Elene Karangozishvili and Trang Le

Professor Xia

CS 150-02: Data Structures and Algorithms

December 07, 2018

Logistics Management Project Report

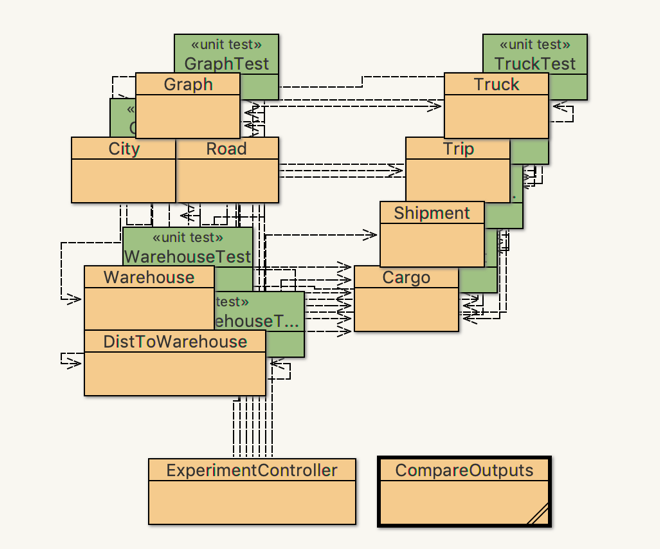
**Introduction**

The premise of the project is implementation of schedule optimization for a coffee shop delivery based on given routes and allowed capacity. The goal is to minimize the total distance traveled between the cities by all trucks, excluding the distance traveled within each city (Xia, 1).

I hypothesized that

**Approach**

A program was designed to read given input files, simulate the logistics of the delivery process, and write an output file of the results.



*Figure 1*. Class diagram of the program

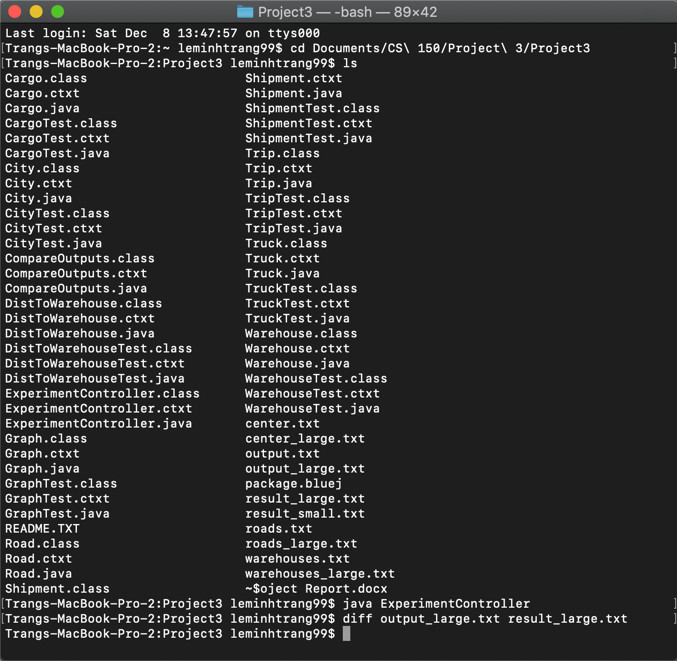
Object-oriented programming was the principle behind the design of the project. For simulation of the cities and roads, we implemented undirected graph (Xia) as the data structure, where cities were vertices and roads were edges. The adaptation of class Vertex to class City was modified to include a variable linking to a warehouse. Classes Cargo, Shipment, Trip, Truck, and Warehouse were designed as simulations of objects involved in the delivery process: class Cargo kept track of the weight and ID of each cargo; class Shipment had an ArrayList<Cargo> (*ArrayList (Java Platform SE 8 )*) of all the cargos delivered to the same warehouse within one trip; class Trip had an ArrayList<Shipment> of all shipments in the same trip; class Truck had an ArrayList<Trip>, the location, and the maximum capacity; class Warehouse contained a link to a city, a PriorityQueue<Cargo> (*PriorityQueue (Java Platform SE 7 )*), since cargos are loaded onto trucks in ascending order of weight, and ArrayList<DistToWarehouse>, which kept track of the shortest distances to other warehouses.

Lastly, class ExperimentController class read given input files, created a graph from given input, ran Dijkstra algorithm (Xia) to find the shortest paths, processed the simulation, and wrote the results to an output file.

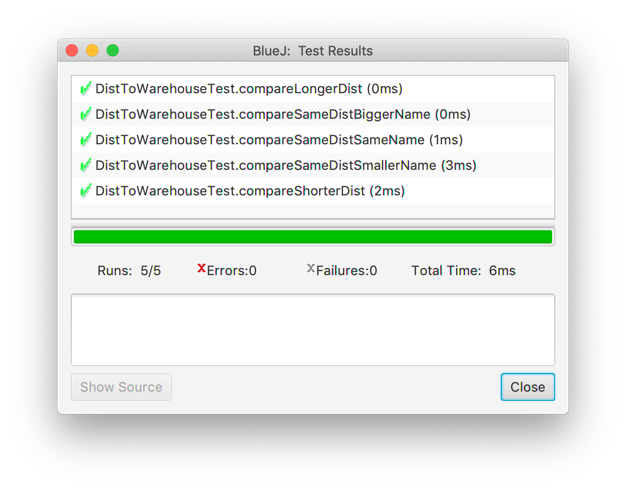
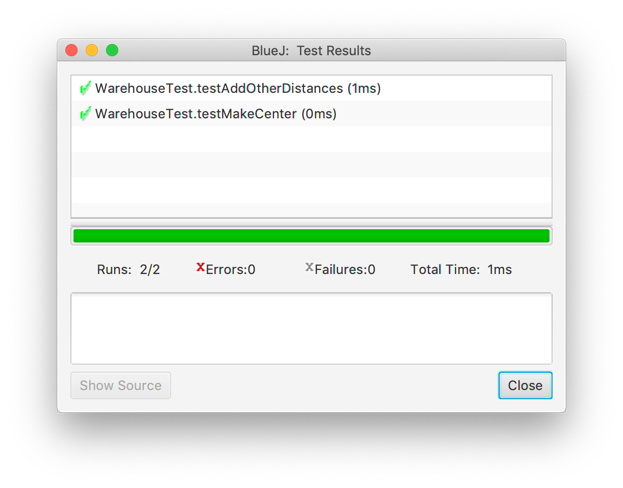
**Methods**

The program was run on two given sets of data, one small and one large.

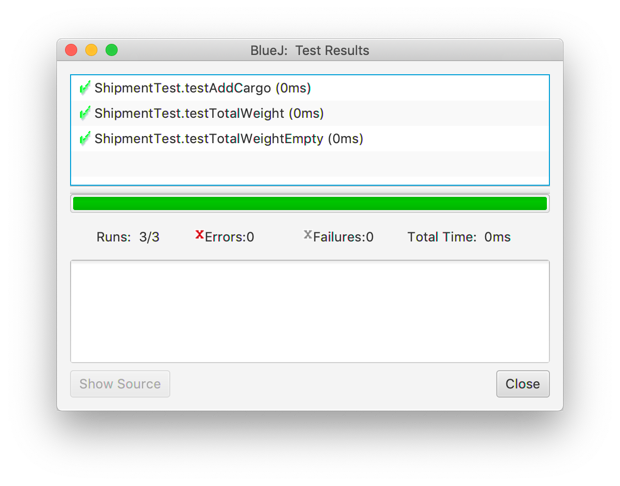
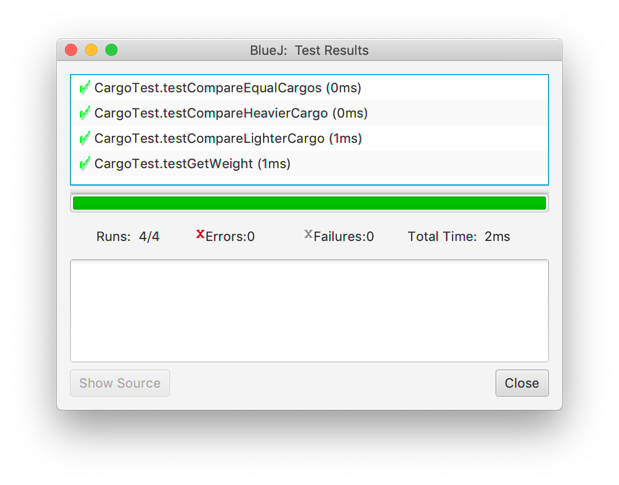
Output files were written with class PrintWriter (*PrintWriter (Java Platform SE 7)*) after experiment. From the terminal, diff command was run to compare the program’s generated output file with the provided sample output. Since the command line returned no difference, it can be seen that the content of the two files matched up completely (Figure 2).



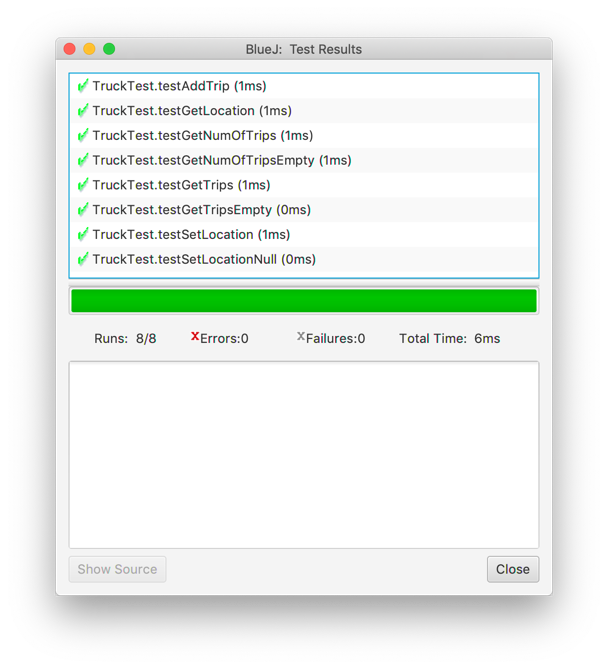
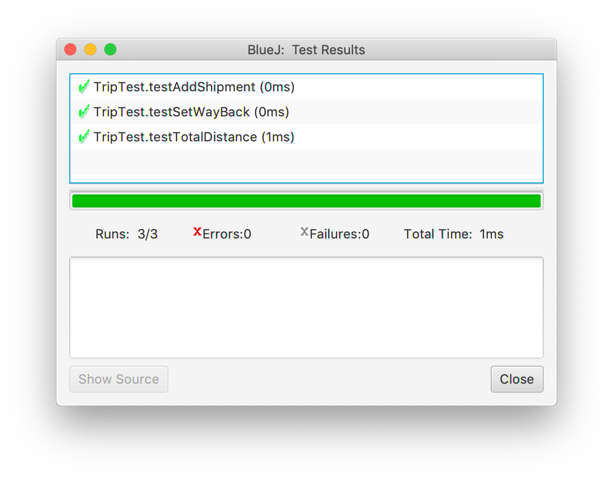
*Figure 2.* Running the diff command from terminal to compare the output and the sample



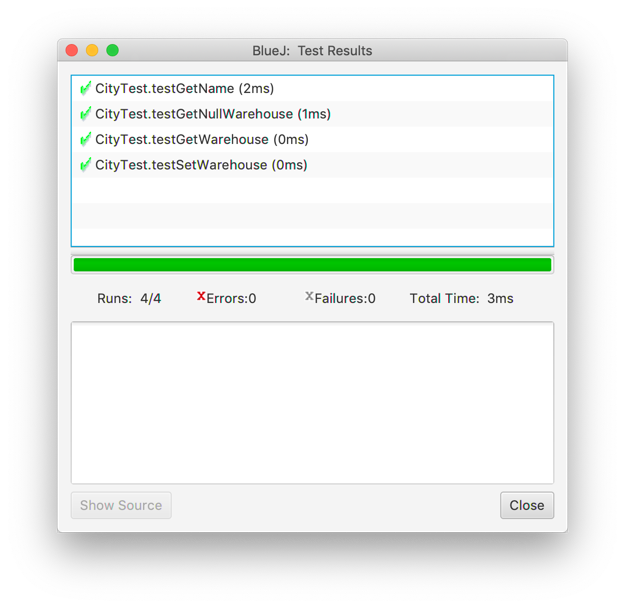
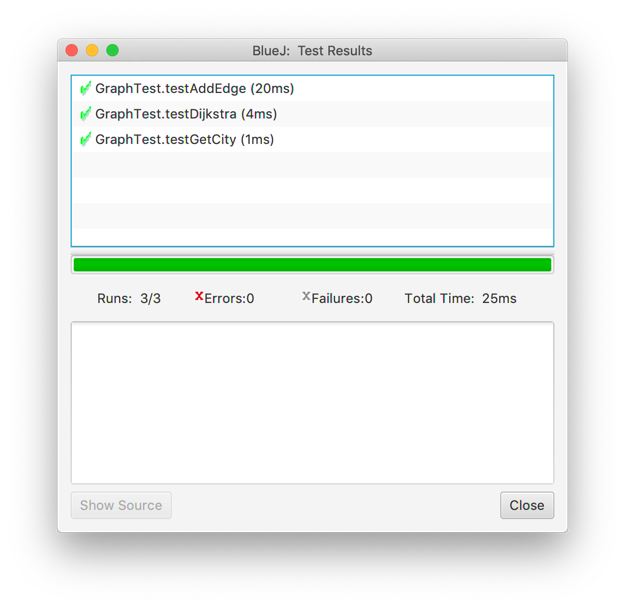
*Figure 3*. Unit tests for classes Warehouse and DistToWarehouse



*Figure 4*. Unit tests for classes Cargo and Shipment



*Figure 5*. Unit tests for classes Trip and Truck

**

*Figure 6.* Unit tests for classes Graph and City

Apart from the main program that ran the experiments, JUnit test classes were also created for the purpose of testing self-written methods (Figure 3, 4, 5, and 6).

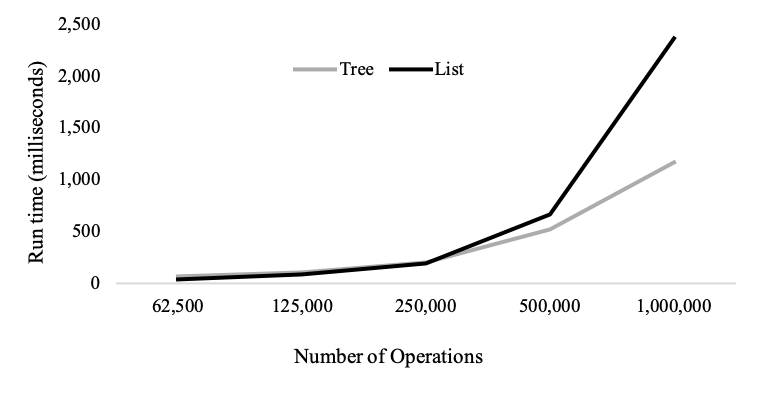
**Data and Analysis**

Table 2

Run time for Tree and List (milliseconds)

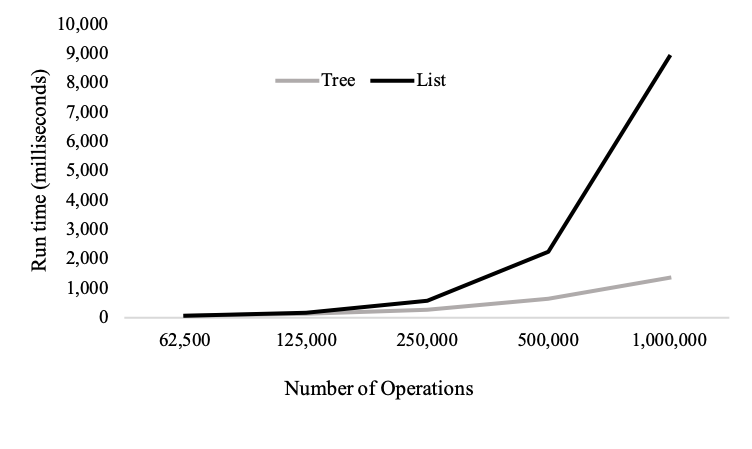
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Data Structure | Tree | | | List | | |
| Scenario  Number of  Operations | (1) | (2) | (3) | (1) | (2) | (3) |
| 62,500 | 62.8 | 73 | 79.6 | 35.6 | 65.2 | 99.2 |
| 125,000 | 106.4 | 135.8 | 153.8 | 89.8 | 176.2 | 332.4 |
| 250,000 | 206 | 270 | 354.2 | 189.6 | 582.6 | 1,273.4 |
| 500,000 | 520 | 620.8 | 655.8 | 668.2 | 2251 | 4,993 |
| 1,000,000 | 1,165.6 | 1,359.4 | 1,608 | 2,358.8 | 8,898.8 | 19,656 |

The average run times for three scenarios (1) 25% of operations are add and 75% are getKth, (2) 50% of operations are add and 50% are getKth, and (3) 75% of operations are add and 25% are getKth were calculated and summarized in Table 1.

****

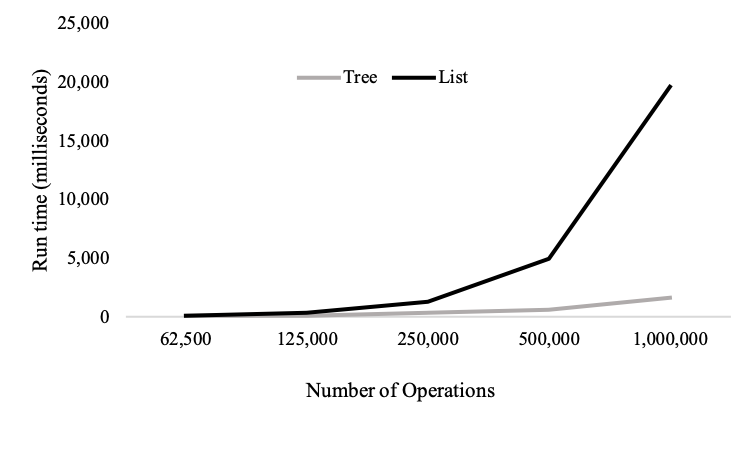
*Figure 3.* Average run times of Tree and List in

scenario (1) 25% of operations are add and 75% are getKth.

****

*Figure 4.* Average run times of Tree and List in

scenario (2) 50% of operations are add and 50% are getKth.

****

*Figure 5.* Average run times of Tree and List in

scenario (3) 75% of operations are add and 25% are getKth.

As indicated in Figure 3, 4, and 5, average run times increased for both the Tree and List approaches when the total number of operations increased across all three scenarios. However, the difference in run time growth became more significant when the percentage of add operations surpassed the percentage of getKth operations.

In scenario (1), at small numbers of operations of 62,500 and 125,000 (Figure 3), List performed slightly better than Tree because List took constant time to retrieve indexed elements in the Array List, whereas Tree took to find the desired element. Nevertheless, even in scenario (1), the growth in run time of List soon surpassed to that of Tree at higher number of operations because when there were more add operations, still grew faster than .

In scenario (2) and (3), Tree consistently performed better than List across all data points of number of operations (Figure 4 and 5). More add operations meant the constant getKth time of List no longer provided the approach with a significant advantage over Tree. Instead, Tree proved too be a prominently more efficiently approach than List with higher number of total operations and a higher proportion of add operations due to the slow growth of its complexity for both add and getKth operations.

**Conclusion**

In conclusion, when it comes to small number of inputs and retrieving data, indexed List is the more efficient approach than Tree due to its constant retrieval time. However, in most cases, Tree is more favorable of an approach than List because the growth in time complexity of Tree is much slower than that of List, especially with significantly larger number of operations and when the program requires more insertions than searches.

References

*ArrayList (Java Platform SE 8 )*. https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html. Accessed 20 Nov. 2018.

*Collections (Java Platform SE 7 )*. https://docs.oracle.com/javase/7/docs/api/java/util/Collections.html. Accessed 20 Nov. 2018.

*PrintWriter (Java Platform SE 7 )*. https://docs.oracle.com/javase/7/docs/api/java/io/PrintWriter.html. Accessed 14 Oct. 2018.

*PriorityQueue (Java Platform SE 7 )*. https://docs.oracle.com/javase/7/docs/api/java/util/PriorityQueue.html. Accessed 14 Oct. 2018.

*Scanner (Java Platform SE 7 )*. https://docs.oracle.com/javase/7/docs/api/java/util/Scanner.html. Accessed 14 Oct. 2018.

Xia, Ge. *Graph.Java*. https://moodle.lafayette.edu/pluginfile.php/437735/mod\_resource/content/2/Graph.java. Accessed 8 Dec. 2018.