Alec Flanigan

Professor Xia

CS150: Data Structures and Algorithms

7 May 2017

Project 3 Report

Introduction

The program was designed to find an efficient way to deliver supplies to shops scattered across a city given there are several warehouses in locations across the same city. Given that there is currently no known way to compute and prove the *best* solution in polynomial time, the answers computed may not be the most optimal solution. While there is no way to prove that a calculated solution is the best solution, the program can compute a solution somewhere in between the worst and the best.

Approach

Beginning the approach of the problem, it was decided that a "greedy" algorithm would be used. A greedy algorithm is a style of algorithm that prioritizes the current best solution—not the best solution over time. For example, the algorithm favors the next closest shop from the current position being compared to it. This way, the algorithm is at least doing better than the worst case scenario, as it is always (or at least usually) making a choice that is best in the current situation. In order to understand the details of the approach, the separate pieces of the program must first be introduced.

To begin the structure of the program, it was known that the layout of the city would be represented by a graph (a set of vertices and edges). The idea of a graph was chosen as it would represent a set of points and paths in between them. The graph would store every shop and warehouse in a list along with the the "edges" between them—both in the form of an array list (Oracle ArrayList). These edges represent the path from one vertex to another, and the distance of the edge is its weight. That being said, each edge stores its beginning vertex, its end vertex, and the distance between the two vertices. These edges only exist from each warehouse to each shop, and each shop to each warehouse, and they are stored in sorted order by using Java's Collections binary search (Oracle Collections). Shops do not have edges back to any warehouse.

A single cargo is representative of one set of supplies that a shop may need. It stores the shop that it needs to be delivered to, the weight of the supplies, and a boolean to track whether or not it has been added to a truck already.

Talking about all of the shops and warehouses as vertices calls for an explanation of its own. There is a single vertex class that has an id, a point of location (coordinate represented by Java's Point class), and a list of its outgoing edges (Oracle Point). Now moving onto the shops. The shop class extends the vertex class—giving it the same attributes—and adds a few other functionalities. Each shop stores a list of supplies needed for that day (represented by the cargo class). This list is represented by an array list because each shop has an unknown length of needed supplies. Each one also has a boolean to keep track of whether or not that shop has had all of its needed supplies accounted for and set for delivery.

While the warehouse class also extends the vertex class, it is a little more complicated. Each warehouse is initialized with its id, location, and a max allowance of trucks from the warehouse. This means that if a warehouse has a max truck allowance of five trucks, the warehouse can send five separate trucks from itself to make deliveries in a single day; however, a warehouse does not *need* to use all of its trucks. Since we know the amount of allowed trucks from each warehouse, these trucks are stored in an array initialized to the size of allowed trucks. The class also has a *MAX_DISTANCE* variable in charge of setting an upper bound of the max distance each truck is allowed to go from its starting warehouse. This distance is computed before the simulation by calculating the average distance of the closest warehouse to all shops and multiplying it by two. This ensures that trucks don't travel too far from their starting place when they are scheduling their deliveries since they eventually have to make a return trip which can be costly if they are too far. Multiplying by two also ensures that there is some leniency in the distance so it is not thrown off by an outlier. The warehouse class is also responsible for planning the delivery routes of each truck it has available if needed. It should also be noted that the last warehouse—also referred to as the base warehouse—is treated as a special warehouse for its high volume of trucks. It resides in the location of (1, 1) and is used separately to fulfill the remaining supplies needed by each shop.

As mentioned, each warehouse has an allowance of trucks. These trucks store their starting warehouse, an array list of cargoes, a hash set of shops that the truck has checked, the distance it has traveled, and a weight *CUT_OFF*. Staring from the beginning, the list of cargoes represents each set of supplies that the truck is set to deliver. This is stored in an array list because each truck needs to be flexible, as it is unsure of how many cargoes it will need to store. Next, a hash set of checked shops is used to keep track of which shops the truck has already checked (Oracle HashSet). Checked means that the truck has iterated through all of the supplies that the shop still needs and can no longer transport anything else that the shop may need. It was decided to use a hash set for instant confirmation of whether a shop has been checked or not. The *CUT_OFF* is designed so that once a truck has reached a certain weight, it doesn't waste time checking all shops for an insurmountable weight. Each truck keeps track of the distance it has traveled as the total distance of all trucks is computed at the end to gauge the efficiency of the program.

The scheduler is the last crucial piece of the program, as it is responsible for operating all of the classes together. The scheduler uses a file with the necessary information for the shops and one for the warehouses. Once given the two files, the scheduler initializes all components and performs the greedy algorithm mentioned before. In more detail, the algorithm iterates through every warehouse and plans the delivery route of its trucks. The planning of this route starts at a single warehouse and a new truck. The closest unfulfilled shop to the warehouse is searched for and if the truck has enough unused weight, it adds the shop to its route, adds as much of its needed supplies as it can, and then searches for the closest unfulfilled shop from the last shop added to its route. The shops are searched for by the closest to the last position in the route of the truck, but it must also be within the *MAX_DISTANCE* specified earlier. The process repeats until the truck reaches its *CUT_OFF* weight or until there are no more shops within the *MAX_DISTANCE*. The warehouse repeats the process until it has no more trucks available or until there are no more unfulfilled shops within its range. This is repeated for each warehouse until the last warehouse, called the base warehouse. The base warehouse then performs the same style of routing trucks, ignoring the *MAX_DISTANCE* parameter, until all shops have been fulfilled.

Methods

Because the optimal solution for any data set cannot be easily found, the testing of the program presented a difficulty. The solution to this problem was to create a few test cases where the best solution could be easily calculated by hand. By doing this the best and worst solution was calculated by hand, then the program was run with the same parameters, and then the results could be compared. To ensure a coverage of possibilities, test cases were catered to unique situations. To calculate the worst case distance for each case, the farthest warehouse was chosen to fulfill each shop, and a truck would deliver a single set of supplies. For example if a shop needed two sets of supplies, it would need two trucks to come from the farthest warehouse to fulfill its needs. By doing this, the worst distance can be computed. The best case solution is still a little more difficult to figure out, but was done by hand. Because the worst case numbers are so much larger than the actual and best results, they will not be included in the graphs to show a closer comparison between the actual and best.

The first test case was to randomly piece together a list of shops with random locations and a random list of supplies along with a randomly located warehouse and the base warehouse (testshops1 and testwarehouses1). The second test case used the same set of randomly distributed shops, but placed a warehouse two units from each shop (testshops1 and testwarehouses2). The third case contains a close cluster of five shops (testshops2 and testwarehouses1). The fourth and final case uses a group of shops close to one warehouse, but with more supplies than the closest warehouse can supply (testshops1 and testwarehouses3).

Data and Analysis

Beginning with test case one, it is the best case to judge how well the greedy algorithm that was implemented would really perform on a real world case. Looking at illustration 1 we see that the actual results from the experiment were higher than the best result as expected; however, when comparing to the worst case of 2064 distance, the actual results seem very promising. The actual result was only behind by 14 distance when actually looking at the numbers, as the chart can seem a bit deceiving. In this case, the results are about 3.65% worse than the best results.

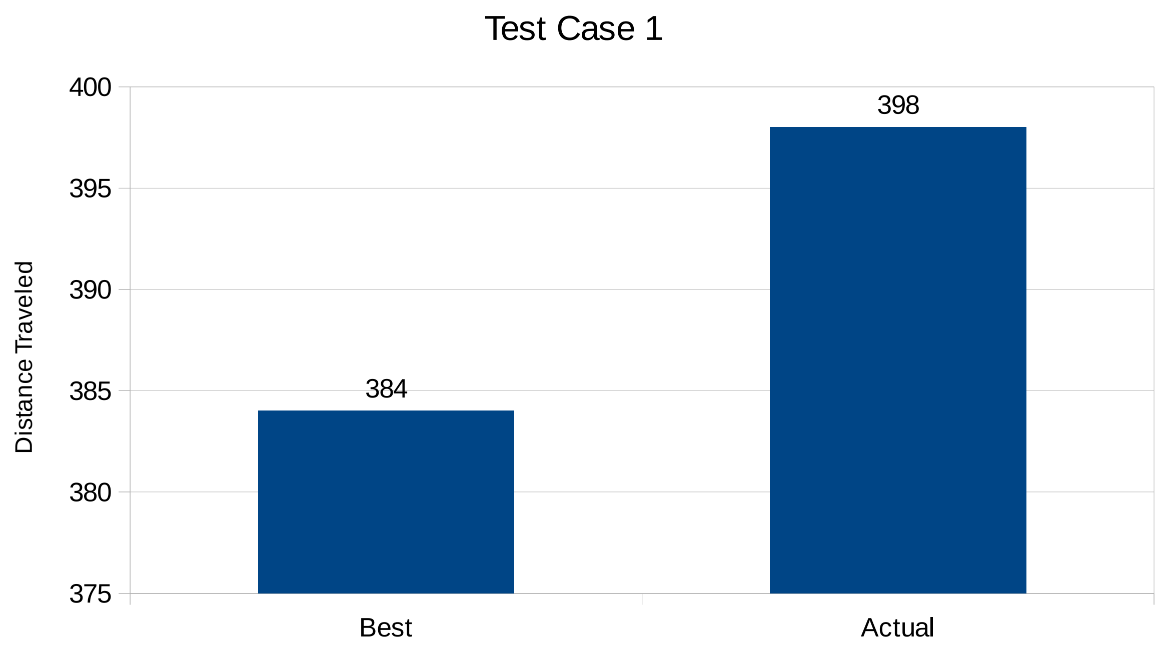

Illustration 1: Test Case 1: Using testshops1.txt and testwarehouses1.txt as parameters. These cases were the most randomly generated compared to later test cases.

As mentioned before, test case two uses a set of warehouses that are right next to each shop. This test case should be able to show if the algorithm is picking the closest shop in most cases. By looking at illustration 2 we can see that the algorithm is indeed picking the closest shops well. The actual results are equal to the best case. Not surprisingly, the results are nowhere near the worst case of 2184 distance.

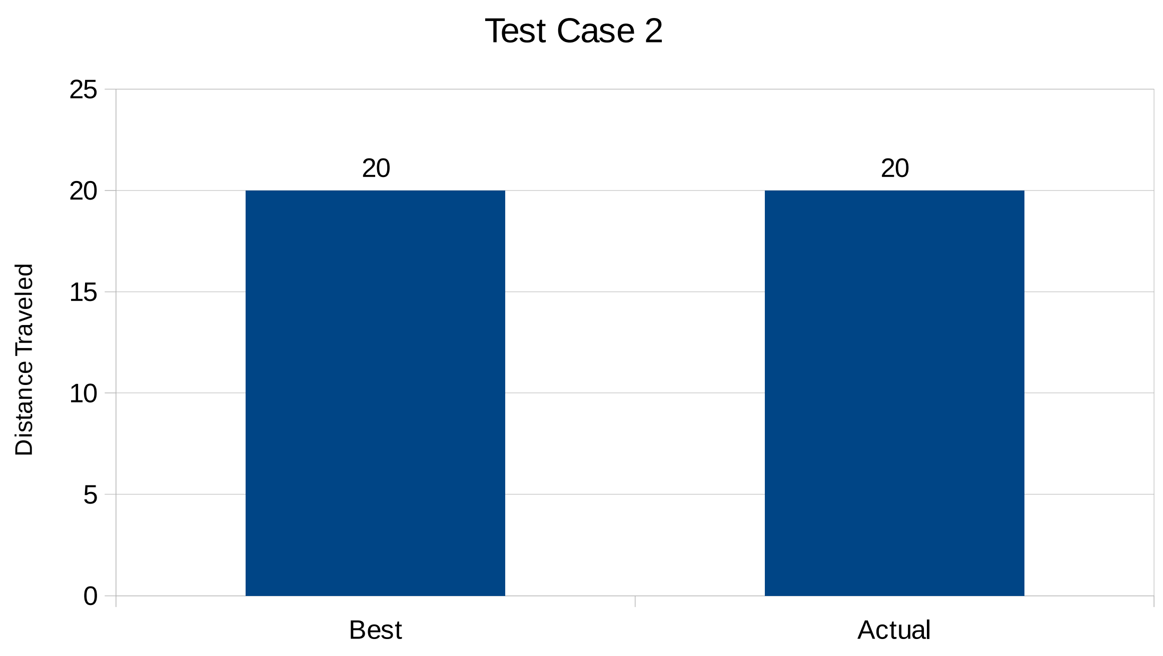

Illustration 2: Test Case 2: Using testshops1.txt and testwarehouses2.txt as parameters. This case has warehouses 2 units from each shop in the file.

Test case three uses a cluster of shops that, together, need a total of 872 supplies. The best case here needs two trucks at the least, so it should be able to show how well the algorithm handles routing multiple trucks efficiently. The actual result gave a distance of 276 compared to the best distance of 272 (seen in illustration 3). This result is still exceptional—especially compared to the worst result of 1584. The actual result is only 1.47% worse than the best case.

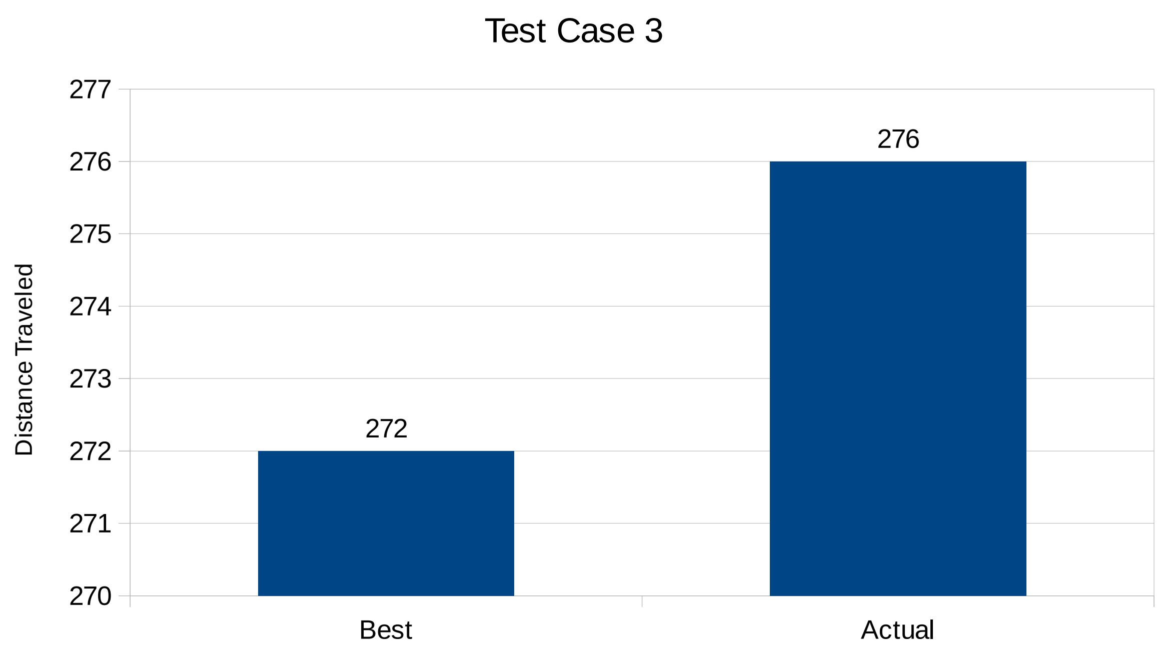

Illustration 3: Test Case 3: Using testshops2.txt and testwarehouses1.txt as parameters. This case consists of a cluster of shops of the middle of the city graph.

The final test case was designed to see how well the algorithm routed a limited number of trucks from a warehouse to its nearby shops. Looking at illustration 4, the actual result was only 8 distance behind that of the best case. Along with these results, the calculated worst case was 1584 again. This result is 4.88% worse than the best possible result. This result is still a very efficient one compared to the best.

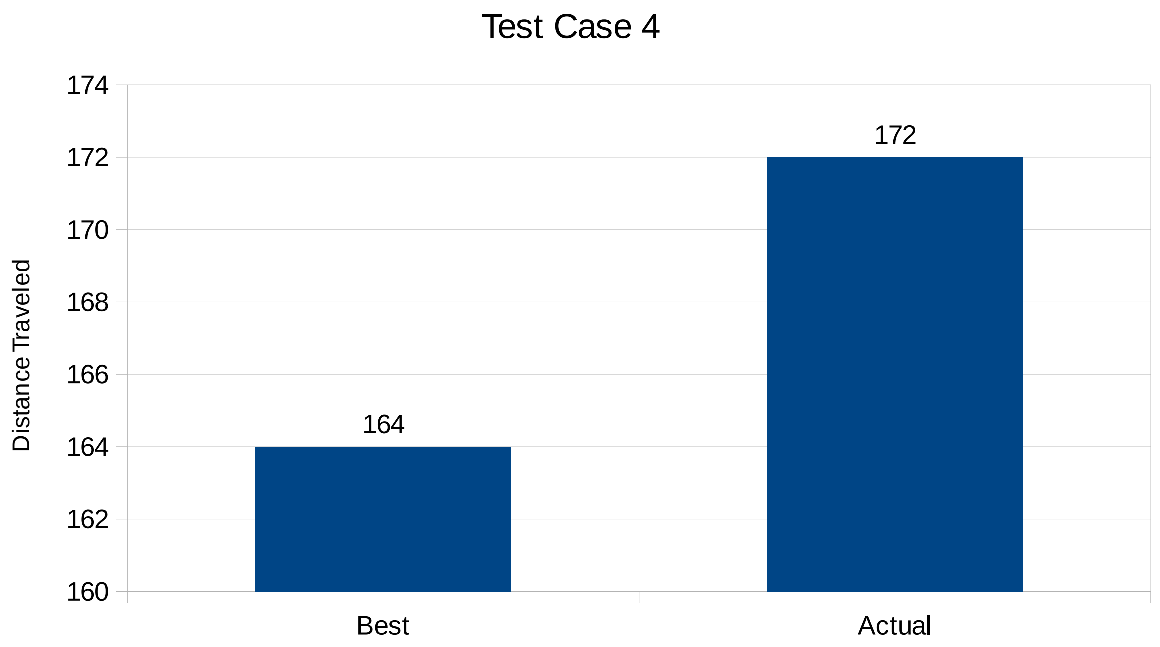

Illustration 4: Test Case 4: Using testshops2.txt and testwarehouses3.txt as parameters. This case tests the use of multiple warehouses efficiently.

Conclusion

While there are only a limited number of test cases performed, they prove that the algorithm and program as a whole is capable of producing desirable results. It is accurate for that fact the it is always between the worst and best case distances. As a bonus, the algorithm is often closer to the optimal result than the worst result. By manually changing a few of the parameters like the *MAX_DISTANCE* or the *CUT_OFF*, the program is able to calculate more optimal results in some cases, but this requires extra time, and a lot of guesswork. Given that optimal solutions are much more difficult to calculate (or even impossible) even in small cases, the test cases used were very limited. Although this is not the best way to compare actual results of real-world cases to what may be the optimal solution, it is a concrete way to show that the algorithm is reliable enough to produce results that are at least quite a bit better than the worst results. By adjusting the current implementation of the algorithm, it may be possible to produce better results than the ones found in this program.

References

Oracle. “ArrayList” https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html

---. “Collections” https://docs.oracle.com/javase/8/docs/api/java/util/Collections.html

---. “HashSet” https://docs.oracle.com/javase/8/docs/api/java/util/HashSet.html

---. “Point” https://docs.oracle.com/javase/8/docs/api/java/awt/Point.html