INTRODUCTION

The aim of this project[1] was to develop an algorithm to find the shortest/most efficient routes for vendors to travel while delivering products to customers in several cities. The project was an extension of Project 2[2] to assume the possibility of a delivery system and calculate the most profitable routes for every vendor. The development goals were primarily focused around creating an efficient algorithm to find the routes/tour for every vendor in an attempt to maximize profit. Minimization of distance travelled while ensuring the satisfaction of every customers needs was focused upon. The analysis goals included comparatively analyzing how the algorithm would perform under different distributions of cities, vendor warehouses, etc. This would serve to test the performance of the algorithm in a variety of situations and determine its overall efficiency.

Certain assumptions were made to simplify the task at hand and allow the programmer to focus on the key requirements and areas of importance. Some of them were as follows-

- The only user input to the program would be the cost per distance travelled for the vendors. This would be input as parameter while starting the program.
- Every customer would be assumed to purchase exactly one unit of every type of item needed.
- Every warehouse for the same type of good would sell their products at the same price. Thus, there would only be one set price for each type of good.
- The cities for each customer were to be randomly generated along a uniform distribution from a given set of cities that can be modified by the client.
- The customer generation times for a simulation run and the individual customer needs would be determined based on values from different Gaussian distributions. They are described as follows:

Customer generation times-

Mean	Standard Deviation
61 seconds	31 seconds

Percentage of customers who would need types of items-

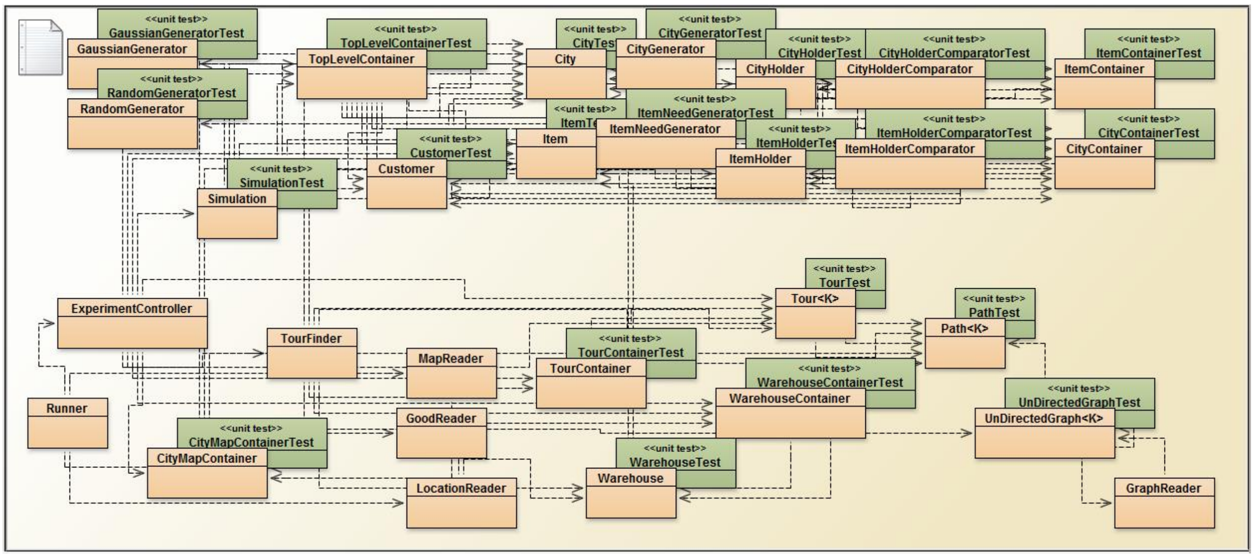
	Mean	Standard Deviation
Baked	37%	17%
Meat	53%	13%
Dairy	59%	19%
Fruit	47%	13%
Vegetable	71%	29%
Beverage	43%	11%

[1] Liew, Chun W. "CS 150: Project 3 Description." (n.d.): n. pag. 22 Nov. 2015. Web. 7 Dec. 2015. < https://moodle.lafayette.edu/pluginfile.php/198324/mod_resource/content/1/p3.pdf>.

[2] Liew, Chun W. "CS 150: Project 2 Description." (n.d.): n. pag. 27 Oct. 2015. Web. 29 Oct. 2015. <https://moodle.lafayette.edu/pluginfile.php/194458/mod_resource/content/3/p2.pdf>.

It was hypothesized that as the number of warehouses for any type of good would increase, the efficiency of the algorithm would reduce due to increased calculation time and a reduction in the probability of the calculated routes being the most profitable. If the density of the distribution of warehouses were to not change, an increase in the number of cities and the overall size of the graph would also be expected to affect the performance of the algorithm in a similar way as described above.

APPROACH



The developed solution was divided into several classes, each serving a different functionality-

- Customer class- This class was used to generate Customer objects with a randomly assigned home city and set of needs. Provides functionality to query customer needs.
- City class – This class was used to generate city objects.
- Item class – This class was used to generate item objects.
- CityContainer class – This class was used to store multiple city objects to be conveniently passed as parameters to/from methods.
- ItemContainer class – This class was used to store multiple item objects to be conveniently passed as parameters to/from methods.
- ItemNeedGenerator class – This class was used to randomly generate random needs for each type of item based on the fixed constraints.
- CityGenerator class – This class was used to randomly generate random cities for customers based on the client provided list of cities.
- TopLevelContainer class – This class was used as the top level container with a HashMap[3] as the data structure for the container. The keys for each entry in the hash map were city names and the values were CityHolder objects that acted as sub containers. This class provides functionality to output the data according to user input commands and constraints.

[3] *Java Platform SE 8 API Specification*. Oracle, n.d. Web. 12 Nov. 2015. <https://docs.oracle.com/javase/8/docs/api/>.

- CityHolder class – This class was used as a secondary container with a HashMap as the data structure for the container. The keys for each entry in the hash map were item types and the values were ItemHolder objects that acted as sub containers.
- ItemHolder class – This class was used as a tertiary container with a LinkedList[4] as the data structure for the container. The actual Customer objects were stored in the linked lists.
- CityHolderComparator class – This class was used as a comparator to compare CityHolder objects based on the number of customers in each container who needed a specific set of items.
- ItemHolderComparator class – This class was used as a comparator to compare ItemHolder objects based on the number of customers in each container.
- Simulation class – This class was used to actually run the market simulation to randomly generate customers and add them to the existing top level container.
- ExperimentController class – This class was used to store the map of cities, container for customers and set of cities for every instance of the program. It provides functionality to store information used to calculate the shortest routes and subsequently the profit. It also provides all the programs outputs to the terminal.
- Runner class – This class provides functionality to accept a user parameter and read certain data sets from external files. It subsequently passes all this data to the ExperimentController.
- GaussianGenerator class – This class provides functionality to generate random numbers in a Gaussian distribution with a given mean and standard deviation.[5]
- RandomGenerator class – This class provides functionality to generate random numbers within a given range of bounds.[5]
- TourFinder class – This class contains the algorithms and functionality to perform the shortest route calculations for a set of warehouses supplying a given item type.
- CityMapContainer class – This class stores the city map as using an undirected graph and provides basic functionality to insert and retrieve information from the graph.
- MapReader class – Used by the Runner class to read and parse the map details from an external file.
- GoodReader class – Used by the Runner class to read and parse the item price details from an external file.
- LocationReader class – Used by the Runner class to read and parse the warehouse location details from an external file.
- UnDirectedGraph class – Uses a HashMap to store the graph data structure for this project. Provides functionality to add nodes, edges and calculate shortest path between two nodes using Dijkstra's algorithm.
- GraphReader class – Used to read a graph from an external file.
- Path class – Used to create path objects to store a LinkedList of nodes comprising a path in the graph.
- Tour class – Used to create tour objects to store a LinkedList of paths comprising a tour(route) in the graph.

[4] *Java Platform SE 8 API Specification*. Oracle, n.d. Web. 12 Nov. 2015. <https://docs.oracle.com/javase/8/docs/api/>.
[5] *Java Practices - Generate Random Numbers*. Hirondelle Systems, n.d. Web. 9 Oct. 2015. <http://www.javapractices.com/topic/TopicAction.do?Id=62>.

- TourContainer class – Used to store a set of tour objects in an ArrayList.
- Warehouse class – Used to create Warehouse objects that store the city and item type of the specific warehouse.
- WarehouseContainer class – Used to store a set of warehouse objects in an ArrayList.

The main **algorithm**[6] used to calculate routes for each warehouse works as follows-

1. The set of cities where a specific type of good needs to be delivered is determined by checking if there is at least one customer in each city that needs that type of good.
2. For every city in this set the shortest path to each warehouse is calculated. The warehouse that is closest to each city will visit that city in its tour. Thus, the total number of cities in the set is divided amongst each warehouse and each will supply to those specific cities.
3. Using the cities for each warehouse, the nearest unvisited city from the list will be visited at each iteration and after the last city is reached, the vendor will return to the warehouse. **The algorithm does not prevent passing through a city that has already been visited if it happens to fall in the shortest path to another unvisited city.**

METHODS

The main parameter used for the experiments was the size and density of the graph determined by the number of cities and number of edges, respectively. Additionally, the distribution and number of warehouses was also varied in the graphs used for testing. The only user input parameter to the program is the cost per distance unit and for testing purposes this was set to 5.

As described in the introduction, the statistical parameters to determine customer generation and city generation were set as fixed constraints.

The experiments were run by varying the parameters sequentially based on given datasets. A total of 11 different graph distributions were tested.

The 11 distributions tested differed according to the following parameters-

1. Number of cities
2. Number of edges
3. Number of warehouses for each type of good
4. Actual distribution of cities and warehouses

Each experiment configuration was run 5 times to obtain **average values.** An overall comparison of timings for calculating the routes over each dataset was performed.

DATA & ANALYSIS

The data obtained has been described and analyzed as follows-

Overall statistics-

[6] Barton, Darren. "Nearest Neighbour Algorithm - Part 1." *YouTube*. N.p., 23 Sept. 2011. Web. 7 Dec. 2015. <https://www.youtube.com/watch?v=JH2IUFmP8JI>.

A comparison of the timing values obtained from each of the configurations has been presented as follows-

- Timing comparison for finding routes-

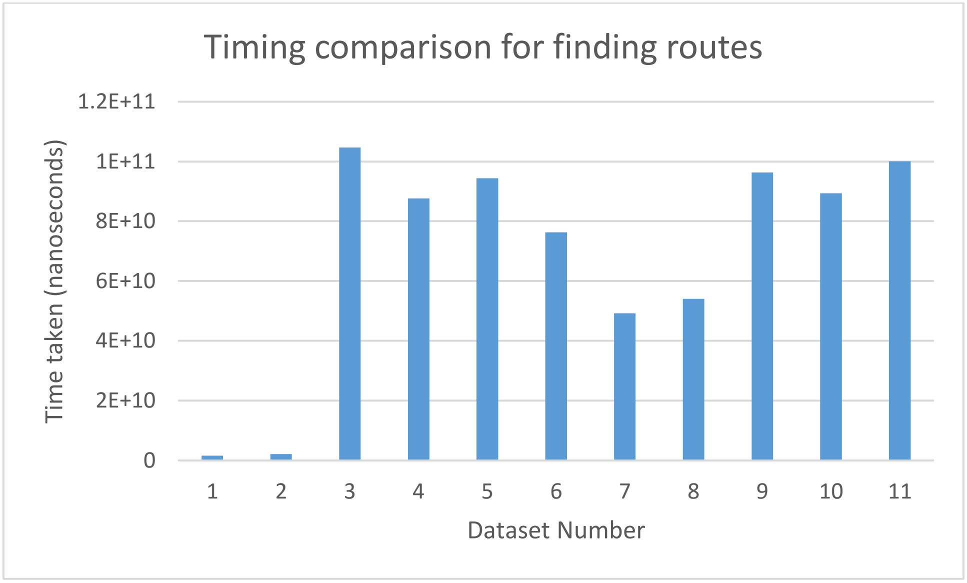


Figure 1

Evaluation of solution-

Evaluating the efficiency of the algorithm in terms of finding the most profitable routes, we realize that the algorithm used is not the most optimal solution. This is because the algorithm is very expensive and often wasteful. The algorithm is expensive due to the fact that every possible path must be calculated at every step of the iteration to determine the shortest path. This in turn leads to the wastefulness because despite the expensive calculations, we can end up with situations where the same city is visited multiple times. Often a city that is to be supplied by one warehouse ends up in the tour of another one, simply because it happens to be in the shortest path to another city being visited. Due to these reasons, the algorithm is not optimal, however, it would provide a much more efficient solution than any algorithm that involves calculating random paths between cities.

CONCLUSION

The comparative performance analysis of the different dataset/graph configurations provided a good idea of performance of the algorithm as the various graph parameters were changed.

As hypothesized, we can see from Figure 1, the most prominent difference in performance was due to the overall graph sizes given by the number of cities. Datasets 1 and 2 had 100 cities whereas the other datasets had twice the number of cities as the first two. It is quite evident that this had the most significant impact on timing performance.

Amongst datasets 3-11, we can further determine which factor was the next most significant in affecting performance. From these sets, dataset 6 and 7 appear to perform relatively more efficiently. This difference is due to a lower number of edges as compared to the others. This seems to be consistent with the hypothesis. It seems that the number of warehouses per type of good did not seem to have a great effect on performance. The expected performance difference due to the varying warehouse count between datasets was not consistent with the observed difference in performance. This conclusion differs from that hypothesized.

This result can be understood by recognizing that an increase in the number of warehouses does not actually lead to an increase in the size or density of the graph. Thus, we can conclude that the algorithm efficiency is most influenced by graph size and density.

REFERENCES

Weiss, Mark Allen. *Data Structures and Problem Solving Using Java*. Boston, MA: Pearson/Addison-Wesley, 2010. Print.

Java Practices - Generate Random Numbers. Hirondelle Systems, n.d. Web. 9 Oct. 2015. <http://www.javapractices.com/topic/TopicAction.do?Id=62>.

Java Platform SE 8 API Specification. Oracle, n.d. Web. 12 Nov. 2015. <https://docs.oracle.com/javase/8/docs/api/>.

Liew, Chun W. "CS 150: Project 2 Description." (n.d.): n. pag. 27 Oct. 2015. Web. 29 Oct. 2015. <https://moodle.lafayette.edu/pluginfile.php/194458/mod_resource/content/3/p2.pdf>.

Liew, Chun W. "CS 150: Project 3 Description." (n.d.): n. pag. 22 Nov. 2015. Web. 7 Dec. 2015. < https://moodle.lafayette.edu/pluginfile.php/198324/mod_resource/content/1/p3.pdf>.

Talwar, Nakul. "Nearest Neighbour Algorithm - Part 1." *YouTube*. N.p., 23 Sept. 2011. Web. 7 Dec. 2015. <https://www.youtube.com/watch?v=JH2IUFmP8JI>.

People discussed project with-

- Agathe Benichou
- Andrew Ortiz
- Jack Plumb
- Liam Mungovan
- Thomas Clagett