more optimal solutions we could choose to employ our greedy algorithm followed by opt-2. In addition, because both algorithms have a time complexity of O(n^2) the overall time complexity of running both algorithms would be O($2n^2$) which simplifies back to O(n^2). Although this approach is well-suited for single delivery routes, we can extend the opt-heuristic to optimize two routes simultaneously as well. In other words, we can further shorten the paths of two routes by swapping segments of one out for segments of the other. This may be more desirable, as it introduces more flexibility, and thus could lead to identifying potentially shorter paths.

Figure 8: An unsorted Route is optimized through both the greedy algorithm (blue) and the opt2 algorithm (green). This demonstrates how the opt2 algorithm alone is not necessarily sufficient to find the shortest Route.



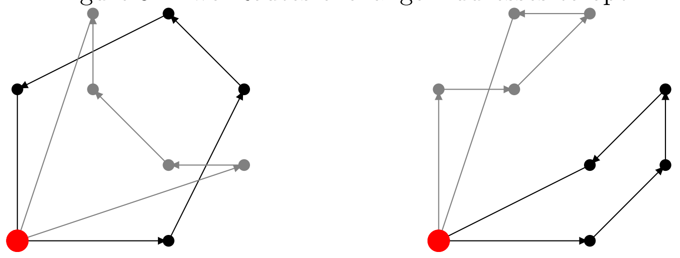
2.3 Multiple Traveling Salesmen Problem

Optimizing multiple routes simultaneously extends our Traveling Salesman Problem to the Multiple Travelling Salesman Problem (MTSP). Seeing how computationally expensive our initial opt2 algorithm was in section 2.2 , one can imagine how much greater this costs would become for an algorithm that optimized multiple routes simeltaneously. For this reason, and others we will, aim to optimize only two routes at once. As mentioned in section 2.2 we can achieve this by extending the opt-2 heuristic to multiple routes in the following manner:

```
For all possible segments for both routes swap both routes with
the following variations: Reverse route 1 then swap Reverse route 2 then swap Just swap
 Reverse both routes then swap If the distance from any of the variations is improved,
keep the modification.
```

Talk about how swap algorithm works. Figure 9 provides a simple example of how this algorithm improves the total distance. (Now prove it with data! What are the distances before and after?) ALSO: Talk about how the gray and black Addresses are symmetrical, but the trucks take different Routes. Is this okay, or does one or both of them need to go through greedy/opt2???

Figure 9: Two Routes exchange Addresses to optimize their distances.



2.4 Developing the Final Product

After our team implemented our solutions to the Single and Multiple Traveling Salesman Problems, we decided to explore dynamicism by constructing a simplified route allocator for a delivery company. Every morning, a regional fulfilment center recives a list of orders to fulfill, corresponding to packages available onsite. It invokes our program, which combines these with unfulfilled orders from the previous day, and delegates them amongst a predetermined number of trucks. The `Routes` are then optimized individually and between one another. At this point, their distances are measured. If a `Route` exceeds a predetermined distance limit, `Addresses` are removed from the `Route` based on their `deliver_by` due date, until it falls within an acceptable length. The delivery routes are then exported to documents for the drivers, the unfulfilled orders are saved to a file which overwrites the old one, and performance statistics are compiled into a report for management.

All of the tasks to be completed before the start of a business day are modularized in a single function. It requires parameters specifying input and output file locations, the number of trucks available, the maximum permissible route distance, the hub address, and a boolean which allows the user to specify if data should be output from intermediate optimization steps. Most of the harder tasks of the simulation have already been solved in our library. This even includes the repetitive tasks of reading or exporting a `Route` or `AddressList` from or to a file, based on a given file path string. To demonstrate this program, we constructed a simulation with pre-generated daily orders spanning two weeks. The results of this simulation are discussed in Section 3.

3 Results

Pretty pictures and tables go here. Describe each situation being displayed and talk about what they mean, e.g. is it the optimal solution? Good enough? Is there a tradeoff between time to execute and quality of results?

Hmm, maybe insert a table comparing number of nodes/trucks to program execution time. What rate does it increase at $(O(n), O(n^2), \&c.)$

3.1 Scenario 1

3.2 Scenario 2

3.3 Performance Data

go over execution time and stuff

4 Conclusion

Talk about what we learned, how this all applies to industry, ideas to scale the problem up, ethics, &c.

Talk about how our simulation is flawed, e.g. address removal isn't intelligent (purely by due date); orders aren't held back to be grouped with later-arriving orders; after stops are removed, the routes arent re optimized; main simulation program isn't super flexible (e.g. to change document names, formattig, you have to go into the code; dat files have to be in a specific format, can't be a database); no UI for ease of use

A Sample of Using Listings Package

Please remove this appendix before publishing! Here's some pretty C++ code from Listing 2.

Listing 2: Example Code

```cpp
#include <iostream>
using sdt::cout;

int main() {
    // Print out hello
    cout << "Hello world!" << '\n';
    return 0;
}
```