Train Dispatch

Team 6:

Emily Hazen
Rachel Froling
Taylor Carter

How the System Works:

Our task was to develop a train dispatching system that can send trains in such a way that the total time it takes for a train to reach its destination is minimized. To do such we have developed an algorithm to optimize the average time cost per train when compared to a baseline dispatching system.

The system contains three packages: *graph, main*, and *schedule*. Package *graph* contains all classes relevant to the Multigraph system. Class *main* contains all classes for dispatching trains and finding their shortest paths. Package *schedule* contains all classes relevant to the scheduling system.

For the train system, we used a priority queue to sort the schedules and poll them in order. While there were still trains in the queue, we would dispatch trains that were ready, that is, trains with a dispatch time greater than or equal to the current time.

A sprite system handles the train's display as well as their location. Sprites move one weight unit per time unit. That is, a sprite on an edge of weight 3 will travel a third of a way down the edge each time one time unit passes. Because the graph displays all edges as the same length, the weight of an edge determines the speed at which the sprite will travel; i.e., a sprite will travel slower on an edge with a larger weight.

The cost of a train is calculated from the time it arrives at its destination minus the time the train was ready to dispatch. When the train arrives at its destination, the sprite for that train is removed. The program ends when there are no more trains to dispatch and all trains have arrived at their destination.

In the base case, we used a basic single-source shortest path algorithm that would skip any edge marked locked. If no path was found, that schedule would be skipped and will be continuously checked during the course of the algorithm until a path opens up. If a path is made available, then all of the edges on that path will be locked for the use of that train. When a train reaches the end of its path, then all edges in that path are unlocked.

In the improvements, we used a reservation system that kept track of when a train was going to be in use. When calculating the shortest path, the amount of time spent waiting for an edge to open up is added to the cost of using that edge. This way, the algorithm can determine if it is more cost-effective to wait for a path to open up or to use a path that is available immediately. Once a path is chosen, each edge is reserved for the time at which the train will be using it. This is determined by the time at which the train will arrive at the edge and the time at which the edge is available for use.

For the reservation system, a list of integers is attributed to each edge. The availability of an edge is checked using a method *checkReservations* that takes an edge and an arrival time as parameters. The arrival time indicates the time we would like to start looking for an availability. Method *checkReservations* determines if the time that a train would like to use an edge interferes with other reservations on that edge. If there is an interference, then the method determines the next time the edge is available and attributes this value to the edge. Dijkstra's uses the availability minus the arrival time of a train to determine the delay of that edge. Sprites keep a list of their own reservations to determine if they must wait at a station or continue to the next edge.

If a sprite arrives at the next edge on its path before that edge is available, then the sprite is delayed. Delayed sprites stay at a station until it is time for their reservation. When a train can use an edge, it locks only the edge that it is on and unlocks the edge once it has reached the next station.

Data Structures:

A *Multigraph* is used to store, retrieve, and display nodes and edges. We extend this data structure to generate graphs, return edge lists, and perform a few other operations.
A *Node* is used to store and retrieve the attributes of a station.
An *Edge* is used to store and retrieve the attributes of a track.

A *SpriteManager* is used to store, retrieve, and display sprites that represent trains.
A *Sprite* is used to store and retrieve the attributes of a train

A *Schedule* contains information about the source node, destination node, and dispatch time of a train.
A *PriorityQueue* is used to sort a list of schedules and poll the next schedule available.

Lists are used throughout the program for the reservation system, paths, removing sprites, etc.

Algorithms:

Dijkstra's single-source shortest path algorithm is used to determine the path of a train. The base case uses a standard Dijkstra's, simply skipping locked edges. The improved case uses an adapted version of Dijkstra's that considers an edge's delay as part of its cost.

Weekly Log:

March 7 - March 13

Everyone met up and began to discuss parameters, goals, implementation of classes, the base case, and pseudocode for the project.

March 14 - March 20

Dijkstra's was studied and everyone collaborated to set up SourceTree as a Git repository to be able to share and maintain one uniform set of code despite the use of several machines.

March 21 - March 27

Emily began modifying Dijkstra's Algorithm for our purposes.
Rachel started a class that generates graphs for Dijkstra's Algorithm when given a file.
Taylor created classes for a schedule object, it's creation, and the method to sort them.

March 28 - April 3

Emily created classes to convert graphs for algorithms into graphical displays and worked extensively on the main driver class for dispatching trains. .
Rachel improved her methods of graph generation and collaborated with Emily to convert algorithmic graphs into displays.
Taylor improved the schedule sort methods, and added priorities to schedules of trains that could be arbitrarily assigned, set to specific values, and compared to other schedules.

April 4 - April 10

Emily resolved several SourceTree merge errors, created a method in Dijkstra's Algorithm
that returns a path as a list of edges.
Rachel added methods within the graph classes to get sources and destinations as vertices.
Taylor changed the schedule class to convert the given source and destination from integers to vertices that could be used in our version of Dijkstra's Algorithm.
The team worked together on understanding the logical flow of the base case and cost calculation.

April 11 - April 17

Emily organized classes into packages and implemented a new data structure to store locked edges. She also created a new driver file that runs the train dispatching and consists of the main method. Emily managed to make the code successfully run both algorithmically and graphically.

Rachel altered existing code to make it much more readable and set clear variable naming conventions.
Taylor heavily worked on the creation of pseudocode for cost calculation, the base case and improvements. He also began work on the documentation of the project.

April 18 - April 24

Emily created a reservation system for edges to efficiently dispatch trains. She made sure both the base case and improvements were completed, as well as organized files to optimize user interface. She also debugged all files and implemented them into the driver. Emily then created the README.txt file for the project.
Rachel created new graphs for train systems akin to those in Athens, Berlin, and Paris. She made sure these worked with the current implementation of the dispatch system. Rachel also worked heavily on the presentation.
Taylor altered the existing random schedule generation code to write files, traced both the base case and improved systems, ran multiple statistics on each graph with multiple schedules, and continued to work on the documentation.

Performance Analysis:

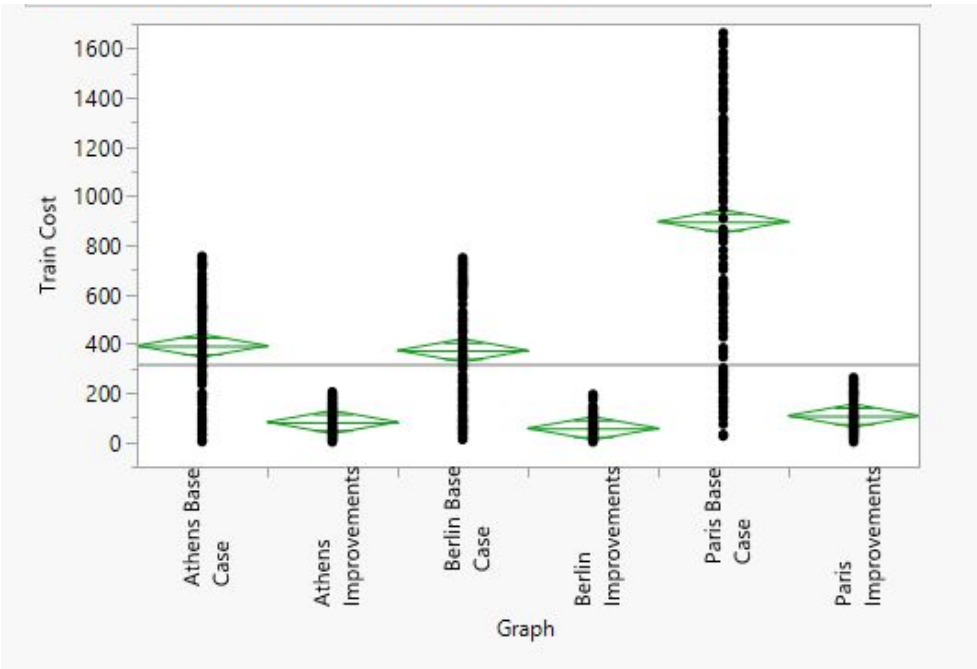

Figure 1. 100 trains dispatched in 25 time intervals ANOVA representation.

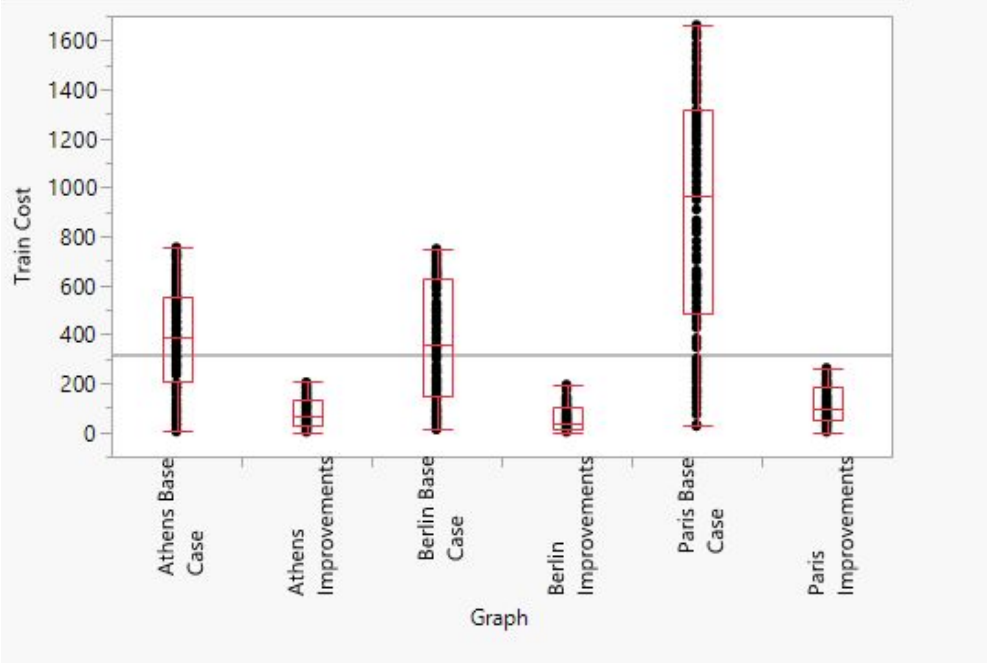

Figure 2. 100 trains dispatched in 25 time intervals box and whiskers representation.

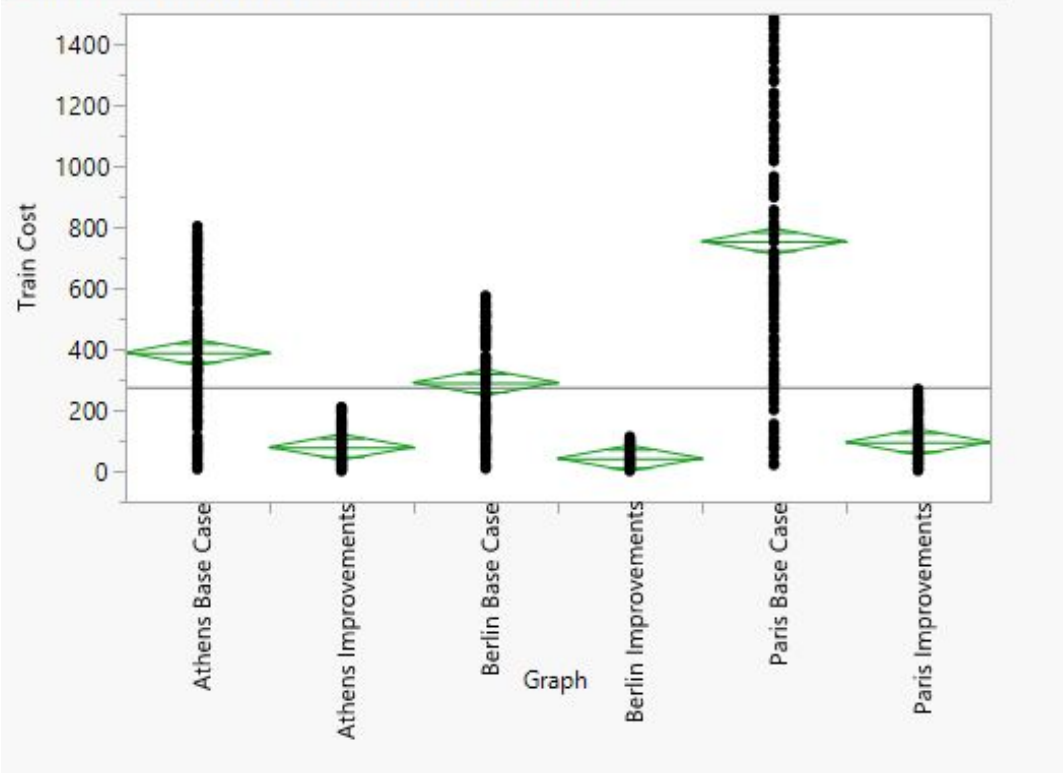

Figure 3. 100 trains dispatched in 50 time intervals ANOVA representation.

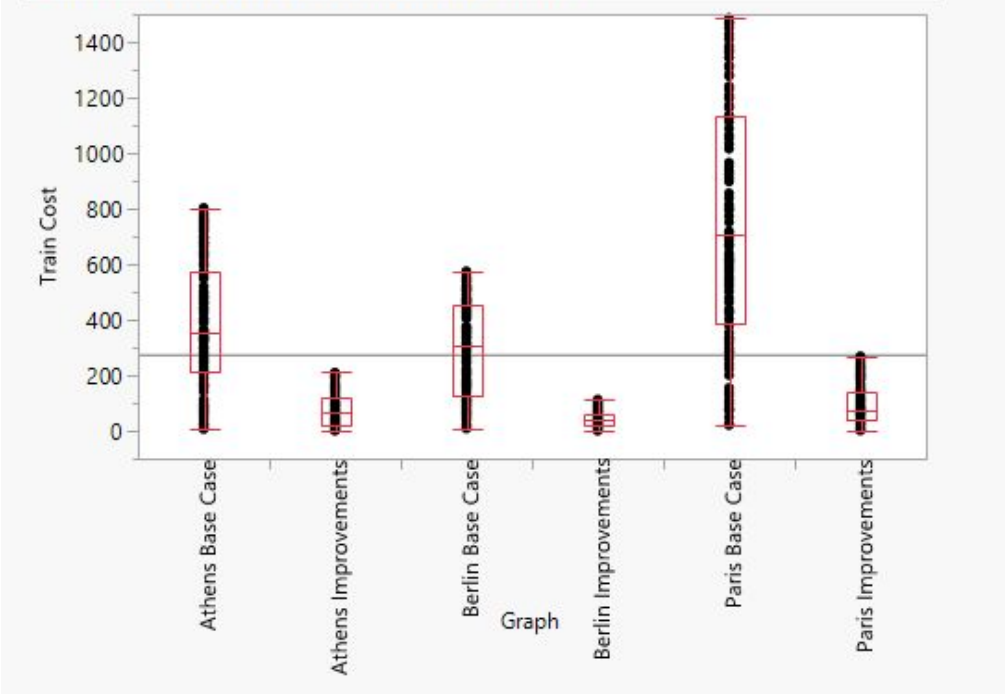

Figure 4. 100 trains dispatched in 50 time intervals bar and whisker representation.

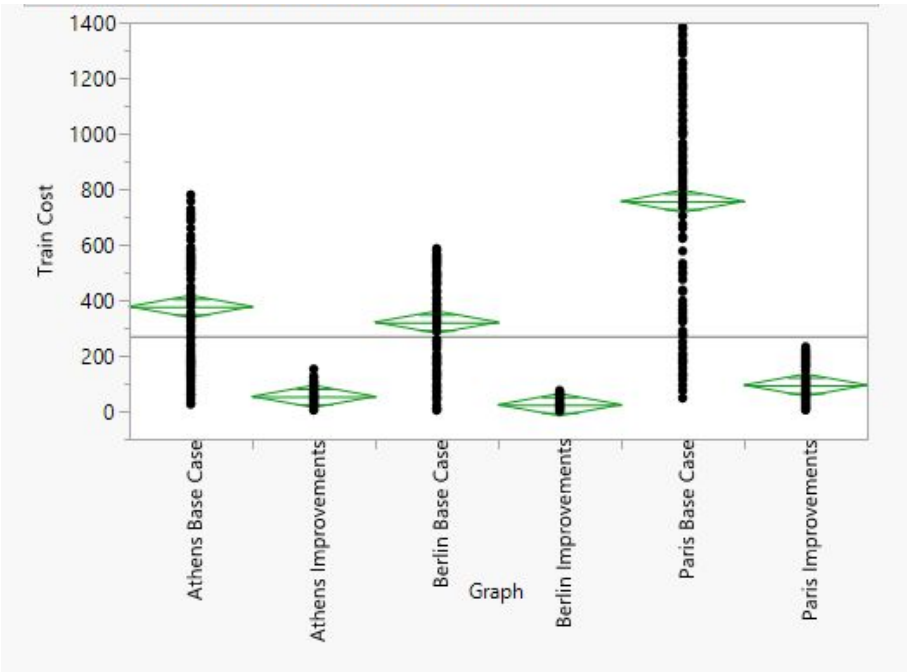

Figure 5. 100 trains dispatched in 100 time intervals ANOVA representation.

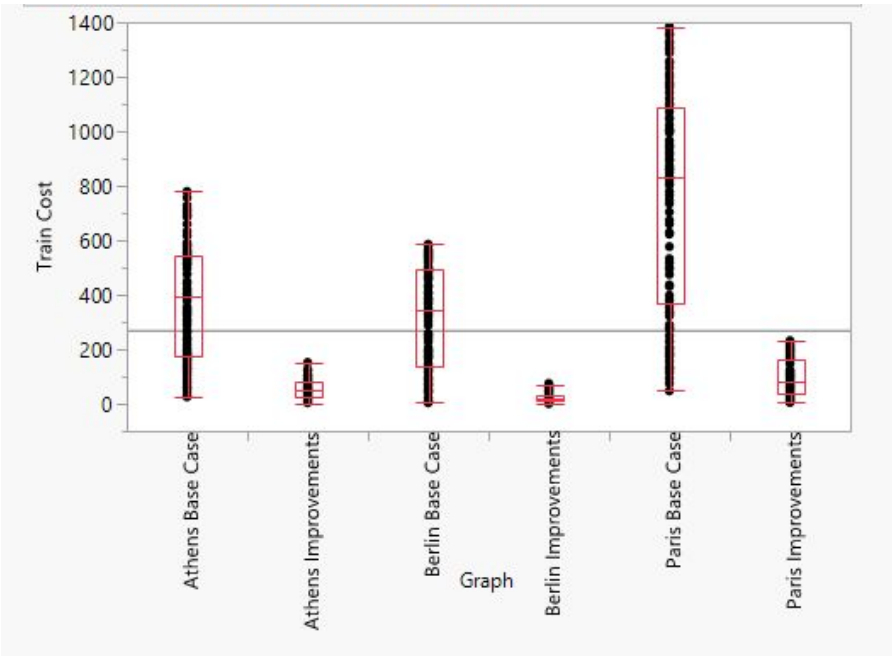

Figure 6. 100 trains dispatched in 100 time intervals bar and whisker representation.

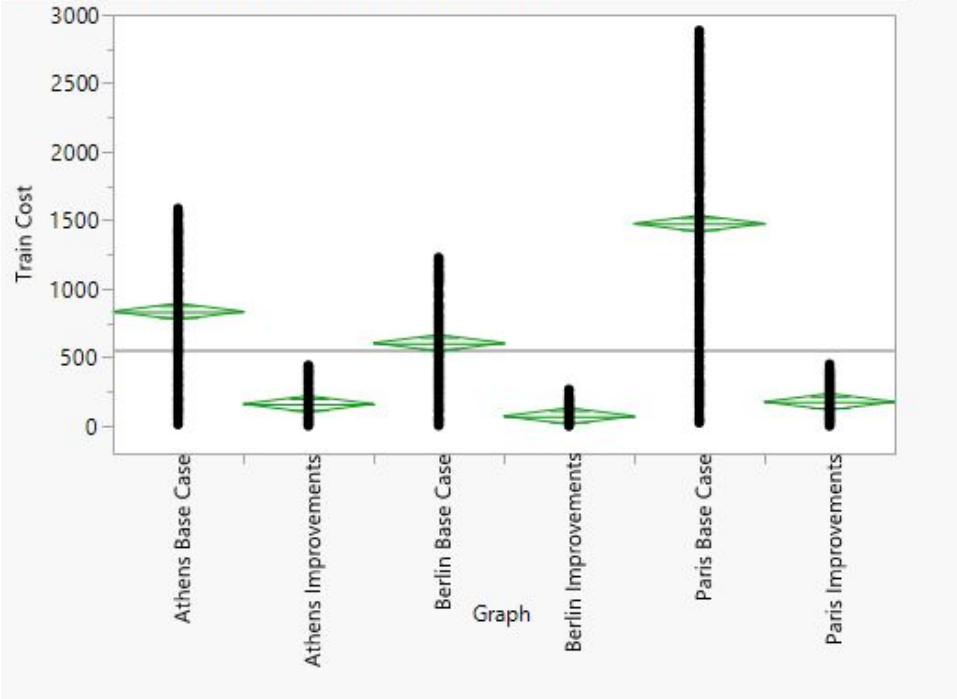

Figure 7. 200 trains dispatched in 25 time intervals ANOVA representation.

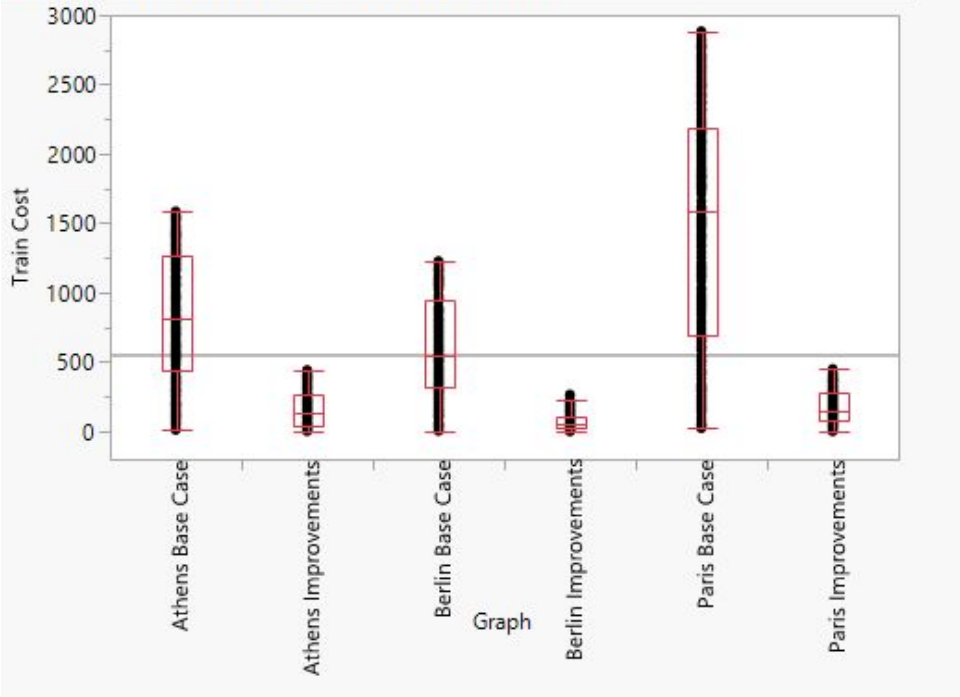

Figure 8. 200 trains dispatched in 25 time intervals bar and whisker representation

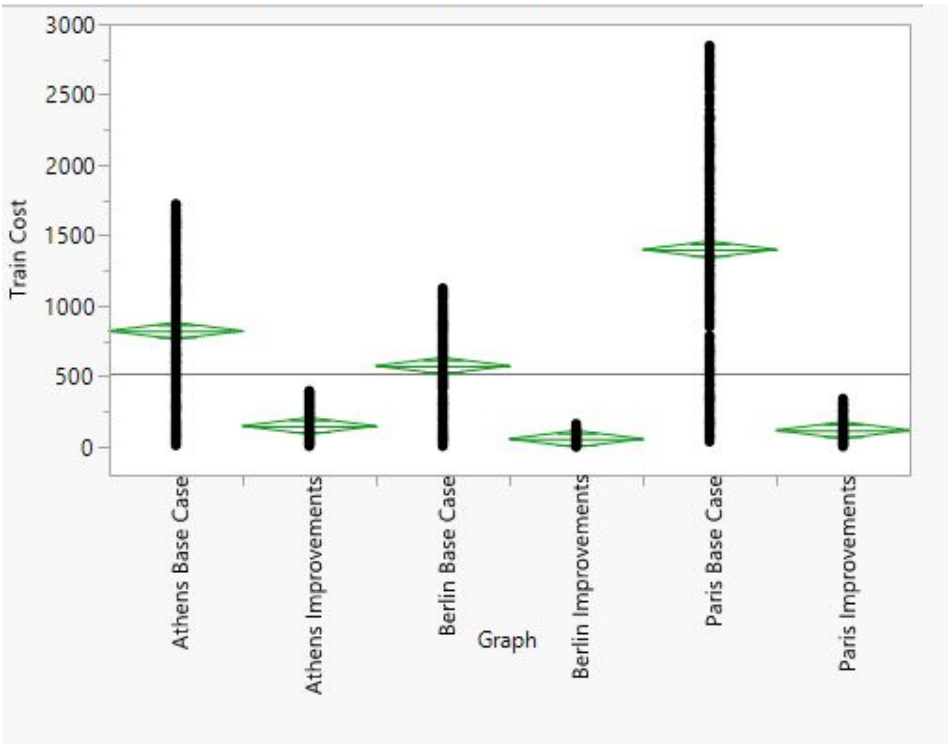

Figure 9. 200 trains dispatched in 50 time intervals ANOVA representation.

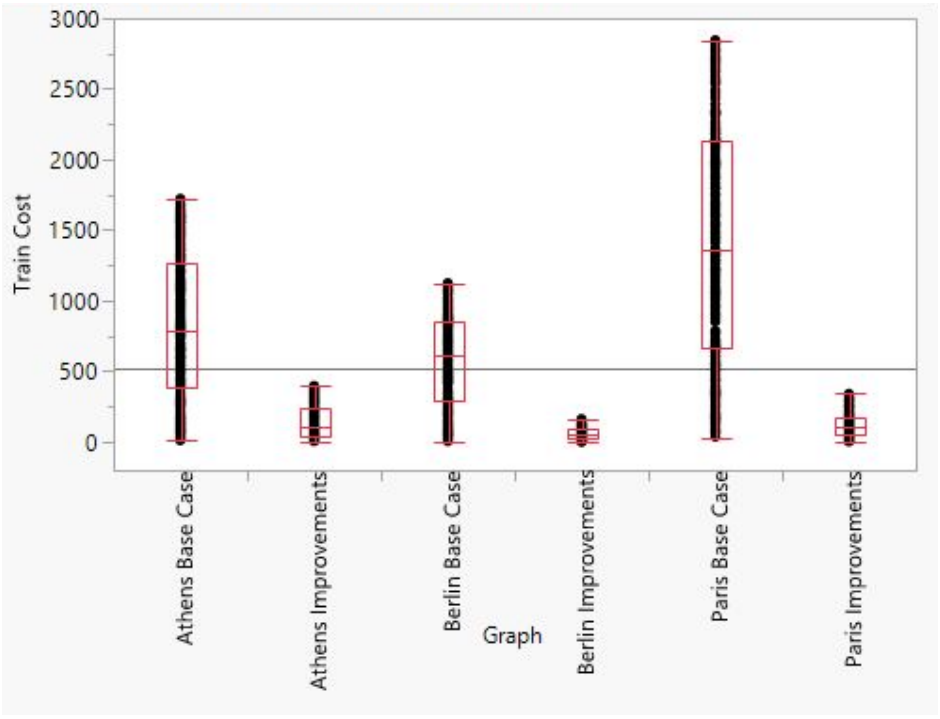

Figure 10. 200 trains dispatched in 50 time intervals bar and whisker representation.

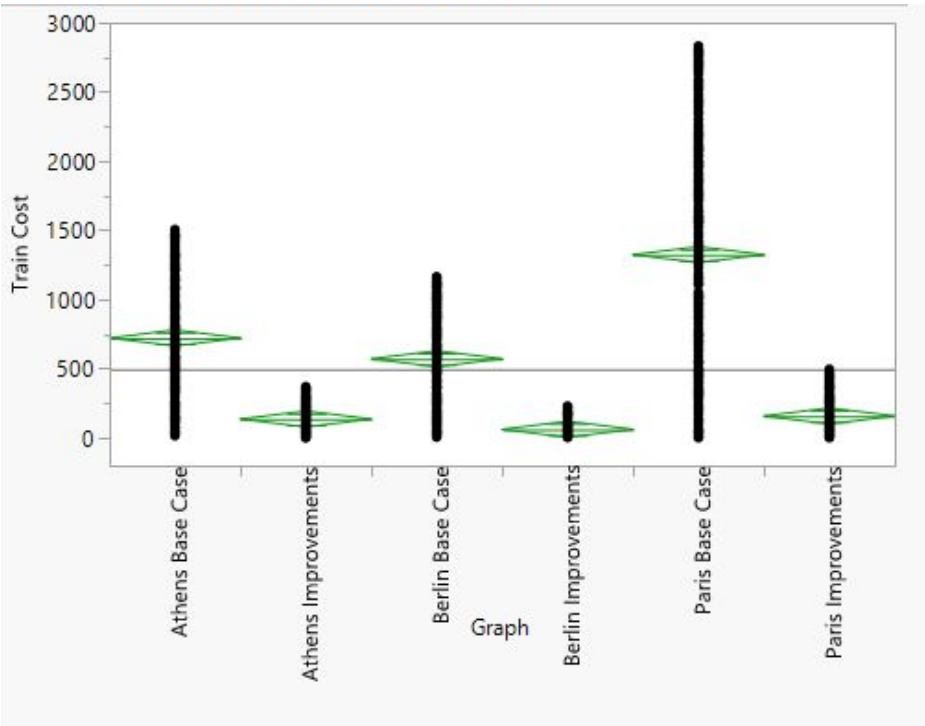

Figure 11. 200 trains dispatched in 100 time intervals ANOVA representation.

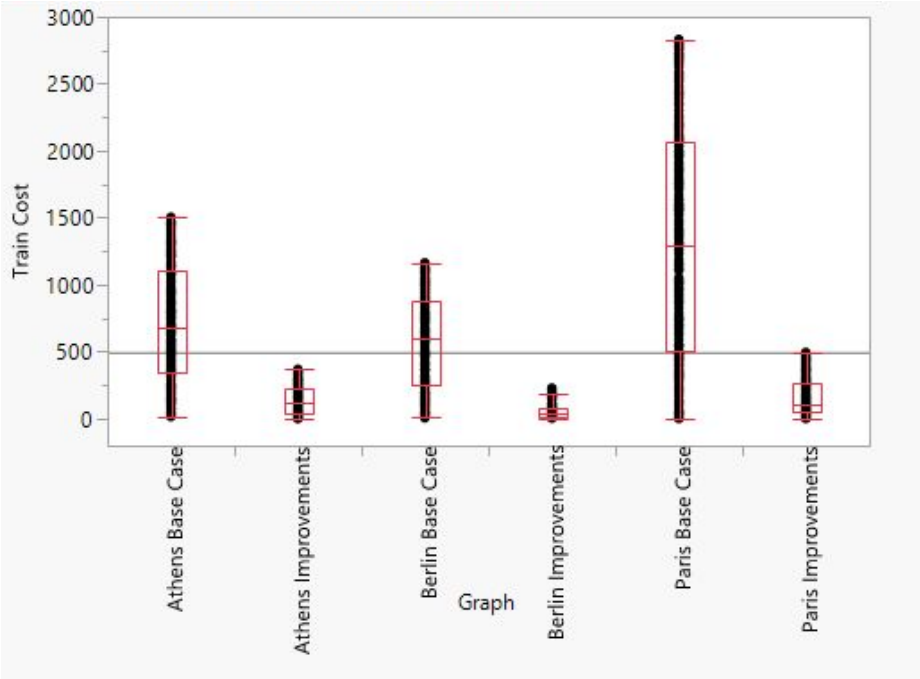

Figure 12. 200 trains dispatched in 100 time intervals bar and whisker representation.

Two types of graph plots were used to analyze the performances of both the Base Case and Improved train dispatch systems. The first was a Means for Oneway Anova (analysis of variance), and the second was a box (bar and whisker) plot. These plots were performed on the same set of data from one figure to the next. It should be noted that the parameters of total number and trains and the latest time at which they could potentially be dispatched were uniform on each plot.

In each figure, the vertical lines are made up of black dots. Every dot is a single data point taken from the respective tests run on the graphs using randomized schedules and the parameters labeled. The horizontal grey bar of the plot represents the mean across all data points in the graph. Every graph is broken down into three sections as followed: Athens Base Case vs Athens Improvements, Berlin Base Case vs Berlin Improvements, and Paris Base Case vs Paris Improvements. These are labeled as the type of Graph on the x-axis. On the y-axis, the time cost of each train was plotted. This was determined as the difference in time in which a train was ready to be dispatched from the time at which it arrived at its destination.

The Means for Oneway Anova test can be used to determine the statistical significance between variables. The variable that was tested was the algorithm for the Base Case vs the Improved algorithm. The green diamond for each column signifies a 95% confidence interval. This means that the true mean of the data set has a 95% chance to be within the green diamond. The middle line within the diamond represents the group's sample mean, calculated as the sum of the costs for every train on that graph using that specific algorithm, divided by the number of trains. The upper and lower bars within the diamond are overlap marks. Overlap marks are used to indicate if two group sample means are statistically different or not. If one group's upper overlap bar is closer to a second group's mean than it is the second group's lower overlap bar then the two groups' sample means are not statistically different in this confidence interval. The overlap bars are calculated as the Group Mean ± ((√2) / 2) * (Confidence Interval / 2).

The box plots are used to display the spread of the data as well as the distribution within that spread. The lowest and highest data points aside from outliers are represented as 'whiskers', or horizontal lines at the end of a vertical line. These can be used to find the range of the sample data. The center line of a box plot is the median or middle value of the data set. This means that half of the data lies above the median line, whereas the other half lies below. The bottom and top lines of the middle box represent the medians of each half of the data. Overall the data is split into quartiles that each represent 25% of the data.

From these plots we have determined that our Improved train dispatching algorithm is statistically different from the Base Case train dispatching algorithm. By comparing the Means for Oneway Anova tests in particular, it can be noted that for every case, the Improved system

results in a statistically significant improvement of the average time cost to dispatch trains. There was no instance in which the overlap bars of Athens, Berlin, or Paris Base Case algorithm overlapped those of the Improved algorithm. In the case of Figure 1, the means of the Base Case and Improved Athens tests were 391.940 and 81.620 respectively. The lower overlap bar of the Athens Base Case test was at 349.10, whereas the upper overlap bar of the Athens Improved test was located at 93.87. Because the distance between the overlap bars is smaller than the distance between one's overlap bar and the other's mean, it can be concluded that the calculated group means are statistically different. It can also be noted that the Improved cases had similar group means both across all distributions of dispatch time and total number of trains, as well as between city graphs. Observing the box plots, it can be determined that the spread of the time cost of each train was much more compact and on average much lower for the Improved algorithm than that of the Base Case. The data sets had ranges of 202 and 753 respectively. The standard deviation for the Athens Base Case test was 215.09, whereas for the Athens Improved test it was 61.721. Such drastic difference in the spread of the data can be observed in the box plots. One should notice that changing the total number of trains or the time density at which they were dispatched seemed to have little effect on both distribution and average train cost per algorithm per city graph.

References:

Dijkstra's Algorithm
https://stackoverflow.com/questions/17480022/java-find-shortest-path-between-2-points-in-a-distance-weighted-map
http://en.literateprograms.org/index.php?title=Special%3aDownloadCode/Dijkstra%27s_algorithm_%28Java%29&oldid=15444

GraphStream Graphical Displays
http://graphstream-project.org/
http://graphstream-project.org/doc/Tutorials/Storing-retrieving-and-displaying-data-in-graphs/

Data Analysis
http://flowingdata.com/2008/02/15/how-to-read-and-use-a-box-and-whisker-plot/
http://www.jmp.com/support/help/Means_Anova_and_Means_Anova_Pooled_t_Options.shtml#402614