Train Dispatch

by

Charles Johnson
Zachery Hibbard
Joshua Jennings

Table of Contents

Description of System

Objective:

Create a graph-based train dispatch system that inputs a train schedule as well as a map of stations and tracks and outputs a result with the shortest path each train can take for the shortest overall time. This system must improve upon a base case. The base case states that when a train traverses a path, all edges are locked for the full duration of the traversal.

Rules:

Only one train may traverse an edge at a time, but multiple trains may occupy a node.

System:

Our system imports data from two text files, one will populate a graph with nodes and edges and the other is a list of trains traversals across the graph (only the starts and ends, not the paths). Once the data is stored, each train path is calculated in order. For the base case, the order is identical to the sequence of the input file whereas the optimized version first accounts for a priority weight and then the preferred leave time provided in the schedule.

This system uses a Depth-First Search. The algorithm searches the complete graph continually checking against predefined end conditions such as the current accumulated traversal weight being greater than the known minimum or the current node being the destination node. Once an initial valid path from the origin to the destination is found, the shortest path and the total weight of said path are saved. If another valid path with a lower total weight is found, the shortest path is reset to the new path. To optimize this, if any search matches or exceeds the total weight of the current shortest path, the search returns.

Conflicting paths are accounted for by storing time intervals during which a train is traversing an edge. While searching for paths, the DFS algorithm checks each edge's list of blocked time intervals to find the next available time and, if the edge is occupied, adds the time it would need to wait before being able to traverse the edge to the overall time weight calculation.

After the shortest path for each train is found, new intervals are created for each traversed edge corresponding to the time when that train was occupying the edge. If the time between two consecutive intervals for an edge is less than the weight of that edge, the intervals are merged into one. This optimization can save space and searching time, especially for larger inputs.

Once all the data for each train's shortest path is calculated and stored, an output is printed with data pertaining to each train's path as well as the time it took to reach its respective destination. The average time for all trains is also printed.

Afterwards, a visual representation of the graph is created using the GraphStream libraries. Both the data from the trains and the input graph are imported into the new one and a time-updating view is displayed. All train paths are then animated and the animation stops when the last train has reached its

destination. As each train traverses its path, the action taken by each train is printed with distinct colors and labels for each train.

Base Case Optimization:

This system improves upon the base case in three ways. First by locking edges only when a train is currently traversing a given edge. If there is enough time either before or after that train, another train may also traverse the same edge at that new time instead of only having access to the edge after the first train has reached its destination. Second, the optimization sorts the adjacency of nodes to edges in the adjacency list. Edges with lower weights are stores at a lower index so that the Depth-First Search traverses the lower weight edges before the higher weight edges. Finally, the trains are sorted in memory such that high priority trains take precedence. Further, trains with an earlier scheduled departure time are operated on before those with later scheduled departure times. Through these optimizations, the input has a more direct effect on the output by accounting for priority levels and in almost all cases, the overall time required for all trains to arrive at their respective destinations is lower than the base case.

Implementation

General Setup

1. Download zip file.
2. Decompress source code and place into a default package.
3. Save all text files and CSS file to a known location on computer.
4. Build paths for the six jar files used from graphstream.
5. Run the program through the TrainDispatch.java class.
6. When Prompted, enter in the appropriate Path.

Modifying Text Input

Two text files are required as input. The first is a list of station to station tracks and the second is a list of trains. The text files must have the following setup.

StationInput.txt

The first line is the total number of stations. All additional lines have three integer values separated by whitespace. There is only whitespace between values, but not at the end of a line or at the end of the file.

1. First Station
2. Second Station
3. Weight

If a Station has not yet been created, a new instance of a Station class is created and the connecting station is added to a list of track IDs for that Station. For each line, a new instance of the Track class is created with both stations and the weight of the edge between them.

Both Stations and the associated Track are given unique integer ID numbers. There can be multiple Tracks between two Stations.

Example of valid input for StationInput.txt:

```
3
2 1 4
1 3 6
2 1 4
3 1 5
3 2 1
```

Trains.txt

The first line is the total number of trains. All additional lines have four integer values separated by whitespace. There is only whitespace between values, but not at the end of a line or at the end of the file.

1. Origin
2. Destination
3. Departure Time
4. Train Priority

For each line beyond the first line, a new instance of a Train class is created. It stores the origin, destination, departure time, and priority. All values are positive integers. Both origin and destination refer to Stations (Nodes) within the graph. Departure time determines when the train leaves the origin station.

Priority is any number greater than or equal to zero (0) and, if not zero, determines which train's path is calculated first.

After the Trains are created, they are put into an array and sorted, first by the highest priority, then by the lowest departure time. Each Train will also have a unique ID number created for it.

Example of valid input for Trains.txt:

```
4
2 5 3 6
3 7 8 0
2 3 6 2
1 5 1 5
```

Code Breakdown

GraphDisplay:

The GraphDisplay class is used to give a visual representation of the graph and Train Paths. Up to this point, all shortest path calculations have been made that the data stored. To display this data, the program uses the GraphStream library from GraphStream Project (citation needed).

Using the resources provided by GraphStream, a new graph is created using the existing Station and Track data. For multiple Tracks between the same two Stations, a BitSet is used.

Once the new graph is complete, three nested for loops iterate through each Train's Path for each unit of time from zero to the last arrival time. When a Track or Station is occupied by a Train, a Train Sprite (each Train has one) is attached to the corresponding element. As "time" progresses, the Sprites traverse the graph.

GraphInput:

The GraphInput class is used to read in all data from the provided text files and assigns the data to its appropriate lo*cation for later access. This class can be broken down into three methods.

The first method in this class, getNumOfStations, returns the total number of stations in the graph, by reading in the first line of the file and parsing it to an integer.

The second method in the class, populateTracks, reads in the StationInput file using a BufferedReader and populates a stack with the data. Once the stack is populated, a Scanner is used to read off the stack and populate the track array with and appropriate arrangement of (id, weight, stationFrom, and stationTo). The stations array is also filled with data off the stack at this time.

The final method in this class, populateTrains, reads in the Trains file using a BufferedReader and populates the Train array with correct data. Once the array is filled with data, the method sorts the array of trains based on the priority the user predefined.

GraphSearch:

An iterative depth-first search on an undirected, weighted graph. It uses the Recursion class to maintain the current stack (path being searched) and constantly compares the current stack to the previously determined best path. Whenever a newly identified, shorter and valid path is found, it is copied and replaced. There is also a wrapper function that loops through all of the input train schedules and does a complete search on every one. This is where the $O(n^2 * k)$ runtime originates as the steps in GraphSearch.fullSearch() grow with the input size at that rate and overpower all other operations asymptotically.

Interval:

The Interval class creates a linked list of IntervalItems that store intervals of time when each track is occupied. IntervalItem is covered in the next part.

Every Interval has a weight integer as well as an IntervalItem set as the head of the list.

Each Track initializes a new Interval. When a train passes through a Track in its shortest path, a new IntervalItem is created containing the start and end time of the traversal. When the following Trains traverse the same Track, it checks the Interval class to find out if it is being used at that time.

The setOccupied method takes in an integer startTime and, if calculating the base case, an integer endTime. It then calls the insertItem method that creates a new IntervalItem with these values and inserts the item into the Interval list such that all nodes are in order from lowest to highest.

This method is used to find out if a time interval is available in the list and, if so, return the next available time. It does this by calling one of a series of boolean methods to compare IntervalItems. If no range either between or within the full range of the list is available, it returns the end time of the last IntervalItem.

If the space between two adjacent nodes is less than the weight of the Track, the nodes are combined using the mergeAdjacent function which is called directly after the insertItem method in setOccupied.

IntervalItem:

IntervalItem is another of the programs basic generic class that store data. This class stores time intervals to be used in Interval mentioned above. The constructor takes in a start and end integer and assigns them to local variables. As with a linked list node, it also takes in another IntervalItem and assign it to a next variable.

IntervalItem contains a toString method that will print out the start and end values in brackets as well as a boolean that is false if next is equal to null.

Path:

Every time a Train is created, a new instance of this Path class is initialzed with it. The Path is ultimately an array of PathItems. PathItems store traversed edges and are covered below. The Path constructor gets passed a PathItem list as a parameter and saves the list to its own as well as a length variable equal to the list size and a totalTime variable which is the arrivalTime of the list parameter.

A second constructor gets passed an integer totalTime and an integer length and assigns them to the local variables of the same name. It also assigns a new PathItem of length zero to the list.

This class contains a toString method that returns a String representation of the Path by calling Arrays.toString on the list.

PathItem:

The PathItem class stores a traversed Track taken by a Train in its shortest path. Four variables are passed to and assigned by the constructor, integer stationID, integer trackID, integer arriveTime, and integer leaveTime.

The toString of this class is important as it determines what the final program output will look like. It returns a String that contains data such as the

Track the Train traveled, the Station it arrived at, the time it arrived, the time it left, and how long it had to wait.

Recursion:

Instead of implementing the depth-first search using real recursion and the native call stack, an iterative approach was needed because direct access to the current call stack allows for the easy copy and manipulation of the currently traversed/searched path in the graph. Recursion is the Stack data structure with semantic function names such that they behave in an iterative manner but are read one to one to their true recursive counterparts.

RecursionItem:

A custom implementation of a "stack pointer" used in the depth-first search. It contains all of the data in an individual layer of the stack for maintaining last position in search, both the Node being searched and the position in that Node's adjacency list.

Station:

Station is a generic class containing both an integer ID for a station and an ArrayList of type Integer with all the Stations that are connected to this Station. It has a single constructor that takes in and assigns an integer parameter to ID and creates a new instance of the Integer ArrayList.

Track:

Track is a generic class containing data on each edge in the graph. There are four variables, an integer ID, an integer weight, an integer stationIDs, and an Interval from the previously mentioned Interval class.

The constructor takes in the ID of the first Station, the ID of the second Station, a separate Track ID, and the weight. It assigns the ID/weight, initializes a new Interval with weight, and stores both Station IDs in the local stationIDs variable using the bitwise XOR operator (^).

The method, getOtherStationID, takes in an integer curID as a parameter, and returns the other Station that this Track is connected to.

Train:

Train is a generic class that stores data on each train including the starting Station, end Station, and ultimately, the ideal shortest path. Train also implements Comparable of type Train with its associated compareTo method as well as a toString method.

Like the previous classes, the constructor takes in a unique integer ID. It also gets passed an integer originID, integer destinationID, integer scheduleTime, integer Priority, and a base Path bestPath.

Path is discussed previously and the scheduleTime is the amount of time the train will wait from zero until it leaves the origin Station. The Train's priority will also determine when its path is calculated in relation to other Trains. The highest priority goes first.

The Overridden compareTo method first compares the priority of the trains and, if the priority is the same, compares the schedule time. If a Train's priority is insignificant, Trains will be calculated based on which Train leaves their respective origin Stations first.

The toString method returns the scheduleTime variable.

TrainDispatch:

The TrainDispatch class contains only one method, the main. The main method for this project prompts the user for three inputs. The first input required is the path to the StationInput file that will tell the program exactly how to build and draw the graph.(see requirement on pg ?) The Second input required is the Trains file that will give the system values representing trains start and end time as well as the trains start and end destinations. (see requirements on pg ?). The final input this method will ask for is if the user would like to run a base case algorithm or a modified (optimized) version of the scheduling system.

Upon having all required input, the main method creates an array for the three key components which include trains, stations, and tracks. These arrays are filled using the GraphInput class which is explained below. After filling theses arrays, the program calls the class GraphSearch and conducts a full search of the program to find the optimal schedule. After all data has been conducted the main class calls on GraphDisplay to provide a visual representation of the selected path.

TrainDispatchTest:

The TrainDispatchTest class is used to test the program. The program creates 100 test files using the TestFilePrinter class.

TestFilePrinter:

The TestFilePrinter Class is used to create 100 files that have random configurations. The files are currently set up to form a connected graph, but the program itself can handle multiple setups at once.

List of Algorithms/Data Structures

Algorithms

Pre-order traversal depth-first search on a weighted, undirected graph
Linear search on a singly linked list
Dual-pivot Quick Sort

Data Structures

Stack (custom implementation)
Linked List (custom implementation)
Adjacency List for Stations->Tracks (custom implementation)
Adjacency List for Tracks->Stations (custom implementation)
Graph (graph stream)
Stack
BitSet
Random
BufferedReader
FileReader
Scanner
IOException

Performance Analysis

Big O Notation:

$O(n^2 * k)$ where n is the number of Tracks and k is the number of Trains

Train Graphs:

As the number of trains increases, the calculated schedule times increase for both the base and optimized versions. For testing purposes, a standard train system containing twenty-five stations and forty connections was used with varying amounts of trains going from station to station. The number of trains started at 25 and increased in increments of 5 up to a total of 520 trains. The start and end points for each set of trains were also randomly generated. The results of the total train transit time (Figure 1.) and the average travel time (Figure 2.) can be seen below.

Track Graphs:

As the number of tracks increases, the calculated schedule times decrease for both the base and optimized versions. For testing purposes, a standard number of stations (25) and trains (25) was used with the tracks being randomly generated and increasing in number by increments of five. The first test will start with 40 tracks and increase by five until a total of 520 tracks exist between the 25 stations. Each test case ensures that there will always be at least one valid path between any two stations. The results of these test cases can be seen below (Figure 3. / Figure 4.) and depict the relation between the base and optimized versions.

Standard Deviation:

The standard deviation was tested using 100 test cases where the number of stations, tracks and trains are constant, but the train schedule as well as the arrangement and weight of each track are randomly generated. The tracks weights varied in a range between 1 and 30. All graphs of standard deviations can be seen in figures 5 and 6.

Graphs

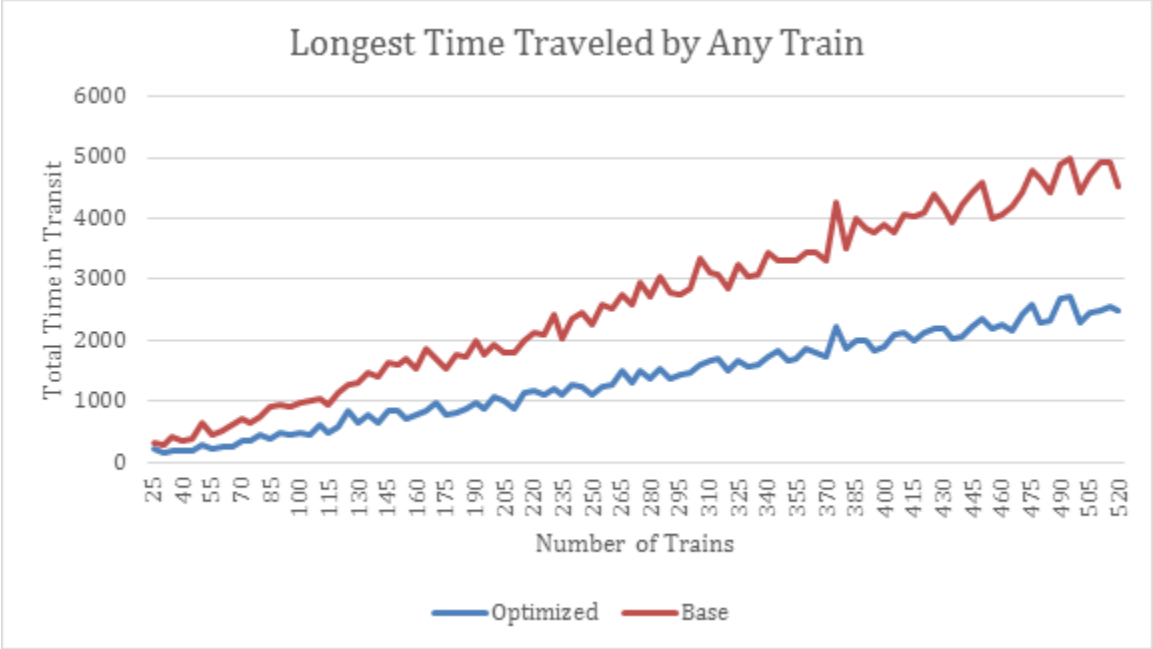

Figure 1.

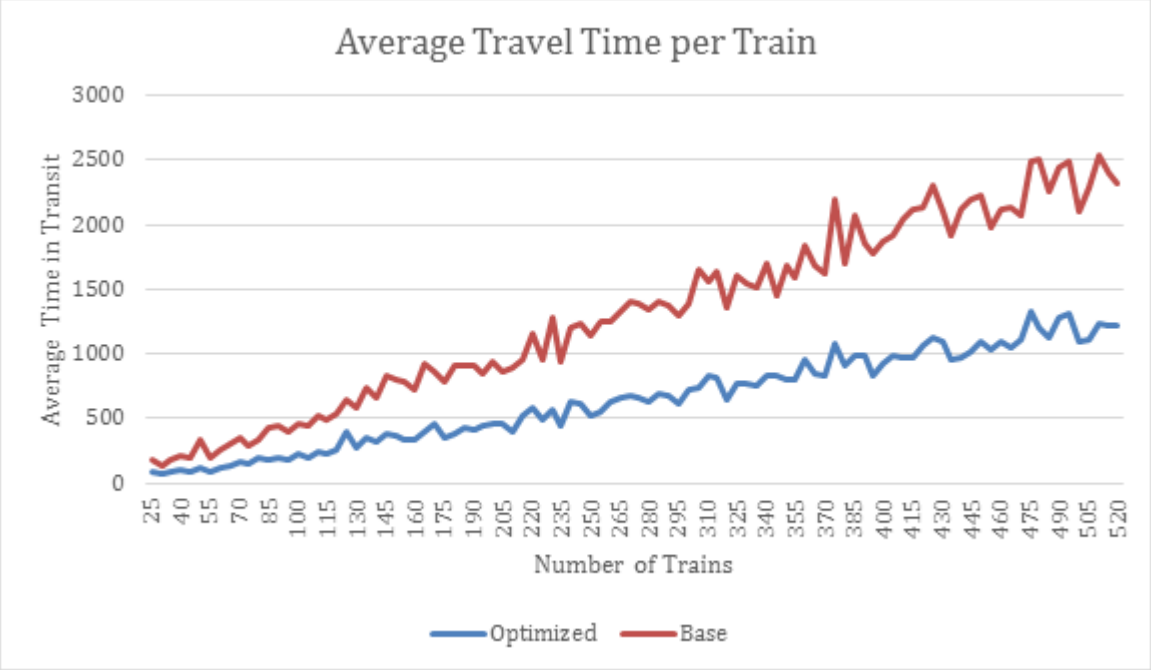

Figure 2.

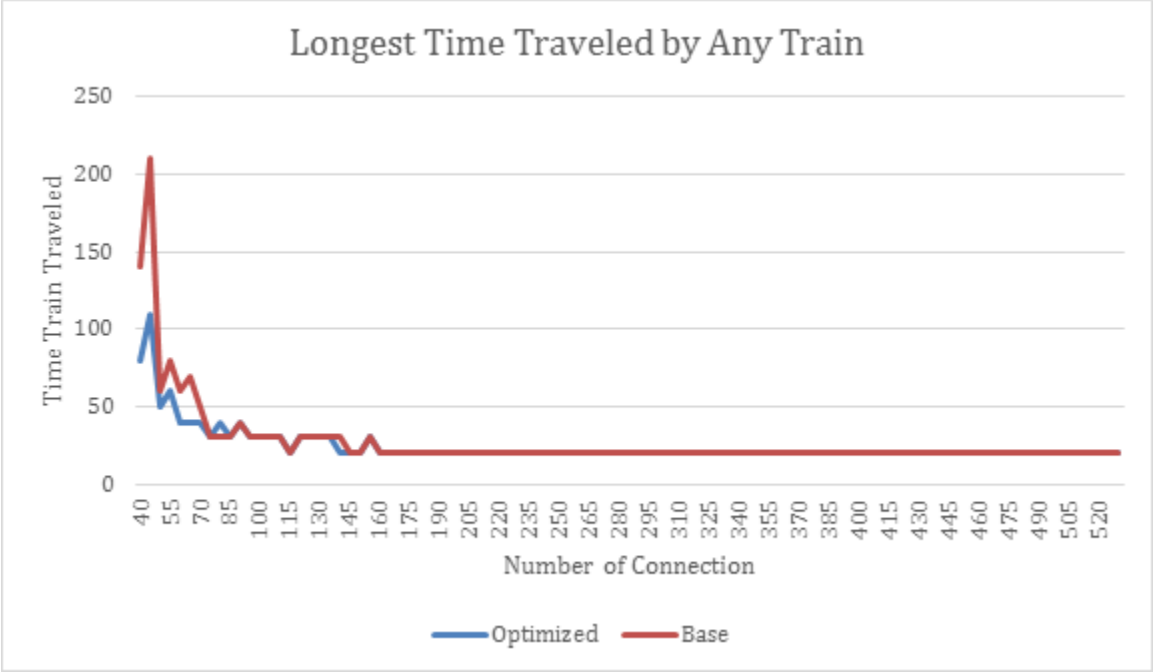

Figure 3.

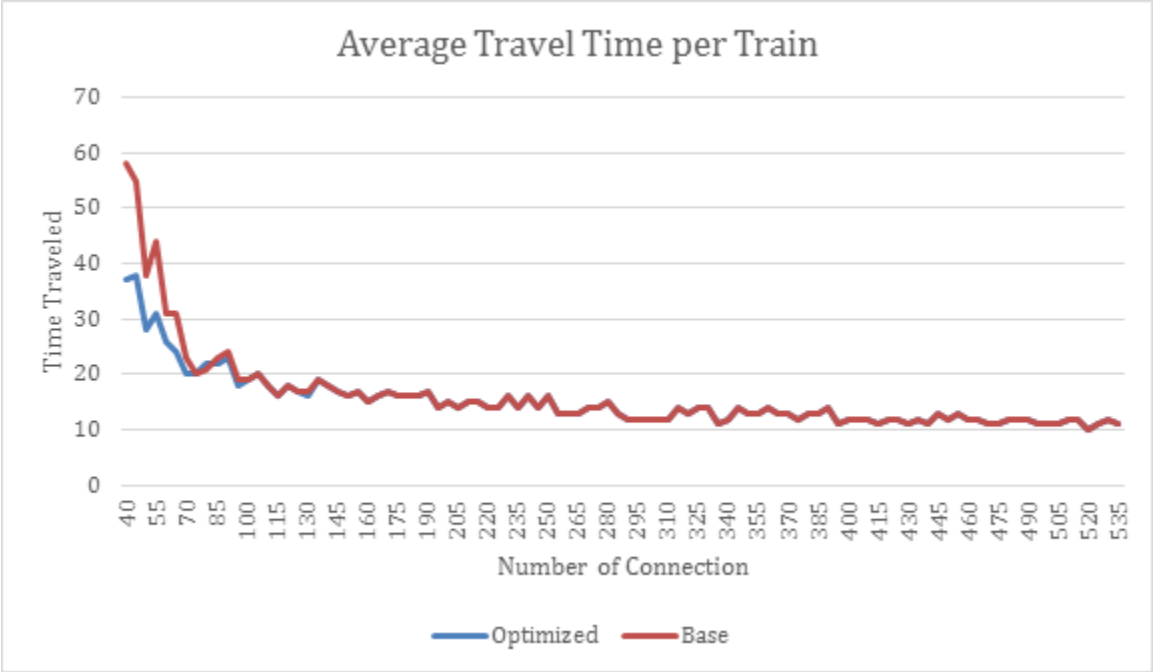

Figure 4.

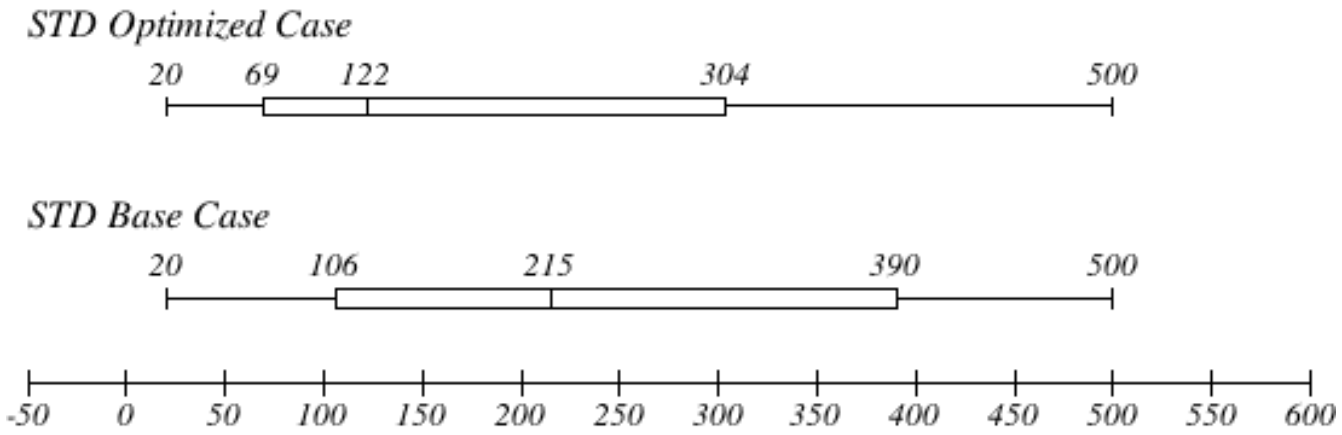

Figure 5. Overall Time

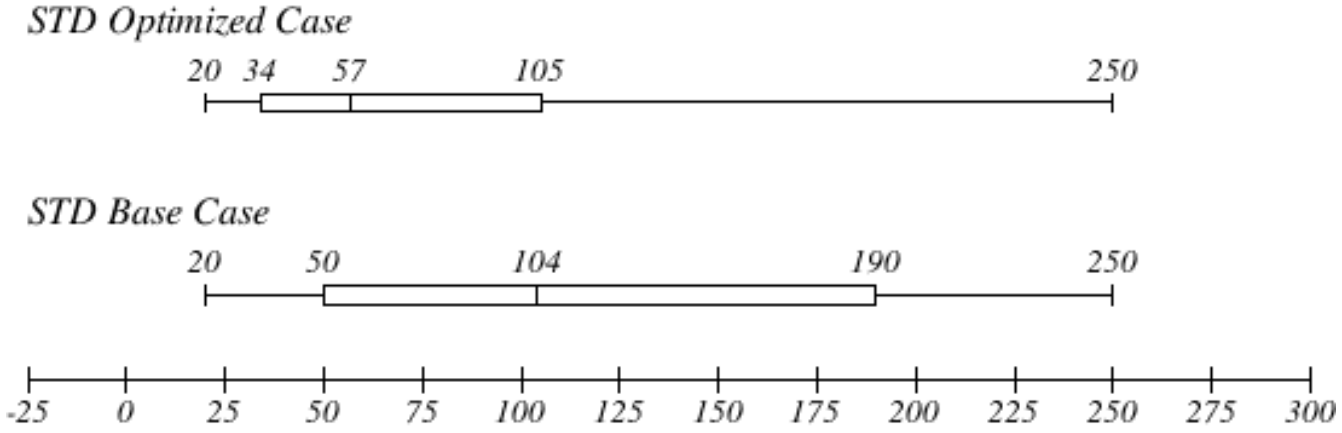

Figure 6. Average Time

Weekly Log

Week 1:

Team met on Wednesday, January 27, to discuss goals and expectations of team project and formulated a list of questions that were sent to Ivan Bogun for clarification.

Week 2:

Team met on Wednesday, February 3, to begin progress on the project. The team watched a youtube video by Tushar Roy titled " Dijkstra's Single Source Shortest Path Graph Algorithm". Upon completion of this video, the team walked through example of the dijkstra algorithm to gain a better understanding of how it works

Week 3:

Team met on Wednesday, February 10, to begin writing code and making actual progress on project. The team downloaded Tushar Roy's shortest path algorithm and parsed through it to gain further understanding on how it actually worked with a coding aspect.

Week 4:

Team decision to have no meeting this week (Algorithm Test Preparation)

Week 5:

Team met on Wednesday, February 24, to try and obtain a better grasp of the project requirements and create the base case. Short meeting with an agreement to rewatch tutorials and run through code on own time.

Week 6:

Team decision to cancel weekly meeting due to midterm test preparation

Week 7:

No Team Meeting (Spring Break)

Week 8:

Team met on Wednesday, March 16, after Algorithms test. The team put together a dijkstra's Algorithm that would provide the shortest time, but did not give the base case just yet.

Week 9:

Team met during lab time Thursday, March 24, and attempted to optimize the dijkstra algorithm to account for occupied and unoccupied tracks. It was also at this meeting that Zach showed the team a potential GUI we could use for our program, Graph Stream.

Week 10:

Team met during lab time Thursday, March 31, and achieved the base case algorithm for the train project and retro fitted the dijkstra system to fit our needs and decided to begin working on the gui and representation.

Week 11:

Charles had an epiphany about the train project and believed a depth first algorithm would be simpler to implement and prove to be a better fit for our goals. The team worked out a plan and was able to get both a DFS and Dijkstra version operational. The team decided to use the Dijkstra as the base case and the DFS as the Optimized.

Week 12:

The team met multiple times throughout the last week and kept hitting walls with the GUI. The library was finally able to be successfully implemented using the DFS algorithm, thanks to the better organization of data and mainly the better naming of tracks as they correlate with stations. The team decided to discontinue the dijkstra algorithm at this point due to its inability to display the proper track taken when multiple tracks existed between two points. The team recreated the base case with DFS and now can provide an option for the user as to what version they would like to implement.

References

GraphStream - GraphStream - A Dynamic Graph Library. (n.d.). Retrieved March 24, 2016, from http://graphstream-project.org/