ANALYSIS OF MSTAVCI.JAVA & LOGIC OF THE CODE – Tamer Avcı

```
public List<SpanningTree> getMinimumSpanningTrees(Graph graph) {
		Set<Integer> visited = new HashSet<>();
		PriorityQueue<Edge> queue = new PriorityQueue<>();
		SpanningTree tree = new SpanningTree();
		List<SpanningTree> trees = new ArrayList<>();
		add(queue, visited, graph, 0);
		getMSTAux(graph, trees, queue, tree, visited);
		return trees;
	}
```

- We start off just like Prim's algorithm except for the list of trees that we are going to use to collect all the minimum spanning trees and a new auxiliary function getMSTAux that we call right after we add the target and every incoming edge to the PQ through add function.

```
		if(!trees.isEmpty() && trees.get(0).compareTo(tree) < 0)
			return;
		if(tree.size() + 1 == graph.size()) {
			trees.add(tree);
			return;
		}
		List<Edge> l = new ArrayList<>();
		for (Edge e: queue) {
			if(visited.contains(e.getSource()))
				l.add(e);
		}
		queue.removeAll(l);
		if(queue.isEmpty())
			return;
```

- getMSTAux begins by checking three end conditions. First one checks in case the list is not empty whether the total weight of the newly found tree is greater than the first MST or not. It is guaranteed by the prim that the first tree will be an MST. If the weight is greater then we don't add the newly found tree to the list.
- We add the tree to the list if the tree.size()+1 is equal to the graph size, which is again the same.
- Then we have a PQ clearance function which serves to eliminate duplicated directed edges. Otherwise we find duplicated trees.
- And finally the last end condition is if the queue is empty.

```
		Edge e;
		do
		{
			e = queue.poll();
			if(!visited.contains(e.getSource())) {
				PriorityQueue<Edge> q = new
				PriorityQueue<>(queue);
				Set<Integer> s = new HashSet<>(visited);
				SpanningTree t = new SpanningTree(tree);
				t.addEdge(e);
				add(q, s, graph, e.getSource());
```

```
                getMSTAux(graph, trees, q, t, s);
            }
        }
        while(!queue.isEmpty() && queue.peek().compareTo(e) == 
        0);
```

- The final part is the logic of the algorithm. We go into a do-while loop and poll the top element from the PQ just like Prim. If the set visited does not contain the source of the edge we create a new PQ, visited set, and a new ST to run a depth-first search algorithm.
- We add the edge that we have polled to the newly created t, SpanningTree, and call the function add with all these newly created PQ's, sets and ST. And then we recursively call getMSTAux again to find all the MSTs by recursively calling Prim.
- **The key part to the algorithm is that we run the do-while loop as long as the polled edge from the queue has the same minimum weight as the edge that is currently on top of the PQ. We check that condition with queue.peek(). This allows us to backtrack and find the other MSTs using edges that have the same minimum weight.**