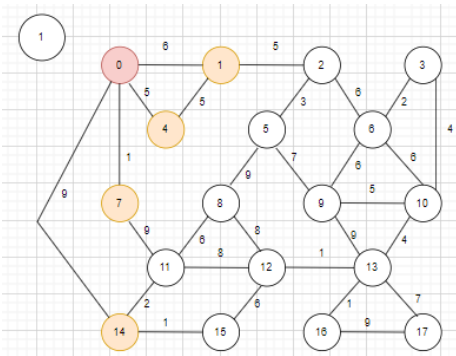
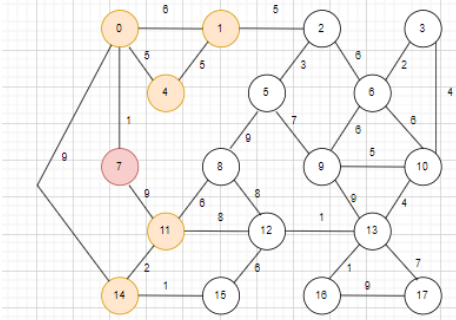
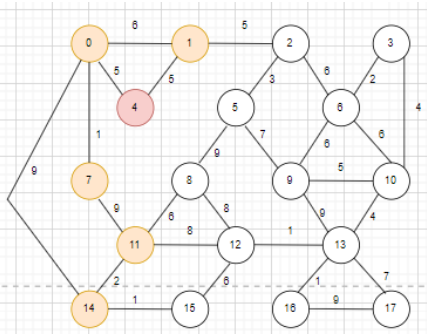


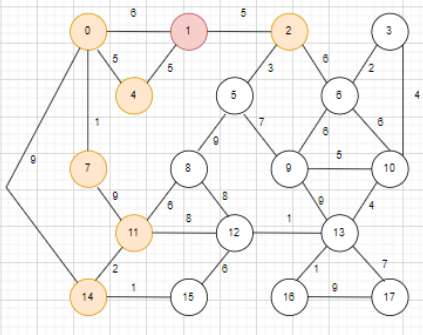
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
dist		6			4			1							9			
path		0			0			0							0			



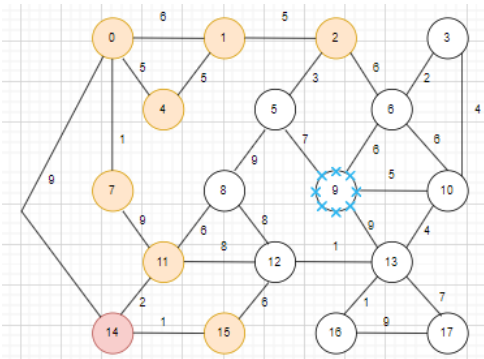
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
dist		6			4			1				10			9			
path		0			0			0				7			0			



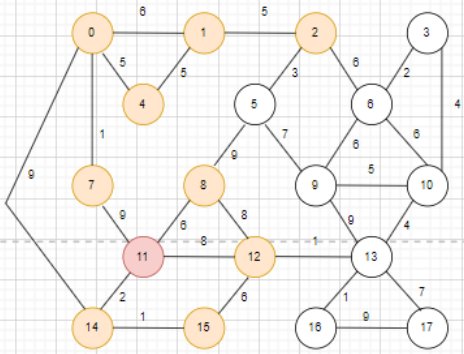
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
dist		6			4			1				10			9			
path		0			0			0				7			0			



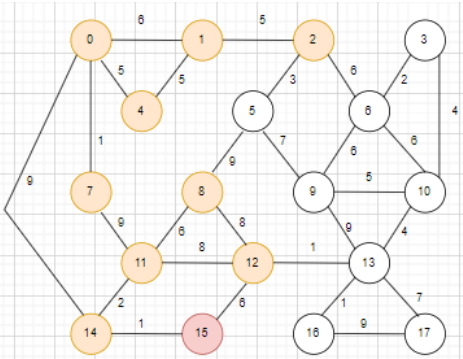
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
dist		6	11		4			1				10			9			
path		0	1		0			0				7			0			



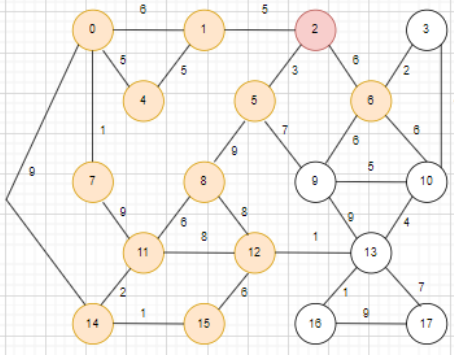
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
dist		6	11		4			1				10			9	10		
path		0	1		0			0				7			0	14		



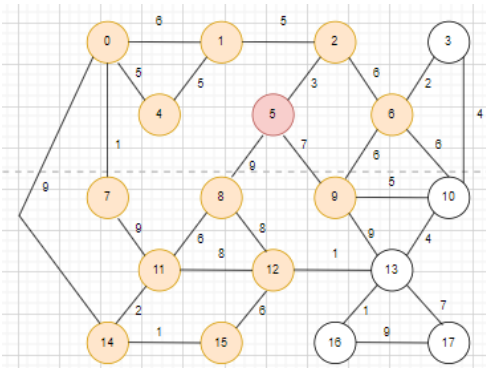
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
dist		6	11		4			1	16			10	18		9	10		
path		0	1		0			0	11			7	11		0	14		



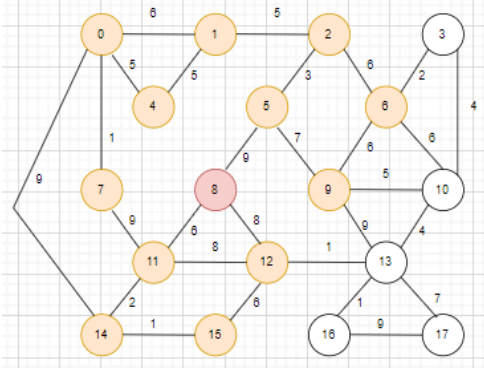
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
dist		6	11		4			1	16			10	16		9	10		
path		0	1		0			0	11			7	11		0	14		



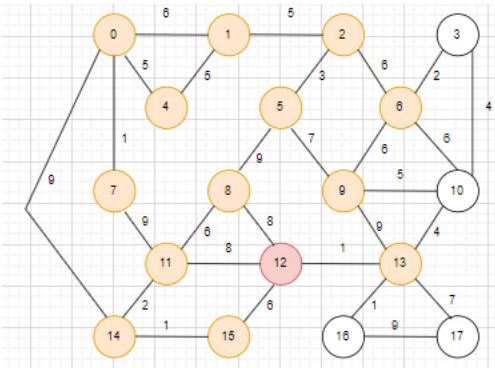
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
dist		6	11		4	14	17	1	16			10	16		9	10		
path		0	1		0	2	2	0	11			7	15		0	14		



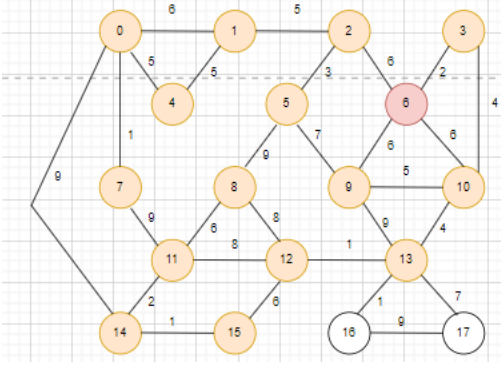
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
dist		6	11		4	14	17	1	16	21		10	16		9	10		
path		0	1		0	2	2	0	11	5		7	15		0	14		



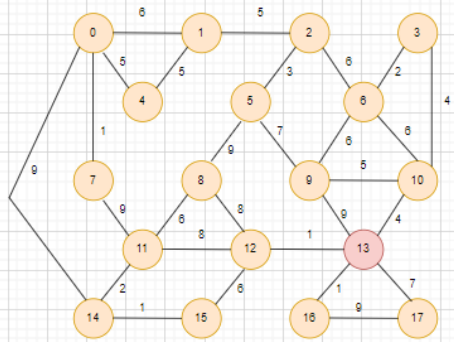
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
dist		6	11		4	14	17	1	16	21		10	16		9	10		
path		0	1		0	2	2	0	11	5		7	15		0	14		



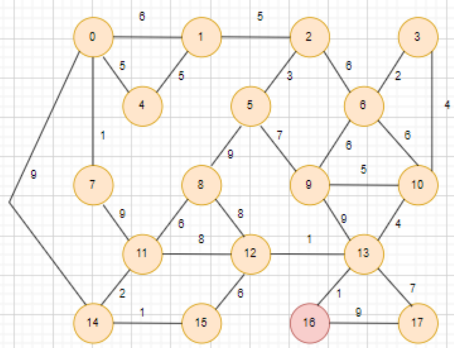
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
dist		6	11		4	14	17	1	16	21		10	16	17	9	10		
path		0	1		0	2	2	0	11	5		7	15	12	0	14		



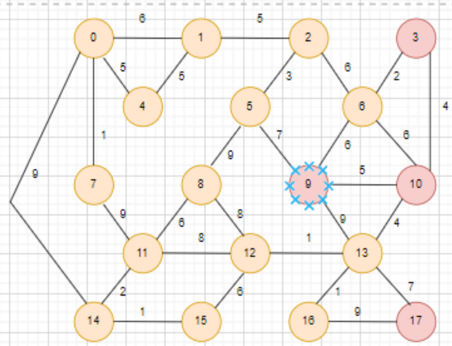
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
dist		6	11	19	4	14	17	1	16	21	23	10	16	17	9	10		
path		0	1	6	0	2	2	0	11	5	6	7	15	12	0	14		



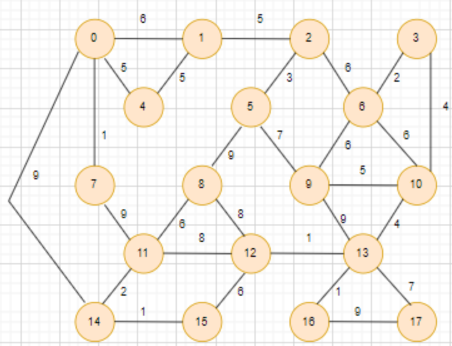
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
dist		6	11	19	4	14	17	1	16	21	23	10	16	17	9	10	18	24
path		0	1	6	0	2	2	0	11	5	6	7	15	12	0	14	13	13



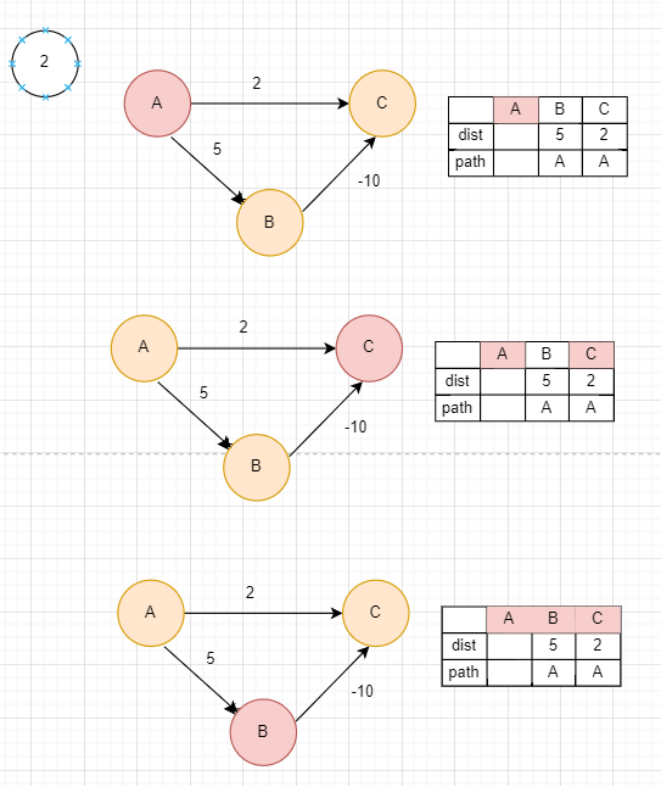
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
dist		6	11	19	4	14	17	1	16	21	23	10	16	17	9	10	18	24
path		0	1	6	0	2	2	0	11	5	6	7	15	12	0	14	13	13



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
dist		6	11	19	4	14	17	1	16	21	23	10	16	17	9	10	18	24
path		0	1	6	0	2	2	0	11	5	6	7	15	12	0	14	13	13



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
dist		6	11	19	4	14	17	1	16	21	23	10	16	17	9	10	18	24
path		0	1	6	0	2	2	0	11	5	6	7	15	12	0	14	13	13



If you notice, rather than visiting C as A->C, we could have visited C as A->B->C. In this way, the distance would have been 5+(-10) = -5. Therefore, it is clear that Dijkstra's algorithm has failed to calculate the correct answer.

As:

The distance of node A from A will be 0; therefore, dis[A] = 0.

Also, in Dijkstra's algorithm, we set the initial distances of all the nodes other than the source to infinity, therefore dist[B]= infinity and dist[C] = infinity.

The priority_queue(PQ) will be containing the pairs consisting of {distance of a node from A, node}.

Initially, pq = [(0,A), (infinity,B), (infinity, C)]

We pop out the pair (0,A) from the priority queue because it's on the top and it has the smallest distance, after which, pq = [(infinity,B), (infinity,C)]. A has a directed edge to B and C, therefore, we update the distance of B and C. dist[B] = 0+5 = 5 and dist[C] = 0+2 = 2. Now, pq = [(2,C),(5,B)].

We pop out the pair (2,C), now PQ = [(5,B)]. C doesn't have any edge to any node. Thus, we move ahead.

Now, we pop out the last pair (5, B). Note that B has an edge to C with weight -10, but since C is not in PQ, it means the distance of C is already calculated. Thus we will not update the distance of C.

Now, the queue is empty and we have calculated the shortest distance of each node. dist[A]=0, dist[B] = 5 and dist[C] = 2.

It happens because, in each iteration, the algorithm only updates the answer for the nodes in the queue. So, Dijkstra's algorithm does not reconsider a node once it marks it as visited even if a shorter path exists than the previous one.

Hence, Dijkstra's algorithm fails in graphs with negative edge weights.

3. Yes, Prim's and Kruskal's algorithms produce a MST for an undirected graph with weights that can be either positive or negative. Both of these algorithms are "greedy" algorithms and the reason why the greedy approaches to finding the MST work is that you can always get a better ST if there is an unused edge that has a lower weight than any edge on the cycle that would be created by adding it on the ST. (You just add the unused edge, and remove one of the higher-weight edges on that cycle.) This principle holds for both positive and negative edge weights.

4. Yes, Prim and Kruskal's algorithm work for this problem as Prim's and Kruskal's algorithms are Greedy algorithms which can be used to find the Minimum Spanning Tree (MST) as well as the Maximum Spanning Tree of a Graph.