Supervision work
Supervision 15

Tudor Avram
Homerton College - tma33@cam.ac.uk

1st March 2016

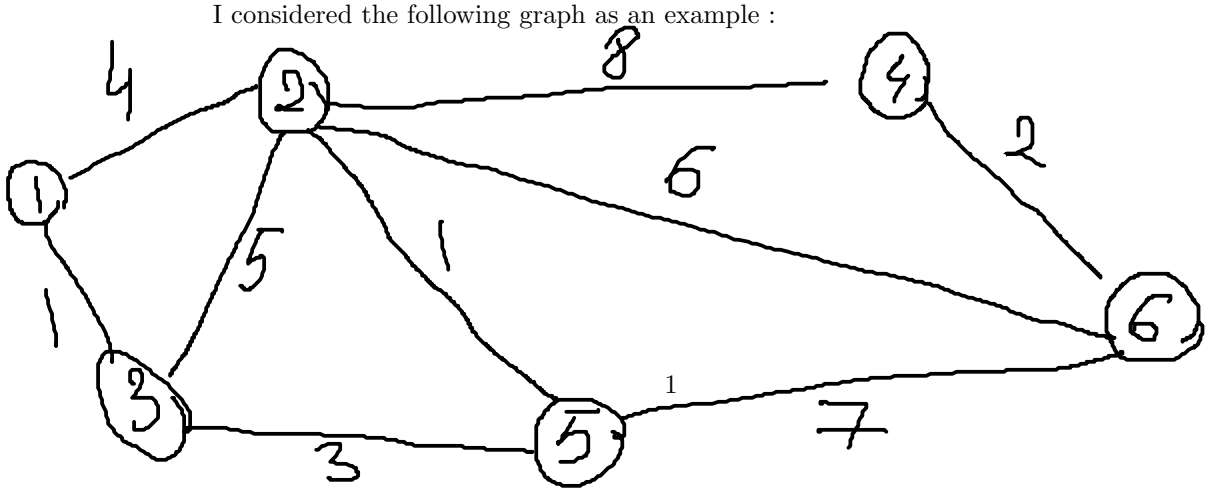
1 MSTs

1.1 Exercise 1

If all the edges are positive and distinct, there is only one possible MST. We can prove this by assuming the contradiction and show that it is not possible. So, we assume that we can have more than one MST in our graph. In other words, this means that there is at least one edge that, removed, can be replaced and we would still have an MST. Let's assume that we removed one edge of cost C. Now we no longer have an MST, because we didn't include all the nodes in the graph. We try to replace the extracted edge. Because all the edges are distinct, we have two possible cases for the cost c of the replacement :
(i) $c < C$ Well, if this would be the case, then our original tree would not have been an MST, because there was more cost-efficient tree possibility. Therefore, this case is not actually possible. to occur.
(ii) $c > C$ In this case, we would increase the total cost of our tree, so it wouldn't be an MST, because the total cost would increase.
In conclusion, we can deduce that, as long as the edges have distinct costs, we can be sure that there is only one possible MST.

1.2 Exercise 2

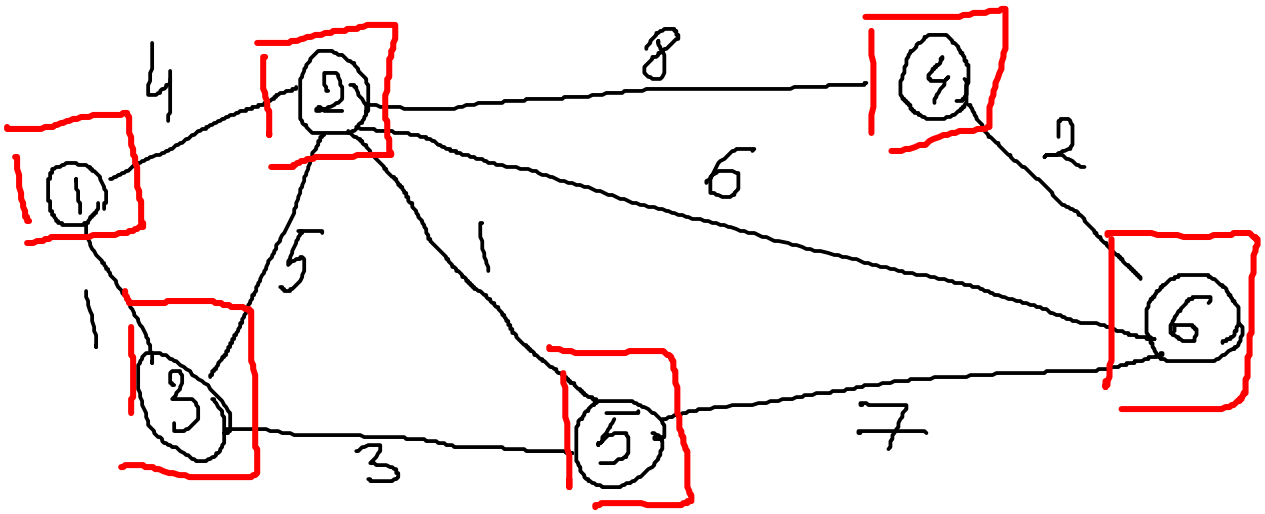
I considered the following graph as an example :



(i) Kruskall algorithm

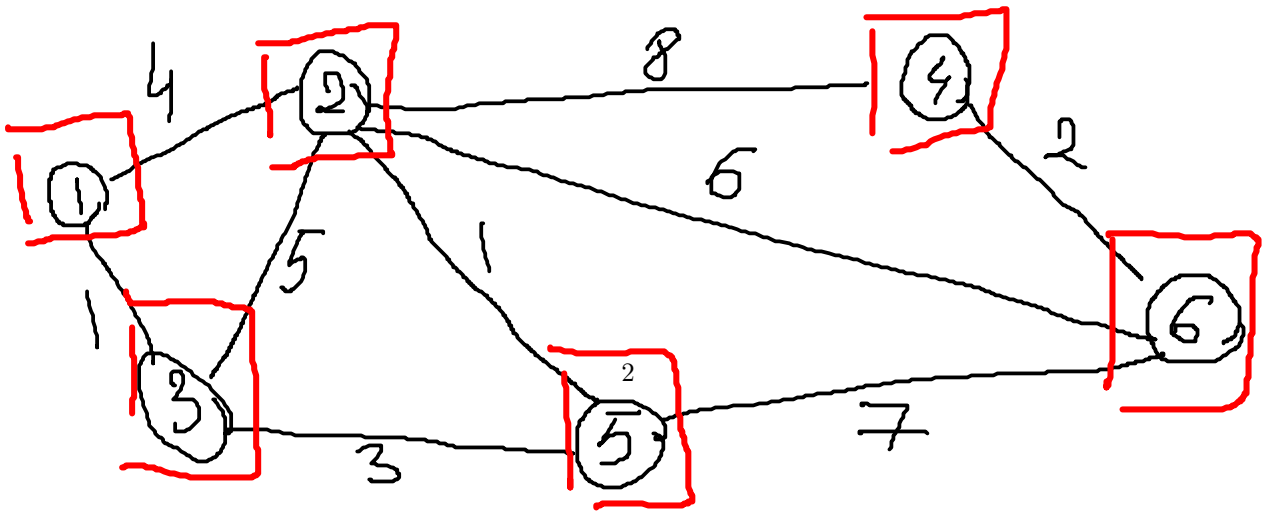
Step 1 : I create a separate set for every node and I insert the list of edges in ascending order.

E = [(1,3),(2,5),(4,6),(3,5),(1,2),(2,6),(5,7),(2,4)]

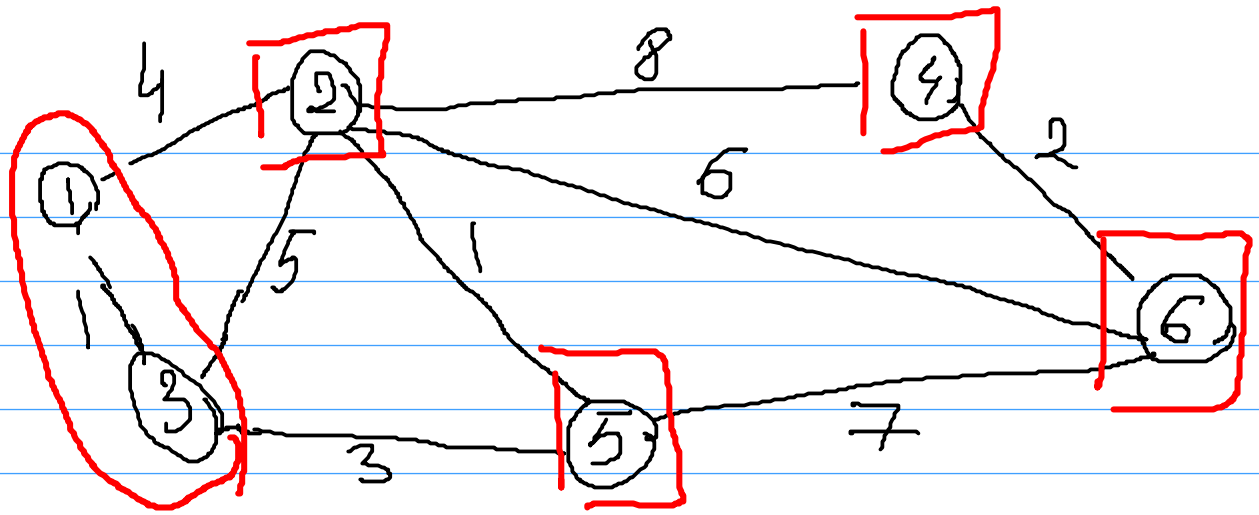


E = [(1,3),(2,5),(4,6),(3,5),(1,2),(2,6),(5,7),(2,4)]

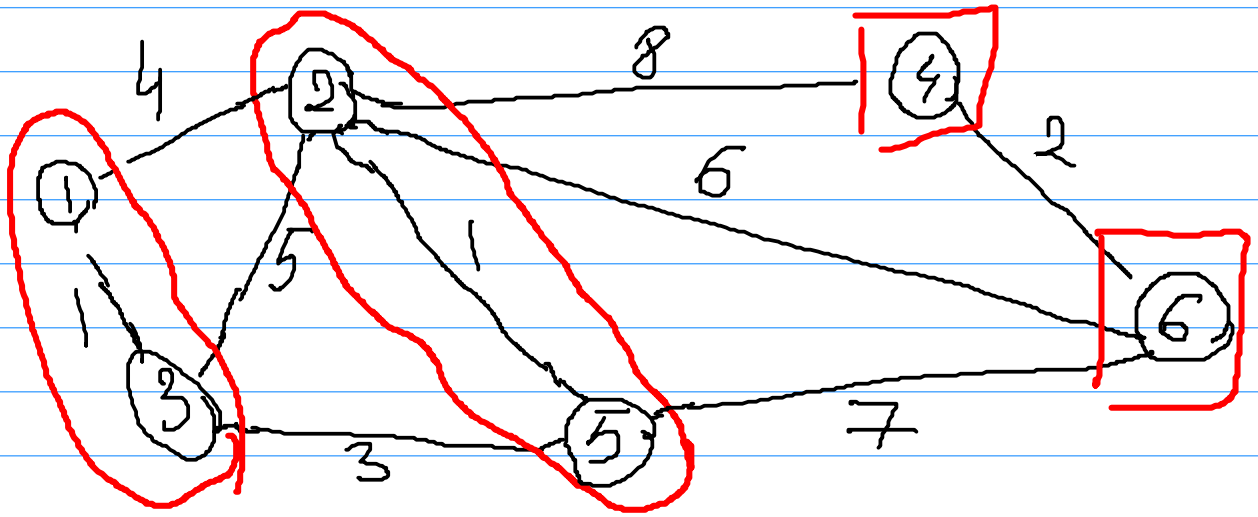
Step 2 : We go through the set of edges and try to merge the disjointed sets



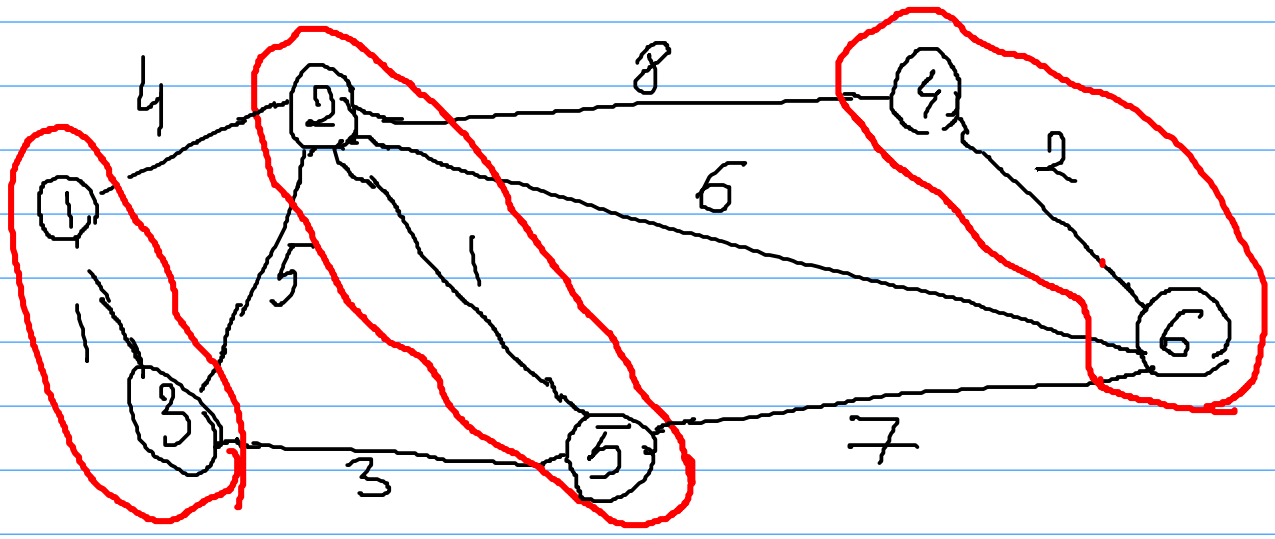
1 and 3 belong to different sets, so I can merge them



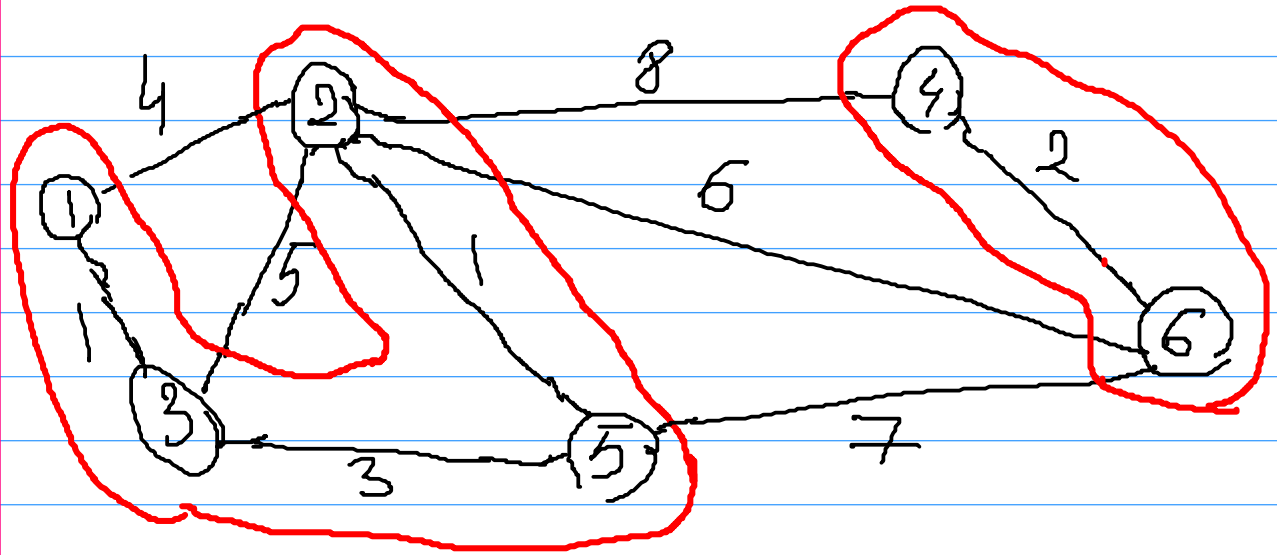
Step 3 : now we check for the next vertice in the set : (2,5) The 2 nodes belong to different sets, so we can merge them too.



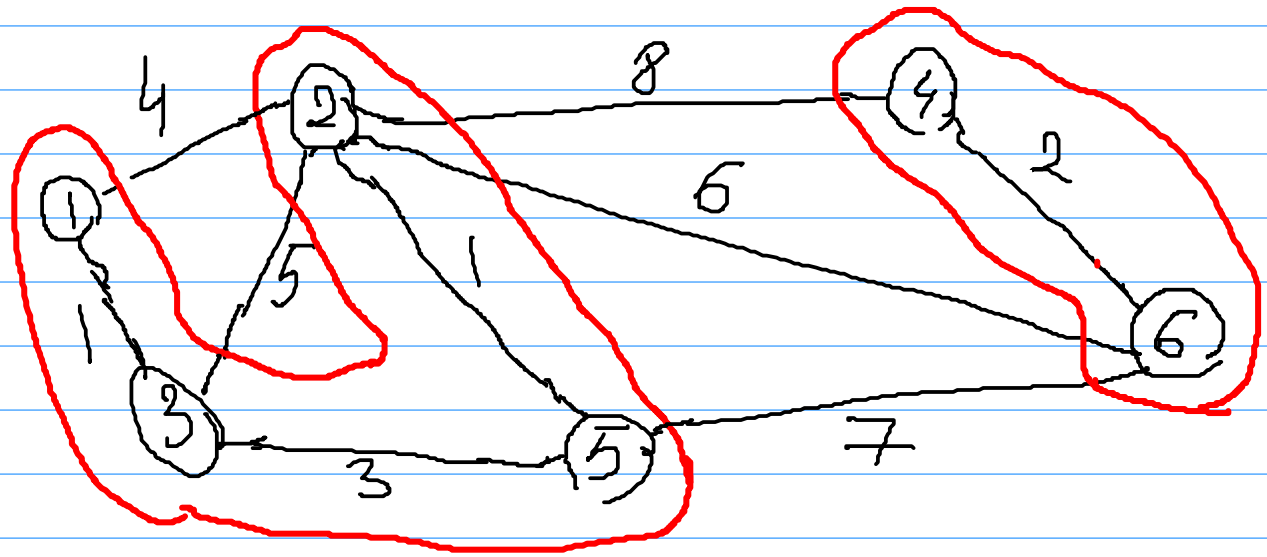
Step 4 : now we check for the next vertice in the set : (4,6) The 2 nodes belong to different sets, so we can merge them too.



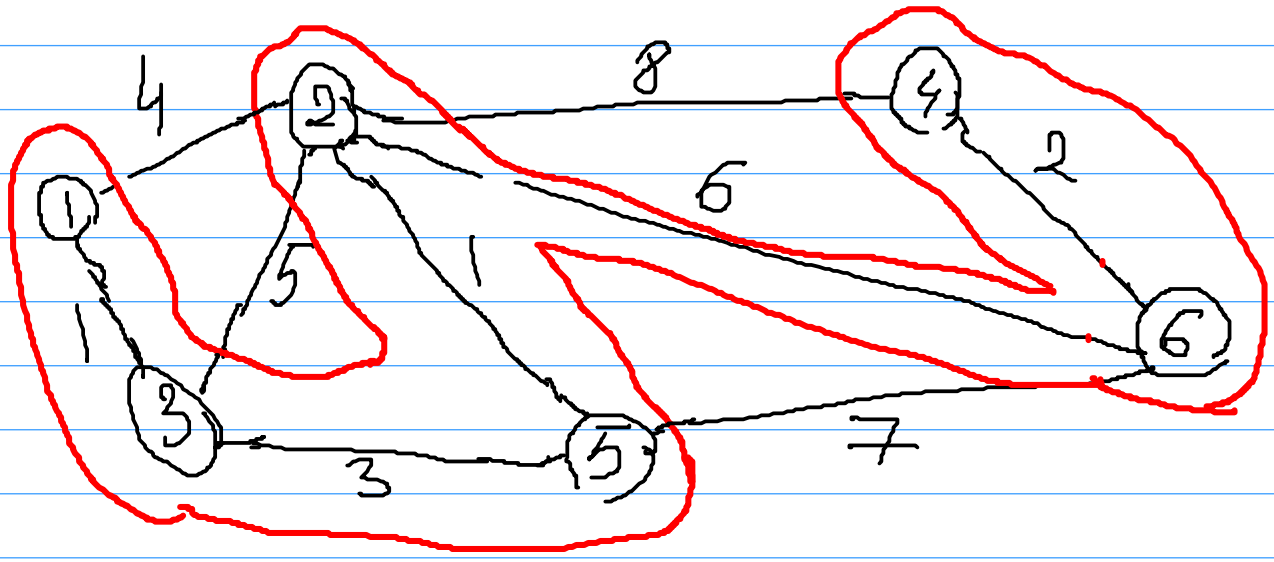
Step 5 : now we check for the next vertice in the set : (3,5) The 2 nodes belong to different sets, so we can merge them too.



Step 6 : now we check for the next vertice in the set : (1,2) The 2 nodes belong to the same set, so we don't merge them, because we will end up with a cycle.



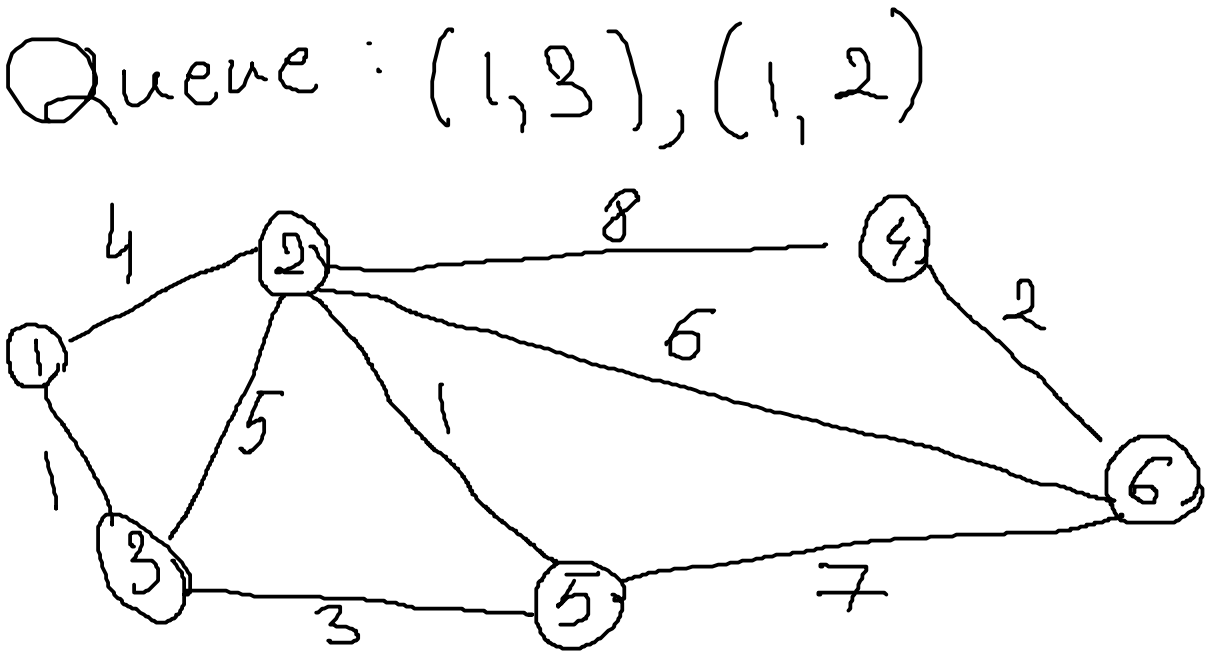
Step 7 : now we check for the next vertice in the set : (2,6) The 2 nodes belong to different sets, so we can merge them too.



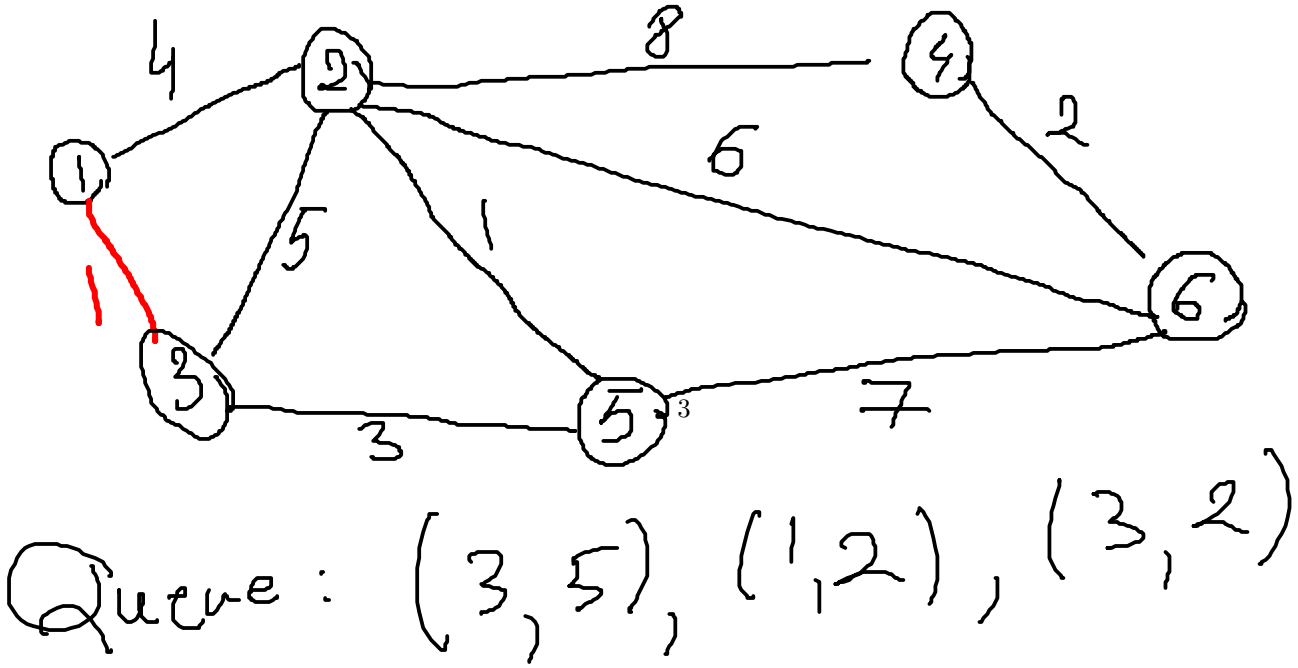
Now we have a complete MST, so we don't have to

(ii) Prim algorithm

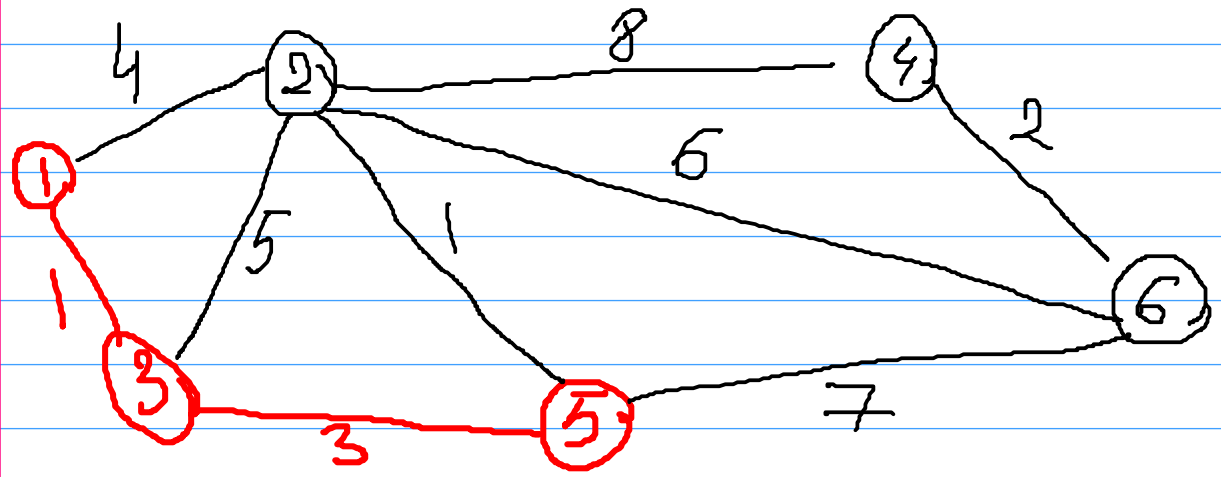
STEP 1 : I start from node 1. I push all the edges that start from it in the priority queue, keeping them sorted by the cost



Step 2 : I go to node 3 (adding the edge (1,3) to the solution) and repeat the actions from Step 1, but this time for node 3.

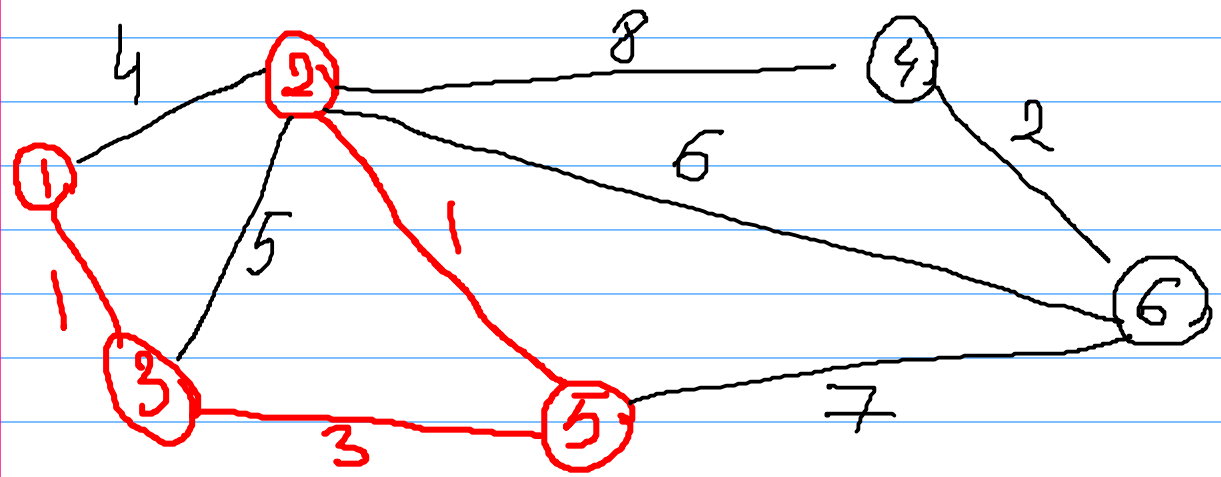


Step 3 : I go to node 5 (adding the edge (3,5) to the solution) and repeat the actions from Step 1, but this time for node 3.



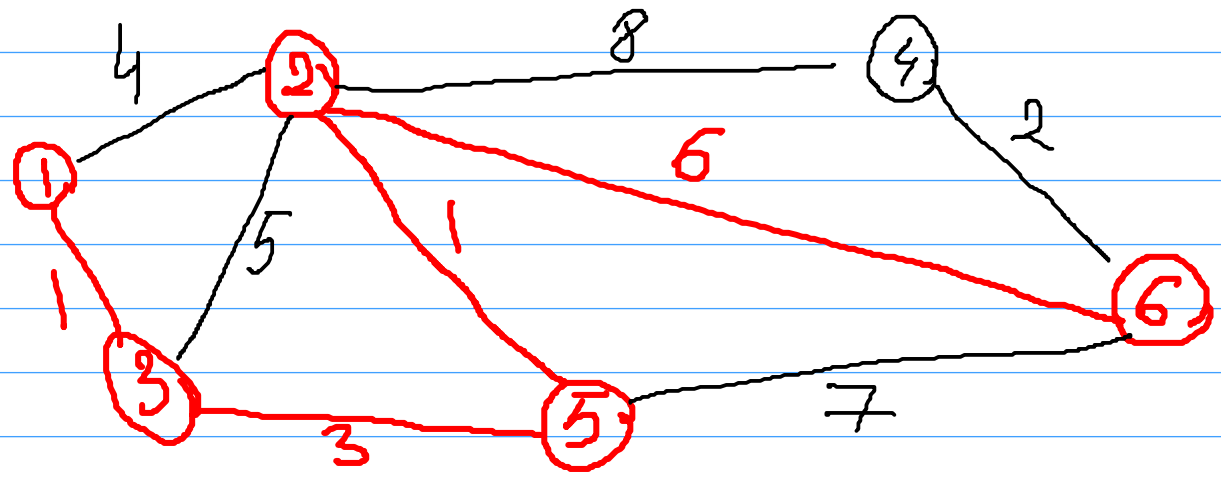
Queue : (5,2), (1,2), (3,2), (5,6)

Step 4 : I go to node 2 (adding the edge (5,2) to the solution) and repeat the actions from Step 1, but this time for node 3.



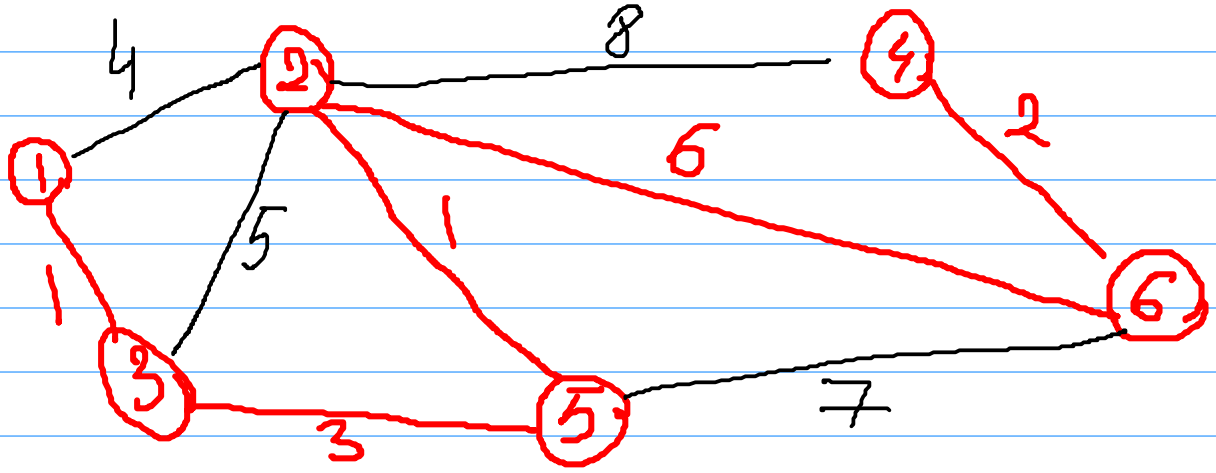
Queue : (1,2), (3,2), (2,6), (5,6), (2,4)

Step 5 : The first two edges from the list are illegal, as they will generate a loop in the MST (which is not allowed). Therefore, we go to node 6, adding edge (2,6) to the solution



Queue : (6,4), (6,5), (5,6), (2,4)

Step 6 : Add (6,4) to the solution.



Solution: (1,3), (3,5), (5,2), (2,6), (6,4)

1.3 Exercise 3

Algorithm 1 Kruskal's algorithm

```
1:  function KRUSKAL(G) :
2:
3:      allEdges = List()                          ▷ The list of all the edges
4:      mstEdges = Set()          ▷ The set of the MST edges, initially empty
5:      dSet = DisjointSet()        ▷ The disjoint set used to implement the
6:                                  ▷ algorithm. Also initially empty
7:      for every vertice v in G do
8:          dSet.makeSet(v)             ▷ Making a separate set for every node
9:      allEdges.add(G.edges) ▷ Adding the edges to the list and sorting them
10:     allEdges.sort()
11:
12:     for every e edge in allEdges do
13:         startSet ← dSet.findSet(e.start())     ▷ Finding the correspondent
14:         endSet ← dSet.findSet(e.end())     ▷ sets for the 2 ends of the edge
15:         if (startSet! = endSet) then
16:             dSet.merge(startSet, endSet)
17:             mstEdges.add(e)
18:     return mstEdges
```

From this pseudocode, we can observe that Kruskal's algorithm performs many *merge*() operations on the disjoint set. Therefore, it is very important how this structure is implemented.
The total running cost of the algorithm can be divided into :
(i) Initialisation cost : This does not depend on the implementation of the disjoint set and has a value of V - from the loop used to create the initial disjoint sets - $+E * log(E)$ - from inserting and sorting the edges. So total cost of $V + E * log(E)$ or $O(E * log(E))$, because $E \geq V$.
(ii) Main loop cost : First, it performs E iterations, in order to go through the entire set of vertices. Then, the cost depends on the way the disjoint set is implemented. We have the following possibilities :

1. **linked-list based :** has a merging complexity of $O(N)$, where N is the size of the longest set. Therefore, the total cost for the main loop would be $O(E * V)$.
2. **weighted union :** in this case we can observe a slight improvement, the total cost of the main loop being $O(E + V * log(V))$
3. **path compression :** the cost of the *merge*() operation in this case would be $\alpha(n)$, so the total cost for the main loop is $O(E * \alpha(n))$.

In conclusion, we have the following costs for our algorithm, depending on the way the disjoint set is implemented in :

1. **linked-list based :** $O(E * V)$
2. **weighted union :** $O(E * log(E) + V * log(V))$
3. **path compression :** $O(E * log(n))$.

1.4 Exercise 4

I will submit the rest of my work later today and I will also have a hard copy on me tomorrow.