Assignment 1 Part 2 Report

Julius Putra Tanu Setiaji (A0149787E), Chen Shaowei (A0110560Y)

25 October 2018

1 Program Design

This train simulation is implemented in OpenMPI. Primary design considerations are:

- Each edge is simulated by one process as required in the specifications.
- There is a master process responsible for the following:
  - Distribution of information (map, lines) to slave processes during initialisation.
  - Keeping track of train states (travelling/stationary).
  - Keeping track of station door open/close times.
  - Synchronising current time tick across all slaves.
- Slave processes hold queues of trains and send them to each other, and reports to master whenever train states change.

Assumptions

- Only one train can open its doors at each station at any one time, regardless of direction.
- Train stations have infinite capacity for waiting trains.
- Time units are discrete and can have no subdivisions
  - **Implication**: It is sufficient to store all time units as integers instead of floating point numbers
- Trains must open their doors for at least 1 unit of time.
  - **Implication**: We round every randomly generated door open time up to the nearest integer

2 Points to Note / Implementation Details

- Current simulation time needs to be shared across all processes. In addition, time can only be advanced after all processes have completed the actions to be done in the current tick.
  - **Implication**: Explicit synchronisation messages need to be sent between master and slaves before and after the advancement of time.

- Each slave process maintains two queues of trains. The first queue contains the trains that are waiting to enter the edge (*entry queue*). The second queue contains the trains that are waiting (arrived and/or door open) at the next station but have yet to close their doors, and the time at which they would close their doors (*exit queue*).
  - For convenience, the same queue implementation is used for both queues. The queue stores pairs where first element is the train and second element is the time to dequeue the train. *Entry queue*'s elements simply have garbage values for their second element.
- The logic for each slave process at each tick is as follows:
  1. If a current train is occupying the edge and it can leave at the current time, the process will query master to find out when this train will finish closing its doors. The process will then add this train and the time received into its *exit queue*. The edge will then be marked as available.
  2. If there are trains waiting to access the edge, the process will dequeue the first one and set it as the current train.
  3. If the slave process should spawn trains for any of the lines its edge falls on, it will do so, query master for their door closing time, and enqueue them (see step 1 above).
  4. If the train at the head of *exit queue* can close its doors at the **next** tick, the train is sent to the process containing the next edge that it should traverse.
     - Since adjacent edges would only receive these trains at step 6, trains sent at this step would not prematurely begin traversing the next edge.
     - This ensures that before step 2, every slave process already holds all trains that can begin traversing the edge if the edge is unoccupied in its *entry queue*.
  5. The slave process sends *"no more trains"* messages to all adjacent edges.
  6. The slave process waits for messages from all adjacent edges. If a train is sent, it is enqueued into the *entry queue*. When *"no more trains"* messages are received from all adjacent edges, this step is complete.
  7. The slave process informs master that it has completed all actions for the current tick.
  8. The slave process waits for an OpenMPI broadcast message that would either advance time or signal shutdown.
- The logic for the master process at each tick is as follows:
  1. The master process waits for messages from slave processes. If a request for next door close time is received, it computes the appropriate value (when the train door will be closed), updates station statistics and sends it back. If it receives a message that all actions for the current tick have been completed, it increments a counter. When all slave processes have reported completion, this step is complete.
  2. The master process prints the per-tick output, and broadcasts the next tick time.
  3. If the simulation has reached completion, instead of broadcasting the next tick time, master will broadcast the shutdown signal.
- The generic `MPI_Send` is used for slave-slave communication. Since messages are small, they are buffered and do not cause any deadlock. Deadlock does not occur when running all test cases in the lab. However, in the event that `MPI_Send` is not buffered and deadlock occurs, this can be resolved in the following ways:

- Non-blocking sends can be used for slave steps 4-5 and `MPI_Wait` called after slave step 6. This assumes that the non-blocking send will make process in the background even before `MPI_Wait` is called.
- Alternatively, a window can be created via `MPI_Win_create`. All information to be communicated in slave steps 4-5 will be written to its own window. When all slave processes have finished updating their own window (synchronised with `MPI_Barrier`), `MPI_Get` calls can be used in slave step 6 to obtain the relevant information.

3 Execution Time

3.1 Testcase Used

The Ruby script used to generate testcases for the previous OpenMP thread-based implementation is adapted to generate the various graphs to be used as input for the OpenMPI process-based implementation. To ensure fairness, we use the same number of threads as number of processes. In order to do this, the graph size thus changes for different number of processes since one process represents one edge. Since the graphs are generated by generating random Minimum Spanning Trees (MST), $e = v - 1$ where e is the number of edges and v is the number of vertices. Thus, we need to generate a graph with e edges, the graph must consist of $e + 1$ vertices. Note that each edge is unidirectional.

This time round, since we are generating smaller maps too, we reconfigured the thresholds of the graph generation to require 3 termini (vertices with degree 1) and that each line must have at least 2 stations. Maximum distance (maximum edge weight) between stations is 9.

The testcases all specified 10,000 time ticks. We ran testcases for 8, 16, 32, and 64 processes/threads. For the OpenMPI programme, we ran it on 8, 16 and 32 cores using a rankfile across 4 Xeon nodes in the lab. We ran each test twice, once with the per-tick status output enabled, and another time with it disabled.

Below you can find a sample input and visualisation of the adjacency matrix and the train lines for 8 edges/trains. All testcase input files and testcase generator script can all be found in the Appendix.

```
5
Sengkang,Bukit Panjang,Mattar,Damai,Botanic Gardens
0 0 0 0 6
0 0 0 8 9
0 0 0 0 5
0 8 0 0 0
6 9 5 0 0
0.8,0.6,0.5,0.8,1.0
Sengkang,Botanic Gardens,Bukit Panjang,Damai
Mattar,Botanic Gardens,Bukit Panjang,Damai
Sengkang,Botanic Gardens,Mattar
10000
3,3,2
```

Figure 1: Testcase for 8 edges/trains

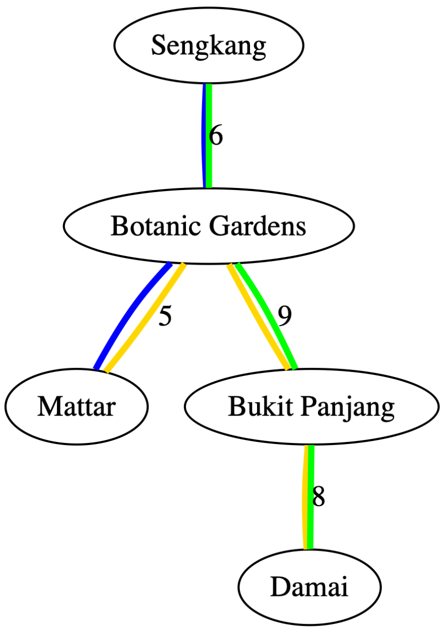

Figure 2: Map of the train lines for 8 edges/trains

3.2 Raw Data Collected

Legend:

- num_edge = Number of edges
- num_trains = Number of trains
- num_cores = Number of cores
- short = TRUE means no per-tick statistics is printed

num_edge / num_trains	num_cores	short	time
8	8	TRUE	1.152
8	8	FALSE	1.229
8	thread	TRUE	0.067
8	thread	FALSE	0.142
16	8	TRUE	1.466
16	8	FALSE	1.942
16	16	TRUE	10.216
16	16	FALSE	10.270
16	thread	TRUE	0.076
16	thread	FALSE	0.289
32	8	TRUE	2.004
32	8	FALSE	2.459
32	16	TRUE	12.244
32	16	FALSE	12.293
32	32	TRUE	15.611
32	32	FALSE	15.843
32	thread	TRUE	1.197
32	thread	FALSE	3.621
64	8	TRUE	3.640
64	8	FALSE	5.306
64	16	TRUE	382.949
64	16	FALSE	470.589
64	32	TRUE	39.227
64	32	FALSE	46.471
64	thread	TRUE	2.772
64	thread	FALSE	8.345

Table 1: Data collected

3.3 Comparison Summary

Number of cores	8 edges/trains	16 edges/trains	32 edges/trains	64 edges/trains
thread	0.067	0.076	1.197	2.772
8	1.152	1.466	2.004	3.64
16		10.216	12.244	382.949
32			15.611	39.227

Table 2: Summarised Comparison when per-tick statistics are not printed

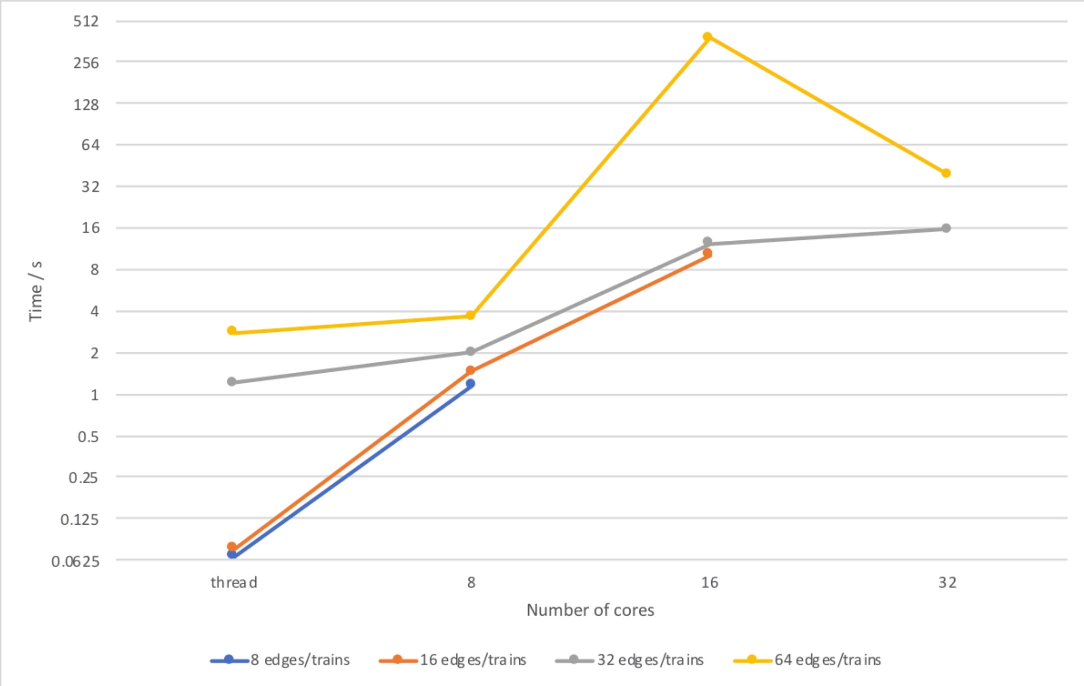

Figure 3: The comparison summary plotted (note the logarithmic vertical axis)

3.4 OpenMPI only

Using each of the testcase in the previous run, but with 22 trains on each line to ensure fairness. Note that the absolute time here might be slower than the previous run because there are other processes other than our train simulation running concurrently on the nodes.

num_edge	num_cores	time
8	8	1.30
16	8	1.46
16	16	10.32
32	8	2.11
32	16	129.27
32	32	40.65
64	8	3.78
64	16	415.07
64	32	264.41

Table 3: Results of running each test case with 22 trains on each line

3.5 Additional results

We ran both OpenMP and OpenMPI implementations with 1000 trains for each line using the map containing 64 edges. For fair comparison, we also disabled per-tick output. Results are shown below:

Configuration	time
65 processes locally	2.833
65 processes over 32 cores in 4 nodes	42.253
Multi-threaded	127.469

Table 4: Results of running map for 64 edges with 1000 trains on each line

4 Discussion

4.1 Behaviour of the multi-process OpenMPI Implementation

We compare the execution time when running on maps with different number of edges but with the same number of trains (22 trains on each line). From Table 3, the trend is that time is minimum when number of cores is 8, it increases to maximum when number of cores is 16, and then the time gets faster somewhat when number of cores is 32. Using 8 cores results in the shortest execution time; this is clearly due to all the cores being in the same node, resulting in OpenMPI using shared memory as message-passing mechanism. Thus, even when the number of edges increases, it is still much faster than when run with higher number of cores. The cost from lower throughput and higher latency message-passing mechanism via TCP outweighs the cost from higher context-switching due to running with lower core count.

With regards to 16 cores being slower than 32 cores, we have a hypothesis to explain this. Firstly, using 16 cores might be the slowest due to bottleneck in the TCP connections over the network. When using 16 cores, the processes are running across 2 nodes, hence the network link between these 2 nodes might get saturated and packets are buffered or even dropped on the router. When using 32 cores, the processes are running across 4 nodes, alleviating the burden on any single link, thus the time is faster for 32 cores than 16 cores. In addition, due to our use of blocking OpenMPI operations, processes may be blocked when waiting for communication from other sleeping processes, further increasing the time taken. This is alleviated when running with more cores, since fewer processes are asleep at any one time. Finally, the higher number of context switching when using 16 cores than using 32 cores could be another reason why it is slower.

4.2 Multi-process OpenMPI Implementation Vs. Multi-threaded OpenMP Implementation

We compare the multi-threaded OpenMP implementation against the multi-process OpenMPI implementation by using the same map and using the same number of trains with the per-tick statistics output disabled. From Table 2, it is apparent that the multi-threaded implementation is faster. This is mainly because threads are much cheaper to create and context switch. It should also be noted that the timing for 8 cores is quite close to the multi-threaded implementation. This is because it is only running on 1 node and OpenMPI uses shared memory as a message-passing mechanism for 2 processes running on the same node which is much faster than TCP, OpenMPI's default message-passing mechanism across nodes.

Nevertheless, despite the seemingly higher performance of the multi-threaded OpenMP implementation, there exists a case when the multi-process OpenMPI implementation will outperform the multi-threaded OpenMP implementation, namely when there is a large number of trains. As observed in Table 2, running both implementations on a map of 64 edges with 1000 trains on each line, the multi-process OpenMPI implementation is the fastest, both when running on only 1 node, communicating via the higher throughput and lower latency shared memory; or even across multiple nodes, communicating via lower throughput and higher latency TCP connections. This difference is likely because the incredibly high number of context switches required when running 3000 processes on a single machine outweighs the message passing cost.

Thus, our conclusion is that the multi-process OpenMPI implementation is better when a large number of trains need to be simulated but the number of edges is relatively small, since each process represents an edge in the graph. Meanwhile, the multi-threaded OpenMP implementation is better when a large map with a large number of edges need to be simulated but the number of trains is relatively small, since each thread represents a train. However, the multi-process OpenMPI Implementation is the more scalable solution since it can run across multiple nodes with little modification to the code. It can thus support even larger maps just by adding more nodes in the computation of the simulation.

Furthermore, it should be borne in mind that this train simulation as implemented is computationally trivial. As a result, communication costs take up a significant proportion of the execution time, causing single machine performance to be much better than distributed execution. One of the primary benefits of distributed computing via OpenMPI is that it allows the program to take advantage of execution units on all machines. This benefit cannot be observed in the case of a program that is computationally trivial.

5 Bonus

We chose to find the minimum number of trains per line required to maximise throughput in the network. We measured throughput by parsing the simulation output and counting the number of edge traversals by the trains. This optimisation was done on the example input map.

5.1 Notes

- Both total number of unweighted traversals and total number of weighted traversals were computed.
  - Every time a train traverses an edge, the number of unweighted traversals is incremented by 1 and the number of weighted traversals is incremented by the weight of the edge.
- For convenience, partial traversals that occur because the simulation ends before they can complete are counted. This has an insignificant impact on the result since the maximum difference from counting of partial traversals is 16 (number of edges in the graph) for unweighted traversals and 94 for weighted traversals while total traversals is in the order of ∼11,000 and ∼65,000 respectively.
- Each line has the same number of trains regardless of line metrics (number of stations, distance between stations, station popularity)
- Simulations were run for 10000 ticks.
- All lines have the same number of trains.

5.2 Results

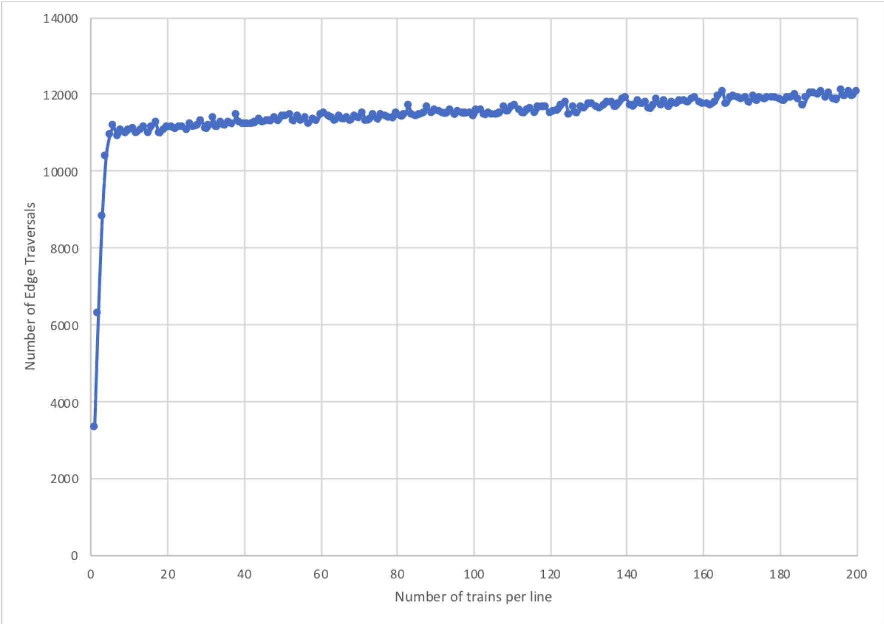

Figure 4: Scatter plot of number of edge traversals against number of trains per line

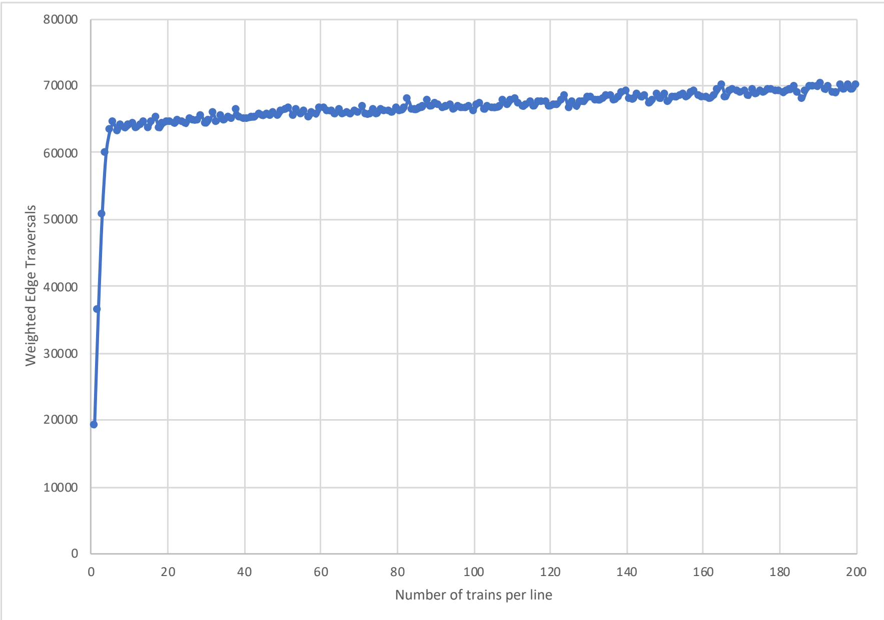

Figure 5: Scatter plot of weighted edge traversals against number of trains per line

As we can see from the graphs above, the law of diminishing returns kicks in quite early, and increasing the number of trains per line beyond 6 has minimal benefits when it comes to overall throughput in the train network. This is because of the bottleneck effect of having limited resources (tracks, door opening slots) in the train network.

Appendix A: Testcases and configuration files used

8 edges

```
5
Sengkang,Bukit Panjang,Mattar,Damai,Botanic Gardens
0 0 0 0 6
0 0 0 8 9
0 0 0 0 5
0 8 0 0 0
6 9 5 0 0
0.8,0.6,0.5,0.8,1.0
Sengkang,Botanic Gardens,Bukit Panjang,Damai
Mattar,Botanic Gardens,Bukit Panjang,Damai
Sengkang,Botanic Gardens,Mattar
10000
3,3,2
```

16 edges

```
9
Cashew,City Hall,Sixth Avenue,Fernvale,Kovan,Holland Village,Tuas Crescent,Sengkang,Orchard
0 0 9 0 0 0 0 0 0
0 0 0 0 0 0 0 0 9
9 0 0 7 0 1 7 0 0
0 0 7 0 0 0 0 6 0
0 0 0 0 0 3 0 0 0
0 0 1 0 3 0 0 0 1
0 0 7 0 0 0 0 0 0
0 0 0 6 0 0 0 0 0
0 9 0 0 0 1 0 0 0
0.6,0.4,0.3,0.5,0.9,1.0,0.7,1.0,0.7
City Hall,Orchard,Holland Village,Sixth Avenue,Tuas Crescent
City Hall,Orchard,Holland Village,Sixth Avenue,Fernvale,Sengkang
Cashew,Sixth Avenue,Fernvale,Sengkang
10000
5,5,6
```

32 edges

```
17
Jalan Besar,Buona Vista,Sixth Avenue,Punggol,Little India,Kovan,Farrer Park,Newton,Kupang,Phoenix,Kranji,Keat Hong,Bras Basah,MacPherson,Compassvale,Mattar,Samudera
0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0 2 0 0 0 0 0 0 0 6
0 0 0 0 0 9 0 0 0 0 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0 0 5 5 0 0 4 8 0
1 0 0 9 0 0 0 0 0 0 6 0 6 3 0 0 0
0 0 0 0 0 0 0 5 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 5 0 0 0 4 0 0 0 0 0 0
```

```
0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 2 0 0 0 0 0 0 0 0 9 0 0 0 0 0 0
0 0 0 0 5 6 0 4 0 9 0 0 0 0 0 0 0
0 0 0 0 5 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 6 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 3 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 4 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 8 0 0 0 0 0 0 0 0 0 0 0 0
0 0 6 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0.7,0.1,1.0,0.7,0.1,1.0,0.3,0.9,0.8,1.0,1.0,0.6,0.9,0.2,0.2,0.4,0.8
Bras Basah,Kovan,MacPherson
Jalan Besar,Kovan,Punggol
Punggol,Kovan,Bras Basah
10000
11,11,10
```

64 edges

```
33
Punggol Point,Fernvale,Orchard,Joo Koon,Ten Mile Junction,Bartley,Kallang,Fajar,Bayfront,Nibong,Sengkang,Toa Payoh,MacPherson,Esplanade,Newton,Boon Lay,Sam Kee,Sengkang,Kangkar,Little India,Tai Seng,Tiong Bahru,Soo Teck,Geylang Bahru,Buona Vista,Choa Chu Kang,Cheng Lim,Bishan,Botanic Gardens,Dhoby Ghaut,Kent Ridge,Changi Airport,Tampines
0 1 0 0 0 0 9 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 0 0 7 0 0 0 0 0 0 0 0 0 0 0 6 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 8 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 7 0 0 0 8 0 0 0 4 0 0 0 0 0 0 0 0 6 9 0 0 9 0 8 0 0 0 0 0 0 0 7
0 0 0 0 0 0 0 3 0 0 0 0 0 0 0 0 0 0 0 0 0 8 0 0 0 0 0 0 0 0 0 0 0
0 0 0 8 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 4 0 4 0 0
9 0 0 0 0 0 0 0 0 0 3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 6 0 0 0 0 0
0 0 0 0 3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 8 4 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 4 4 7 0 0 0 0 0 0 4 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 4 0 0 0 0 3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 6 0
0 0 0 0 0 0 0 0 0 0 0 4 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
2 0 0 0 0 0 0 0 2 0 0 7 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 6 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 6 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 6 0 0 0 0 0 0 0 0 0 0 0 0 0 6 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 9 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 6 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 5 0 0 0 0 3 0 0 0 0 0 0
0 0 0 0 8 0 0 0 0 0 0 4 0 0 0 0 0 0 0 0 5 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 9 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 5 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 6 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 8 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 5 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 3 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 6 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 4 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

```
0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 4 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 6 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 7 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0.3,0.6,0.6,1.0,0.4,0.5,0.3,0.4,1.0,0.4,0.2,1.0,0.2,0.2,0.3,0.8,0.8,0.4,0.8,0.8,0.9,0.6,0.4,0.9,0.7,0.7,0.4,0.8,0.1,0.5,0.3,0.7,0.5
Fajar,Ten Mile Junction,Tiong Bahru,Toa Payoh,Newton,Punggol Point,Fernvale,Joo Koon,Little India,Geylang Bahru
Sam Kee,MacPherson,Toa Payoh,Newton,Punggol Point,Fernvale,Joo Koon,Little India,Geylang Bahru
Kent Ridge,Bartley,Joo Koon,Fernvale,Punggol Point,Newton,Toa Payoh,MacPherson,Changi Airport
10000
21,21,22
```

Appendix B: Ruby code used to generate test cases

Below is the code listing of the ruby script used to generate test cases. Essentially, it does:

1. Create a random adjacency matrix with diagonal = 0
2. Find the MST of the random graph created
3. Ensure that there are enough vertices with degree = 1, else go back to step 1
4. Enumerate the 2-combinations of the vertices with degree = 1, and pick 3 randomly.
5. For each of the three 2-combinations, assign them to be the termini of each line.
6. Using breadth-first-search, find the path between the two vertices for each pair of termini.
7. Ensure that the path is long enough, else go back to step 4.

```ruby
require 'set'

MIN_NUM_TERMINI = 3
MIN_NUM_STATIONS_LINE = 2
```

```ruby
MRT_STATION_NAMES = ["Jurong East", "Bukit Batok", "Bukit Gombak", "Choa Chu Kang", "Yew Tee", "Kranji", "Marsiling", "Woodlands", "Admiralty", "Sembawang", "Yishun", "Khatib", "Yio Chu Kang", "Ang Mo Kio", "Bishan", "Braddell", "Toa Payoh", "Novena", "Newton", "Orchard", "Somerset", "Dhoby Ghaut", "City Hall", "Raffles Place", "Marina Bay", "Marina South Pier", "Pasir Ris", "Tampines", "Simei", "Tanah Merah", "Bedok", "Kembangan", "Eunos", "Paya Lebar", "Aljunied", "Kallang", "Lavender", "Bugis", "City Hall", "Raffles Place", "Tanjong Pagar", "Outram Park", "Tiong Bahru", "Redhill", "Queenstown", "Commonwealth", "Buona Vista", "Dover", "Clementi", "Jurong East", "Chinese Garden", "Lakeside", "Boon Lay", "Pioneer", "Joo Koon", "Gul Circle", "Tuas Crescent", "Tuas West Road", "Tuas Link", "Expo", "Changi Airport", "HarbourFront", "Outram Park", "Chinatown", "Clarke Quay", "Dhoby Ghaut", "Little India", "Farrer Park", "Boon Keng", "Potong Pasir", "Woodleigh", "Serangoon", "Kovan", "Hougang", "Buangkok", "Sengkang", "Punggol", "Dhoby Ghaut", "Bras Basah", "Esplanade", "Promenade", "Nicoll Highway", "Stadium", "Mountbatten", "Dakota", "Paya Lebar", "MacPherson", "Tai Seng", "Bartley", "Serangoon", "Lorong Chuan", "Bishan", "Marymount", "Caldecott", "Botanic Gardens", "Farrer Road", "Holland Village", "Buona Vista", "one-north", "Kent Ridge", "Haw Par Villa", "Pasir Panjang", "Labrador Park", "Telok Blangah", "HarbourFront", "Bayfront", "Marina Bay", "Bukit Panjang", "Cashew", "Hillview", "Beauty World", "King Albert Park", "Sixth Avenue", "Tan Kah Kee", "Botanic Gardens", "Stevens", "Newton", "Little India", "Rochor", "Bugis", "Promenade", "Bayfront", "Downtown", "Telok Ayer", "Chinatown", "Fort Canning", "Bencoolen", "Jalan Besar", "Bendemeer", "Geylang Bahru", "Mattar", "MacPherson", "Ubi", "Kaki Bukit", "Bedok North", "Bedok Reservoir", "Tampines West", "Tampines", "Tampines East", "Upper Changi", "Expo", "Choa Chu Kang", "South View", "Keat Hong", "Teck Whye", "Phoenix", "Bukit Panjang", "Petir", "Pending", "Bangkit", "Fajar", "Segar", "Jelapang", "Senja", "Ten Mile Junction", "Sengkang", "Compassvale", "Rumbia", "Bakau", "Kangkar", "Ranggung", "Cheng Lim", "Farmway", "Kupang", "Thanggam", "Fernvale", "Layar", "Tongkang", "Renjong", "Punggol", "Cove", "Meridian", "Coral Edge", "Riviera", "Kadaloor", "Oasis", "Damai", "Sam Kee", "Teck Lee", "Punggol Point", "Samudera", "Nibong", "Sumang", "Soo Teck"]

def generate_random_graph(s, max_weight)
  Array.new(s) { |i| Array.new(s) { |j| i == j ? 0 : rand(1..max_weight) } }
end

def print_graph(matrix)
  matrix.map { |row| row.join(' ') }.join("\n")
end

def prim(matrix)
  cost = Array.new(matrix.length, Float::INFINITY)
  parent = Array.new(matrix.length, nil)
  visited = Array.new(matrix.length, false)


  # start from the first vertex
  cost[0] = 0
  parent[0] = -1

  matrix.length.times do
    u = nil
    min_weight = Float::INFINITY

```

```ruby
    # Find unvisited vertex with minimum cost
    cost.zip(visited).each_with_index do |zipped, i|
      c, v = zipped
      if c < min_weight and !v
        min_weight = c
        u = i
      end
    end
    visited[u] = true

    matrix[u].zip(cost, visited).each_with_index do |zipped, i|
      m, c, v = zipped
      if m > 0 && !v && c > m
        cost[i] = m
        parent[i] = u
      end
    end
  end

  result = Array.new(matrix.length) { Array.new(matrix.length, 0) }

  (1...matrix.length).each do |i|
    result[i][parent[i]] = result[parent[i]][i] = matrix[i][parent[i]]
  end

  result
end

def bfs(matrix, termini)
  from, to = termini
  open_set = []
  closed_set = Set[]
  meta = {}

  root = from
  meta[root] = nil
  open_set.unshift(root)

  while !open_set.empty? do
    subtree_root = open_set.shift
    if subtree_root == to
      return construct_path(subtree_root, meta)
    end
    matrix[subtree_root].each_with_index.select { |w, i| w > 0 }.map { |x| x.last }.each
    ↪ do |child|
      next if closed_set.include?(child)
      if !open_set.include?(child)
        meta[child] = subtree_root
        open_set.unshift(child)
      end
    end
    closed_set.add(subtree_root)
  end
```

```ruby
end

def construct_path(state, meta)
  result = [state]
  while !meta[state].nil? do
    state = meta[state]
    result.append(state)
  end
  result.reverse
end

def permutate_sum(n)
  (0..n).to_a.flat_map { |i| (0..(n - i)).to_a.map { |j| [i, j, n - i - j] } }
end

def usage_message
  puts "Invalid args"
  puts "Usage: ruby test_case_generator.rb <max_weight> <num_tick>"
  exit 1
end

# Start of main
if ARGV.length != 2
  usage_message
end

max_weight, tick = ARGV.map { |a| a.to_i }

if max_weight <= 0
  usage_message
end


dir_name = "test-#{Time.now.strftime("%Y%m%d-%H%M")}"
Dir.mkdir(dir_name)

[[3, 3, 2], [5, 5, 6], [11, 11, 10], [21, 21, 22]].each do |trains|
  n = trains.inject(&:+)
  s = n / 2 + 1

  puts n, s

  primmed = nil
  termini = []

  while termini.length < MIN_NUM_TERMINI do
    graph = generate_random_graph(s, max_weight)
    primmed = prim(graph)
    termini = primmed
      .each_with_index.select do |row, i|
        row.reduce(0) { |acc, weight| acc += weight > 0 ? 1 : 0 } == 1
      end
      .map { |pair| pair.last }
```

```ruby
  end

  stations = MRT_STATION_NAMES.sample(s)
  popularities = Array.new(s) { rand(1..10) / 10.0 }

  green_line = []
  yellow_line = []
  blue_line = []
  while green_line.length < MIN_NUM_STATIONS_LINE || yellow_line.length < MIN_NUM_STATIONS_LINE || blue_line.length < MIN_NUM_STATIONS_LINE do
    green_termini, yellow_termini, blue_termini = termini.combination(2).to_a.sample(3)

    green_line = bfs(primmed, green_termini)
    yellow_line = bfs(primmed, yellow_termini)
    blue_line = bfs(primmed, blue_termini)
  end
  File.open("#{dir_name}/#{n}_#{trains.join("-")}", "w") do |f|
    f.puts s
    f.puts stations.join(",")
    f.puts print_graph(primmed)
    f.puts popularities.join(",")
    f.puts green_line.map { |s| stations[s] }.join(",")
    f.puts yellow_line.map { |s| stations[s] }.join(",")
    f.puts blue_line.map { |s| stations[s] }.join(",")
    f.puts tick
    f.puts trains.join(",")
  end
end
```

Appendix C: Python code used to count edge traversals

Below is the code listing of the python script used to count the number of edge traversals (for bonus).

```python
import os
from collections import defaultdict
import sys


def count():
    files = list(filter(lambda x: x.endswith(".out"), os.listdir(".")))
    for f in files:
        fn = f[:-4]
        count, duration = count_one(f)
        print("%s,%d,%d"%(fn, count, duration))


def count_one(f):
    train_history = defaultdict(list)
    contents = list(filter(lambda x: x, open(f, "r").readlines()))[:-3]
    for row in contents:
        parse_row(row, train_history)

    count = 0
```

```
    duration = 0

    for k in train_history.keys():
        for _, step, dur in train_history[k]:
            if type(step) == tuple and len(step) == 2:
                count += 1
                duration += dur

    return count, duration


def parse_input(contents):
    contents = [r.strip() for r in contents]
    num_stations = int(contents[0])
    station_names = contents[1].split(",")
    station_map = []
    for i in range(num_stations):
        r = contents[2+i]
        station_map.append([int(x) for x in r.split()])

    lines = []

    for i in range(3):
        line_names = contents[-5+i].split(",")
        line_idx = list(map(lambda x: station_names.index(x), line_names))
        lines.append(line_idx)

    num_trains = [int(x) for x in contents[-1].split(",")]

    start_train_ids = [0 for i in range(3)]
    start_train_ids[1] = num_trains[0]
    start_train_ids[2] = num_trains[1] + start_train_ids[1]

    return num_stations, station_map, lines, num_trains, start_train_ids


def parse_row(row, train_history):
    time, records = [c.strip() for c in row.split(":")]
    records = list(filter(lambda x: x, [c.strip()
                                        for c in records.split(",")]))
    for r in records:
        train, res = parse_record(r)
        hist = train_history[train]
        if len(hist) == 0 or hist[-1][1] != res:
            hist.append([time, res, 1])
        else:
            hist[-1][2] += 1


def parse_record(record):
    train, state = record.split("-", 1)

    if len(state.split("->")) == 2:
```

```python
        res = tuple(state.split("->"))
    else:
        res = state

    return train, res


if __name__ == "__main__":
    count()
```