An implementation of Parallelizing Bellman-Ford Algorithm

Problem statement

- Given a graph, Let G = (V, E) be a directed graph, |V| = n, |E| = m, let s be a distinguished vertex of the graph, and w be the value to the weight of each edge,
  which represents the distance between the two vertexes.
- Single source shortest path: The single source shortest path (SSSP) problem is that of computing, for a given source vertex s and a destination vertex t, the weight of a path that obtains the minimum weight among all the possible paths.

Structure

Graph - can be initialized from a file composed of multiple lines representing connection (A B
2). The last value being the cost.

Edge - the connection between nodes

```
public class Graph {
    int vertices;
    int start;
    List<Integer> dist = new ArrayList<>();
    List<Edge> edges = new ArrayList<>();

```

```
    // A class to represent a weighted edge in graph
    class Edge {
        int src, dest, weight;

        Edge(int src, int dest, int weight) {
            this.src = src;
            this.dest = dest;
            this.weight = weight;
        }
    }
}
```

Input

The graph is initialized using data from files.

The first line in the input files containes the number of vertices, of edges, and the staring point

The next N lines contain the infromation about the edges and the weights

```
6 6 1
1 2 803
2 3 233
3 4 158
4 5 134
5 6 774
6 2 871
```

Output

the program will output the distances from the start node to all other nodes

```
1 -> 0
2 -> 803
3 -> 1036
4 -> 1194
5 -> 1328
6 -> 2102
```

Sequential implementation

Bellman–Ford is based on the principle of relaxation, in which an approximation to the correct distance is gradually replaced by more accurate values until eventually reaching the optimum solution. The approximate distance to each vertex is always an overestimate of the true distance, and is replaced by the minimum of its old value and the length of a newly found path. Bellman–Ford algorithm simply relaxes all the edges, and does this |V|-1 times, where |V| is the number of vertices in the graph. In each of these repetitions, the number of vertices with correctly calculated distances grows, from which it follows that eventually all vertices will have their correct distances.
Bellman–Ford runs in O(|V| * |E|) time, where |V| and |E| are the number of vertices and edges respectively.

Pseudocode interpretaion

```
   // This implementation takes in a graph, represented as
   // lists of vertices and edges, and fills two arrays
   // (distance and predecessor) with shortest-path
   // (less cost/distance/metric) information

   // Step 1: initialize graph
   for each vertex v in vertices:
       distance[v] := inf             // At the beginning , a
ll vertices have a weight of infinity
       predecessor[v] := null         // And a null predecess
or

   distance[source] := 0              // The weight is zero a
t the source

   // Step 2: relax edges repeatedly
   for i from 1 to size(vertices)-1:
       for each edge (u, v) with weight w in edges:
           if distance[u] + w < distance[v]:
               distance[v] := distance[u] + w
               predecessor[v] := u

   // Step 3: check for negative-weight cycles
   for each edge (u, v) with weight w in edges:
       if distance[u] + w < distance[v]:
           error "Graph contains a negative-weight cycle"
```

Parallel implementation

The parallel implentation is the same as the sequential one but with in few changes. So during step 2 and 3 the edges are split equally between T threads. Each thread knows only about (E / T) edges, and during step 2 it makes |V - 1| iterations over the (E / T) edges updating the distances. In parallel the other threads do the same until they all finish.

The distances are represented using a Java Concurrent CopyOnWriteArrayList. Which makes sure we don’t run into any concurrency issues.

Testing

To test the efficency of the parallel program we are going to run it agains a graph containing 3353 vertices and 8870 edges, and compare it against the sequestial program

Graph	Sequential	1 thread	2 threads	3 threads	4 thre
45 vertices and 26 edges	1.168242	3.674947	3.613901	3.367118	2.383
3353 vertices and 8870 edges	190.456063	274.022689	199.36519	174.192022	166.247

Conclusion

As we can see from the testing, the parallel program performs worse on small graph as a lot of time is wasted in synchronizing the threads.
On larger graphs the parallel program performs better, but only when using fewer threads. As the number of threads increases so does the execution time, as the program loses a lot time trying to synchronize the threads