**MST Report**

**Description of the problem:**

A minimum spanning tree (MST) or minimum weight spanning tree is a subset of the edges of a connected, edge-weighted undirected graph that connects all the vertices together, without any cycles and with the minimum possible total edge weight. That is, it is a spanning tree whose sum of edge weights is as small as possible.

**Sequential algorithm Details:**

The algorithm computes the shortest path starting from one source and then computes the cost of that path. After that it repeats the operation for all other sources. The short path is computed using the Dijkstra algorithm.

Dijkstra algorithm starts by adding the source in the priority queue then from that source it reaches the adjacent vertices and updates their distance cost from the source. In each step we remove the vertex with the least cost from the source. In the end we will get a minimum path from the specific source.

As a result we select the source which can reach all the vertices with the least path cost and this path will represent MST.

**Parallel algorithm Details:**

In the Parallel version of the algorithm, the minimum path for each source will be calculated in parallel. And the results will be saved to a shared object. Then we will wait for all the parallel jobs to finish the execution. After that we will find the minimum spanning tree.

**Implementation details of parallel algorithm:**

The work is divided into Runnable objects. Each runnable will be responsible for calculating the minimum path from one source to the other vertices and saving the result in a shared Concurrent HashMap because it is a thread safe object.

The Runnable objects are passed to a thread pool which contains a specific number of threads that can execute the Runnable objects. The thread pool is used to minify the resource usage of threads. By doing so a small number of threads can handle a big number of Runnable objects.

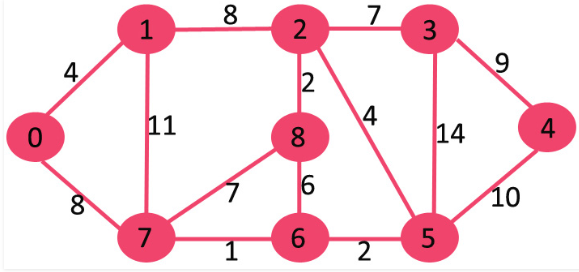
After that we will call get for each Future object to wait for its execution to be done. At the end we will get the minimum path from the Concurrent HashMap and print it using sequential code.

**Experimental results and comments:**

The data for testing the algorithms exist in the main directory of the project under the data folder.

* tinyEWG.txt contains 8 vertices and 16 edges
* mediumEWG.txt contains 250 vertices and 1,273 edges
* 1000EWG.txt contains 1,000 vertices and 8,433 edges
* 10000EWG.txt contains 10,000 vertices and 61,731 edges

Also there is a file tiny.txt containing an example with integer numbers to make it easier to understand the algorithm like the following:



The following table shows the execution time for different graphs and thread count.

|  | Sequential | Parallel  10 threads | Parallel  20 threads | Parallel  30 threads | Parallel  50 threads |
| --- | --- | --- | --- | --- | --- |
| tinyEWG | 47 ms | 62 ms | 62 ms | 62 ms | 62 ms |
| mediumEWG | 297 ms | 282 ms | 266 ms | 265 ms | 266 ms |
| 1000EWG | 1594 ms | 766 ms | 766 ms | 672 ms | 786 ms |
| 10000EWG | 83244 ms | 45195 ms | 44273 ms | 46783 ms | 52024 ms |

We can notice from the results above that sequential code is better for small data than parallel code.

Adding more threads will enhance the performance for bigger datasets but after a specific number of threads we cannot get better results. Instead we will start getting worse results because of the time needed for thread context switching.