Train Dispatch

“Better than the base case”

Micheal Lindenmeyer, Ibrahim Oyekan, Manan Davda

CSE 2010 | DATA STRUCTURE

Contents

The source code consists of nine classes in total, namely:

Main

The main class handles the primary functionality of the train dispatch system. It is the main structure that holds all the other classes together. The simulation of the day is controlled in the Main class. Our simulator runs keeping real time in mind. Each iteration of the loop represents one second of real time. In each iteration we control all the movements of the train.

For each train that has not reached its destination we first check if the train has a path assigned to it. For any trains that are waiting we need to assign a path using our Controller class. Using the controller class we generate the shortest path for each train, if a path exists. For the base case we only assign the train a path once. For the optimized case each time a train reaches a station we compute a new path. For the base case when a path is assigned we lock all the edges that are included in that path. For the optimized case we only lock the path that the train is scheduled to move next. Once a train has a path then we can start moving the train.

The main class communicates with the Train class to move each train. Once a train has reached a station we can unlock the last edge the train moved on. The base case only unlocks all the tracks once the train reaches the destination. Once a train, for the optimized case, reaches a station then we need to compute a new path. Since we move only one edge at a time we need to update all the locks on the grid and recalculate a shortest path with the new locks. This method allows us to have the least number of locked tracks. Tracks are only locked when the train is moving on the track. Once the train reaches its destination we remove the train from the list of active trains and we free the last track it moved.

Since there is a likelihood of deadlock using our implementation of dispatching, we organize the trains into a priority queue. The priority queue is organized by 5 different train statuses. The first is a train that has not been assigned a track, and is still at its starting station. This train is the least important. The second is a train that currently on the track. Trains that are on the track cannot free any tracks on this iteration so they are not very important. The third is a train that has been waiting for a long time. This situation is our deadlock situation. We want to move trains that have been waiting longer higher in the priority of the queue of trains. The fourth is a train that is not at its starting track and waiting for a piece of track to open for their path. The last train status is a train that has reached its final destination. These trains will always release a piece of track, making their movement very important for future path calculation.

The main class reads the train data and grid data from a file to generate the two different classes.

Grid

The grid class holds all the data associated with the grid. This class holds the data structure that holds the collection of edges. For our project we choose to use an adjacency list to hold all of our edges. The grid class also has a few functions for handling edges.

The grid has a function for adding edges to the grid. The grid can add all the adjacency for a single station at a time. Using ArrayList.add() we simply add the station to the end of the grid. For this reason it is important to add the stations in numerical order or they will not coincide with the station you want to travel to.

The grid also has two functions for setting the lock state of an edge. By knowing the start and end station we can access any edge in the grid. As a setback, we are unable to have multiple tracks from one station to another using this implementation. Once we find the edge we are interested we can choose to lock or unlock the edge. Since the grid is bidirectional we unlock /lock the edge and its reverse, to follow the rules of the example. The grid holds all the edges and can be accessed by the Main class only.

Edge

The edge class is a data structure that holds all the data for any edge in the grid. Each edge holds 4 values: start, end, locked state, and cost. The cost is a value that is generated from a file. Cost represent the distance, in miles, of the piece of track. The edge structure allows the grid to easily hold the lock state of any track.

Train

The train class is a data structure that holds all the data for a specific train. The train holds its path, distance traveled, speed, status, last station, and wait time. The train class also holds the function for moving the train on the track. Our program runs with the concept of real time. each iteration calls the train to move. Based on the trains speed it will move over a distance of the track. Once the train has traveled the length of the track it will return the edge it just completed for it to become available for the grid.

Controller

The controller class holds the functions for computing the Dijkstra Algorithm for the trian. The Algorithm computes for each train individually. The controller takes in the start station, end station, grid, and case. Any edge on the grid that is locked is not considered in the algorithm. If no path can be found for the train, due to locked edges, then it returns a null path. The controller uses a priority queue of Stations for computing the algorithm.

Priority Queue and Station

The priority queue data structure holds the list of stations. It also has three functions. The first is the queue function which adds the station to the queue. The dequeue function removes a station from the queue. The sort queue function is used to sort the stations by the cost. The costs are held in the Station class, along with the station number. The queue will allow the controller to access the shortest edge at any time when computing a path.

TestCase

The TestCase class generates a text file containing a randomly distributed sequence of 50 trains through the grid's 13 stations and a time span of 12 hours (43,200 seconds). Each line represents a train, with the first column being the starting station, the second column being the destination station, and the third column being the time in seconds.

Simulator

The Simulator class uses Standard draw to present the visualizations. Simulator class inputs two text files

1. Stations.txt (contains all the station x and y coordinates along with station numbers)
2. Stations_path.txt (contains all the tracks to be drawn in the format of x and y coordinates)

Then the visualization is drawn using points, picture and line of the standard draw class. Each track is originally green which means it is unlocked, as the path gets locked, the track color changes to Red.

Compiler instructions

```
javac trainDispatch\Main.java
```

java trainDispatch.Main (Number of Trains) (Time Frame in seconds) (Base case = 1; optimized case = 0)

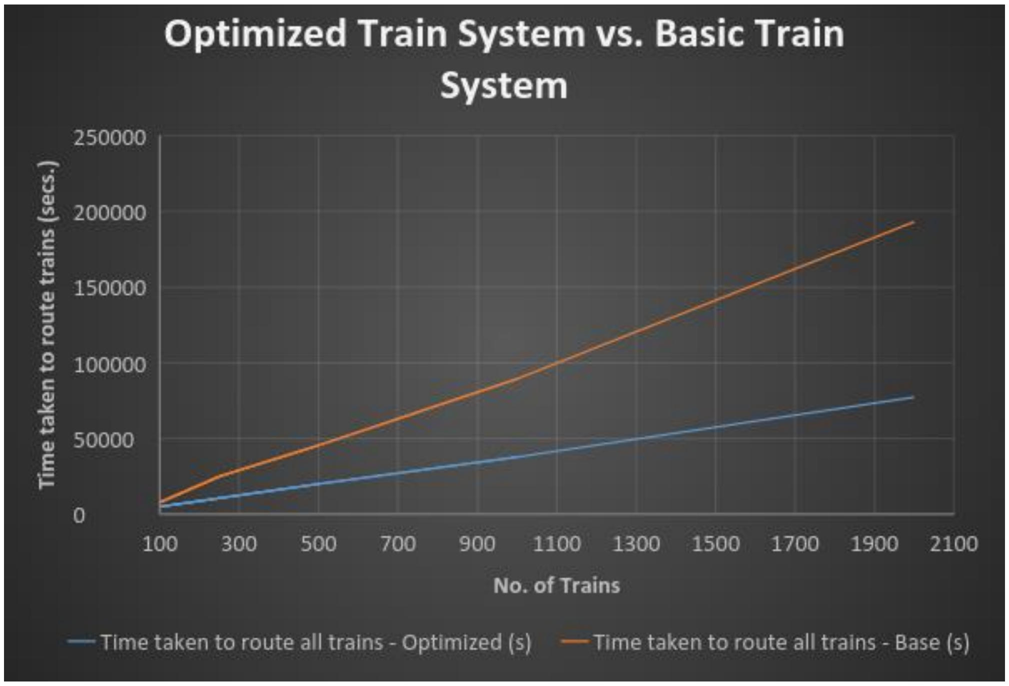
For example: java trainDispatch.Main 10 1000 1

Results / Data Analysis

Table 1: All trains dispatched at the same time (Time 0)

No. of Trains	Time taken to route all trains - Optimized (seconds)	Time taken to route all trains - Base (seconds)
100	4831	7747
250	10516	24873
500	19929	45436
1000	37694	89353
2000	77230	193151

Figure 1: Comparison of Optimized train system vs the Base case train system

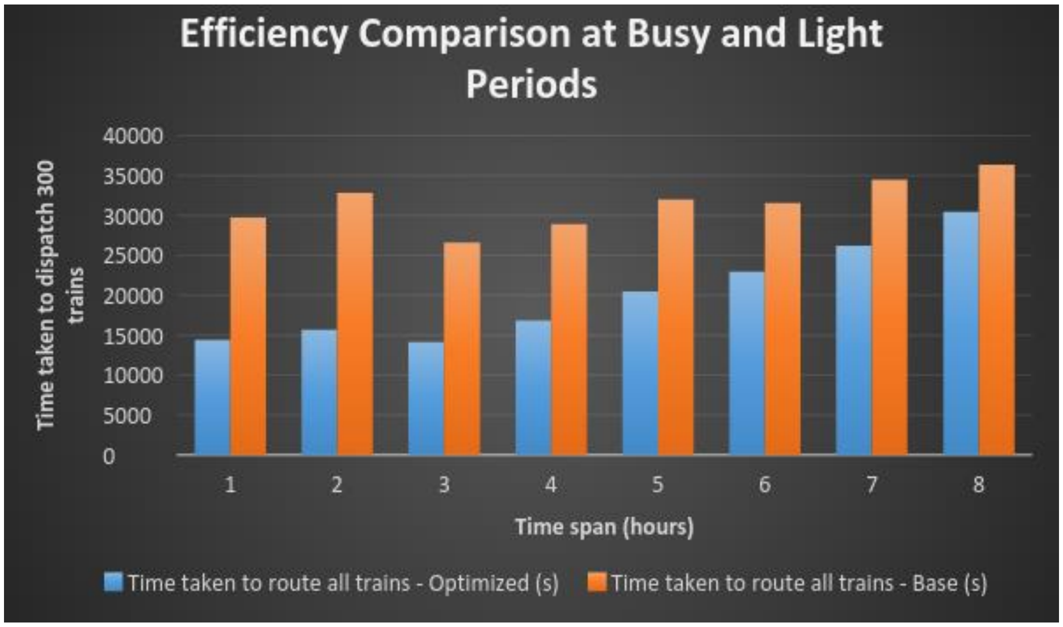


The graph above plots a distribution of both the Base and Optimized Train Systems against equal numbers of trains dispatched at the same time (time 0). Each individual train sequence was generated using the Test Case class, randomly assigned attributes, and run on both train systems. As shown in the graph above, the Optimized Train System constantly beats the Base Train System, with continually wider margins as the number of trains increase.

Table 2: 300 Trains dispatched over varying time lengths

300 trains		
Train dispatch time span (in hours)	Time taken to route all trains - Optimized (s)	Time taken to route all trains - Base (s)
1	14398	29730
2	15656	32813
3	14153	26593
4	16846	28902
5	20498	32009
6	22980	31537
7	26198	34466
8	30475	36345

Figure 2: Efficiency comparison between different time periods



The chart above represents the time taken to route 300 trains with equal attributes using both the Optimized and Base Train Systems. The Time Span values represent time ranges (in hours) within which each of the 300 trains are spawned. Smaller values simulate busier periods of activity while larger values simulate less active periods. As shown in the chart above, the Optimized Train System beats the Base System with wider margins during busier periods, and closer margins during less active periods.

Group Member Contribution

Manan Davda

Manan worked chiefly on the drawing simulator. He took charge making the drawing class simulate the locking and unlocking of the edges. He also helped develop the base and optimized cases for the final implementation.

Michael Lindenmayer

Michael worked primarily on the final algorithm implementation using the Dijkstra's Algorithm at each station. He also helped develop the base case and optimized case with the other two group members.

Ibrahim Oyekan

Ibrahim took charge in creating the randomized data generation. He made sure the data could be generated dynamically when running the program. He work alongside Michael and Manan to complete the base and optimized cases, as well.