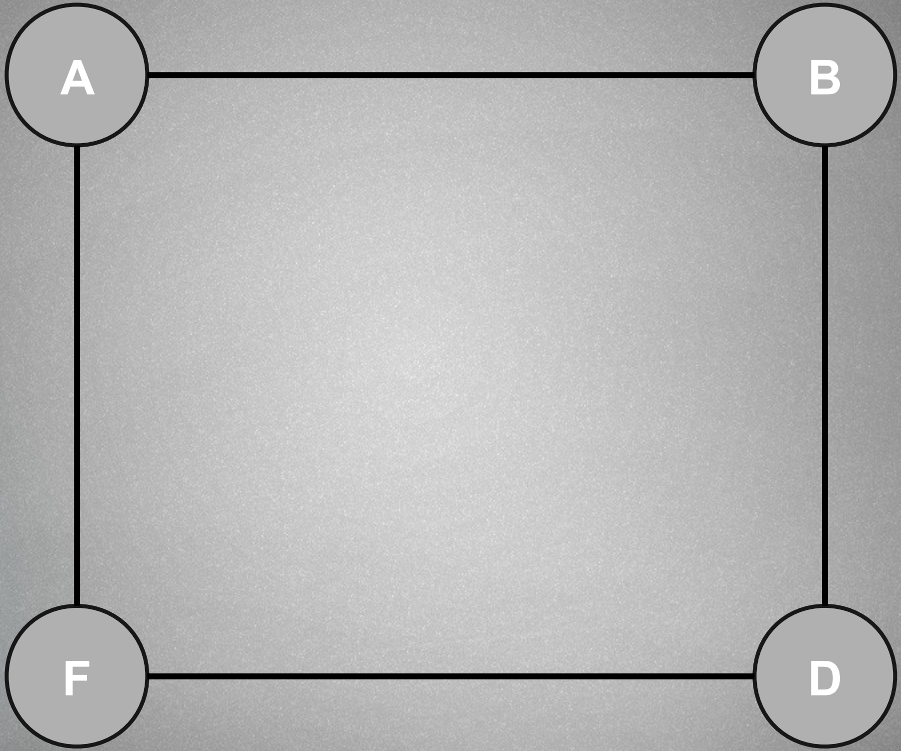
SPANNING TREES

Kruskal algorithm

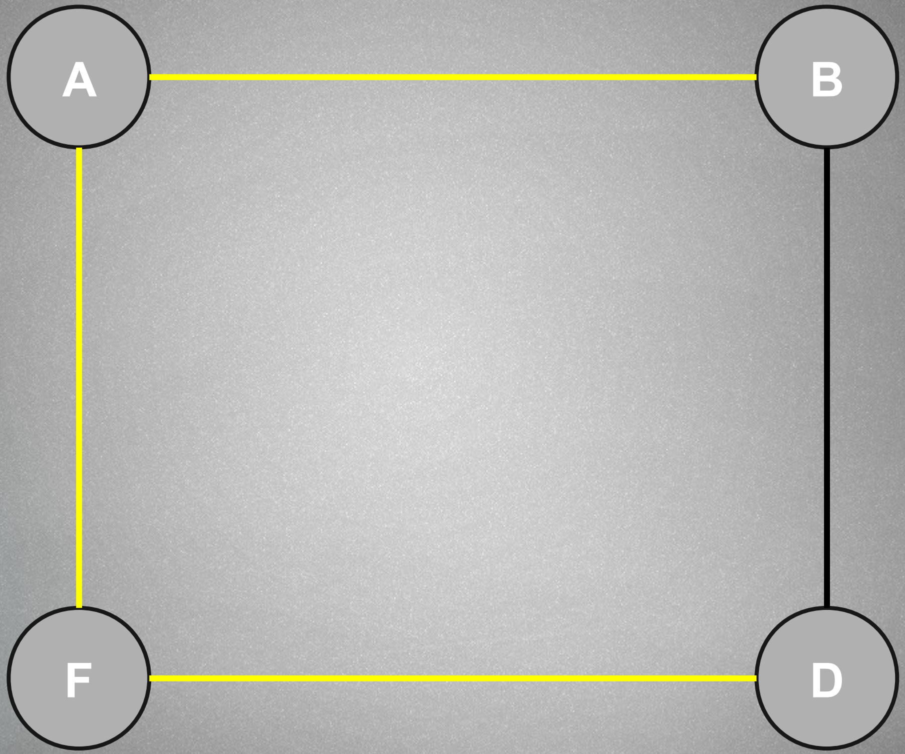
Spanning trees

- A spanning tree of an undirected **G** graph is a subgraph that includes all the vertices of **G**
- In general, a tree may have several spanning trees
- We can assign a weight to each edge
- A minimum spanning tree is then a spanning tree with weight less than or equal to the weight of every other spanning tree
- Has lots of applications: in big data analysis, clustering algorithms, finding minimum cost for a telecommunications company laying cable to a new neighborhood
- Standard algorithms: Prim’s-Jarnik, Kruskal → greedy algorithms

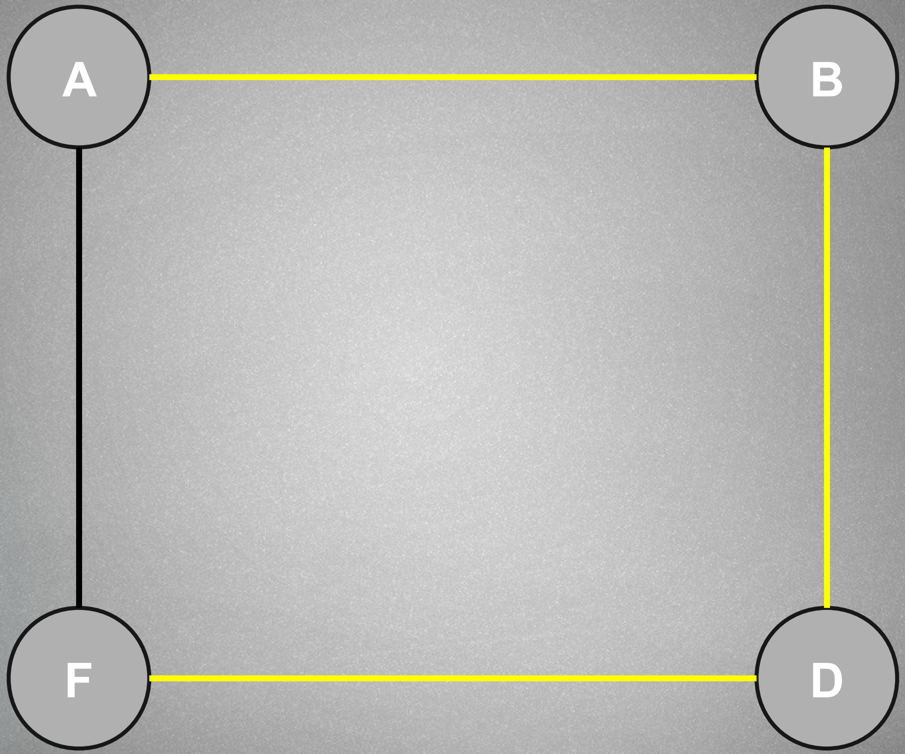
A graph may have several spanning tree !!!



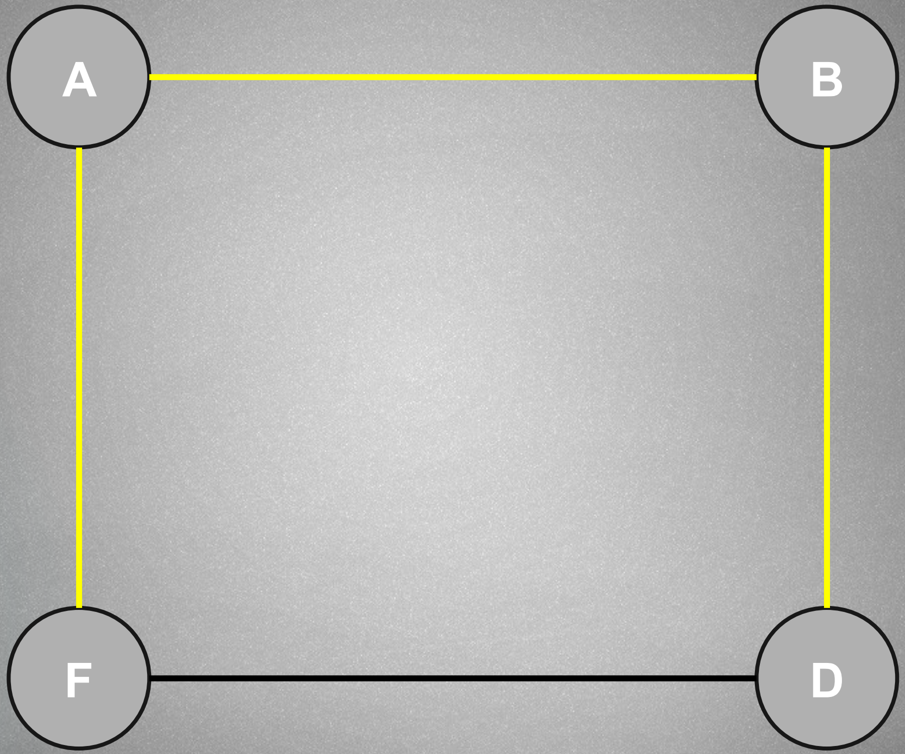
A graph may have several spanning tree !!!



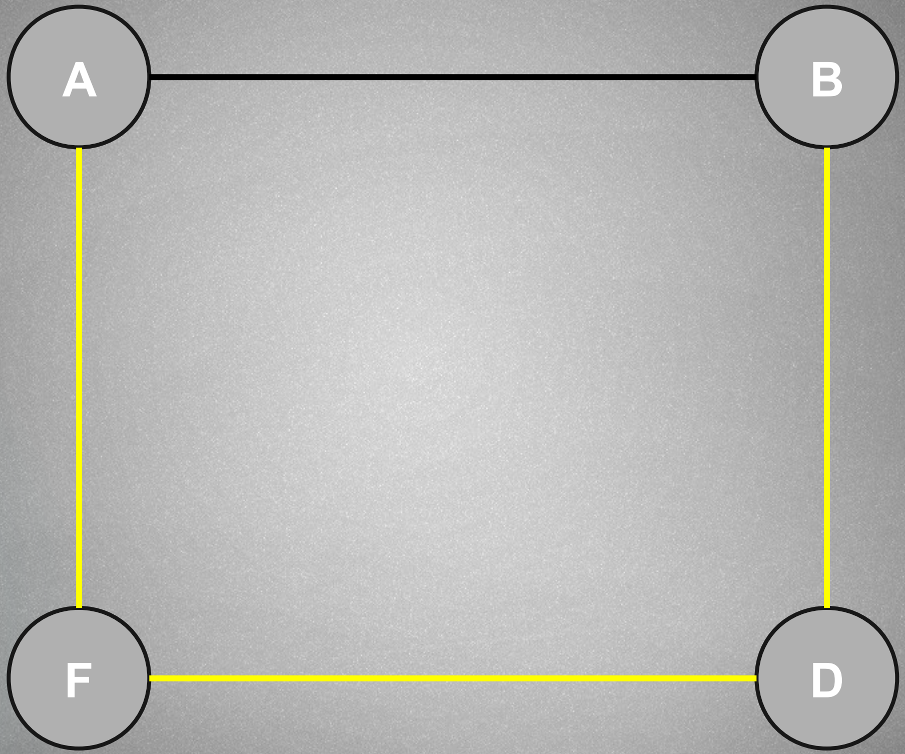
A graph may have several spanning tree !!!



A graph may have several spanning tree !!!



A graph may have several spanning tree !!!



Usually we are looking for the minimum spanning tree: the spanning tree where the sum of edge weights is the lowest possible !!!

Kruskal-algorithm

- We sort the edges according to their edge weights
- It can be done in **O(N*logN)** with mergesort or quicksort
- Union find data structure: „disjoint set"

We start adding edges to the MST and we want to make sure there will be no cycles in the spanning tree. It can be done in **O(logV)** with the help of union find data structure

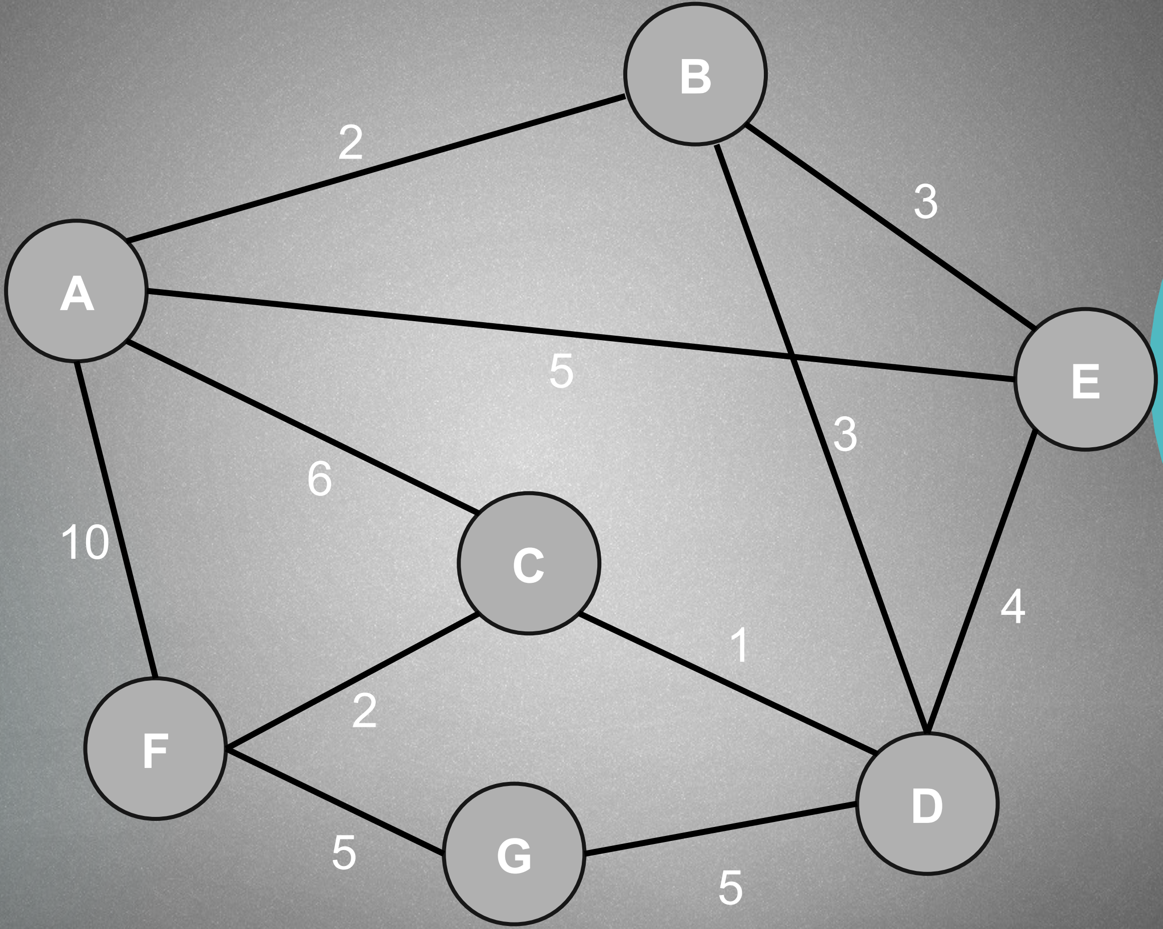
// we could use a heap instead sorting the edges in the beginning but the running time would be the same. So sometimes Kruskal's algorithm is implemented with priority queues

Kruskal-algorithm

- Worst case running time: **O(E*logE)**, so we can use it for huge graphs too
- If the edges are sorted: the algorithm will be quasi-linear
- If we multiply the weights with a constant or add a constant to the edge weights: the result will be the same

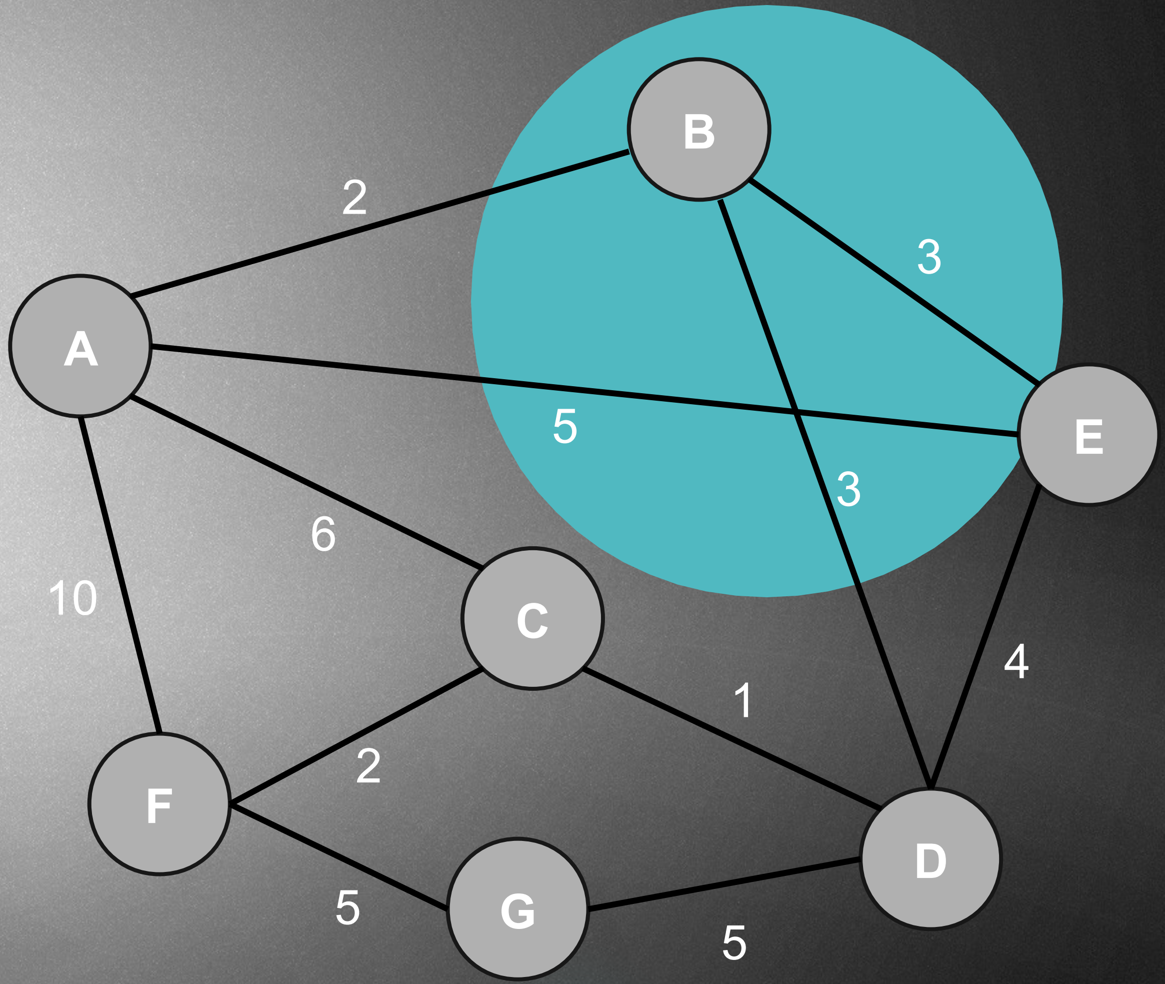
  // in physics, an invariant is a property of the system that remains unchanged under some transformation.

  In Kruskal algorithm, spanning trees are invariant under the transformation of these weights (addition, multiplication)


A
B
C
D
E
F
G
2
3
5
3
6
10
1
4
2
5
5

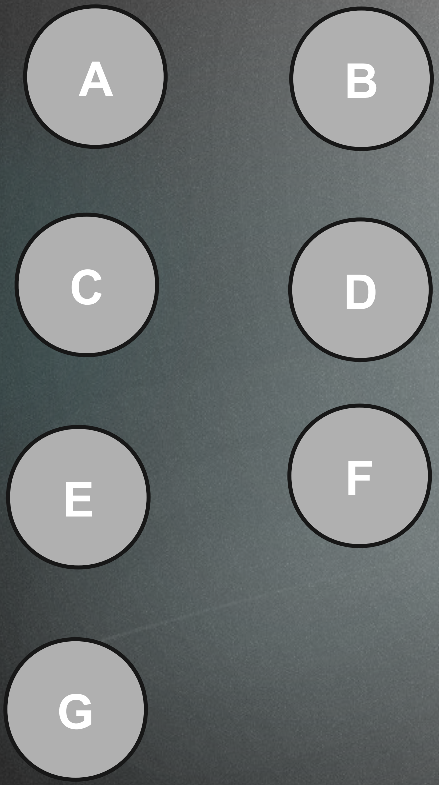
We have to sort the edges: 1, 2, 2, 3, 3, 4, 5, 5, 5, 6, 10

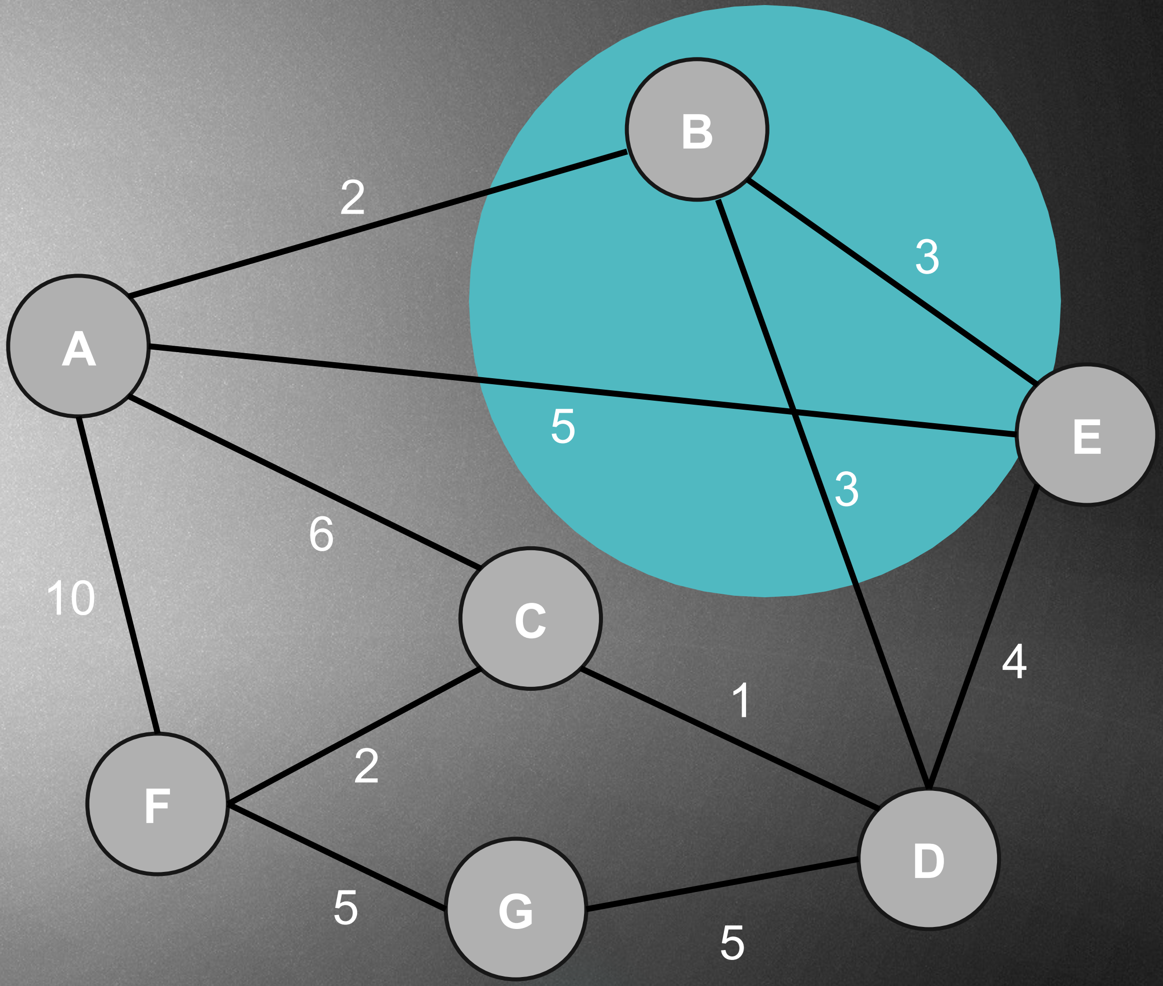
On every iteration we have to make sure whether by adding the new edge → will there be a cycle or not ...



We have to sort the edges: 1, 2, 2, 3, 3, 4, 5, 5, 5, 6, 10

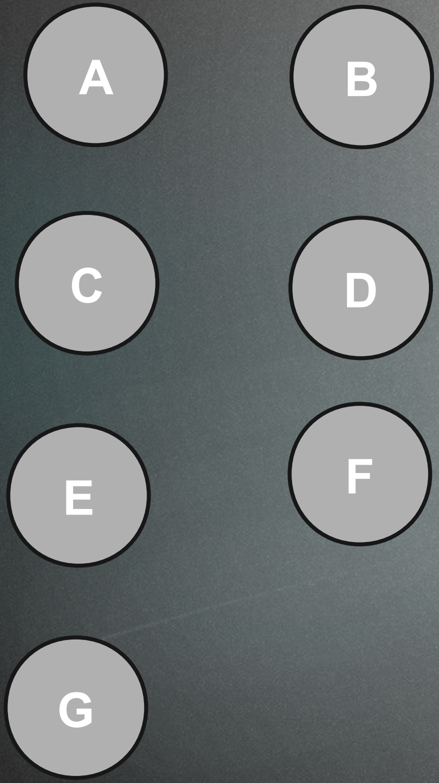
Disjoint sets

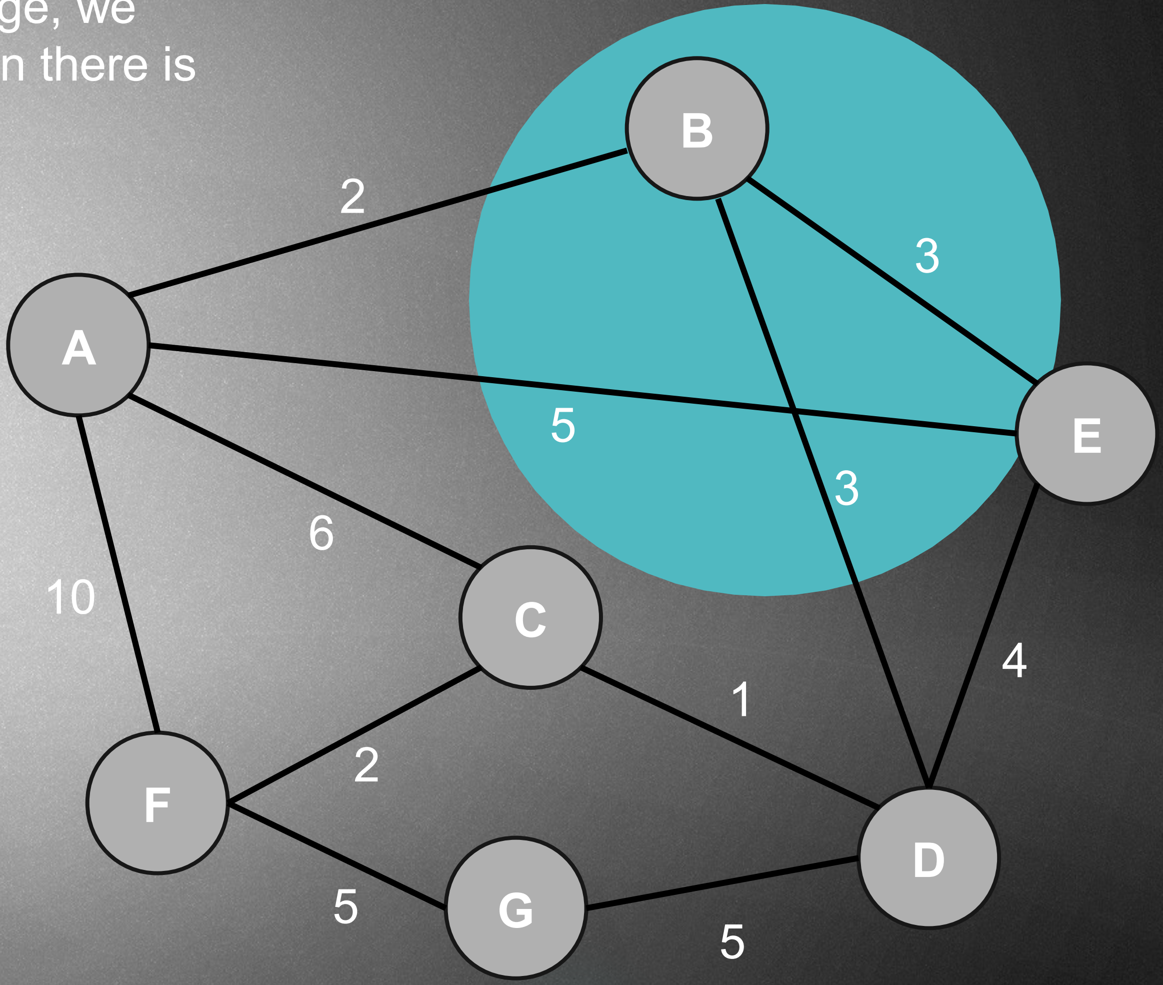




We have to sort the edges: 1, 2, 2, 3, 3, 4, 5, 5, 5, 6, 10

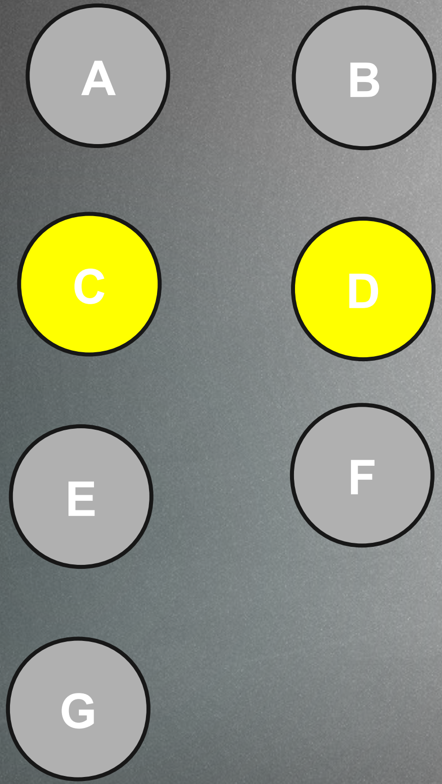
Disjoint sets: at the beginning we have as many sets as the number of vertices. When adding an edge, we merge two sets together ... the algorithm stops when there is only a single set remains

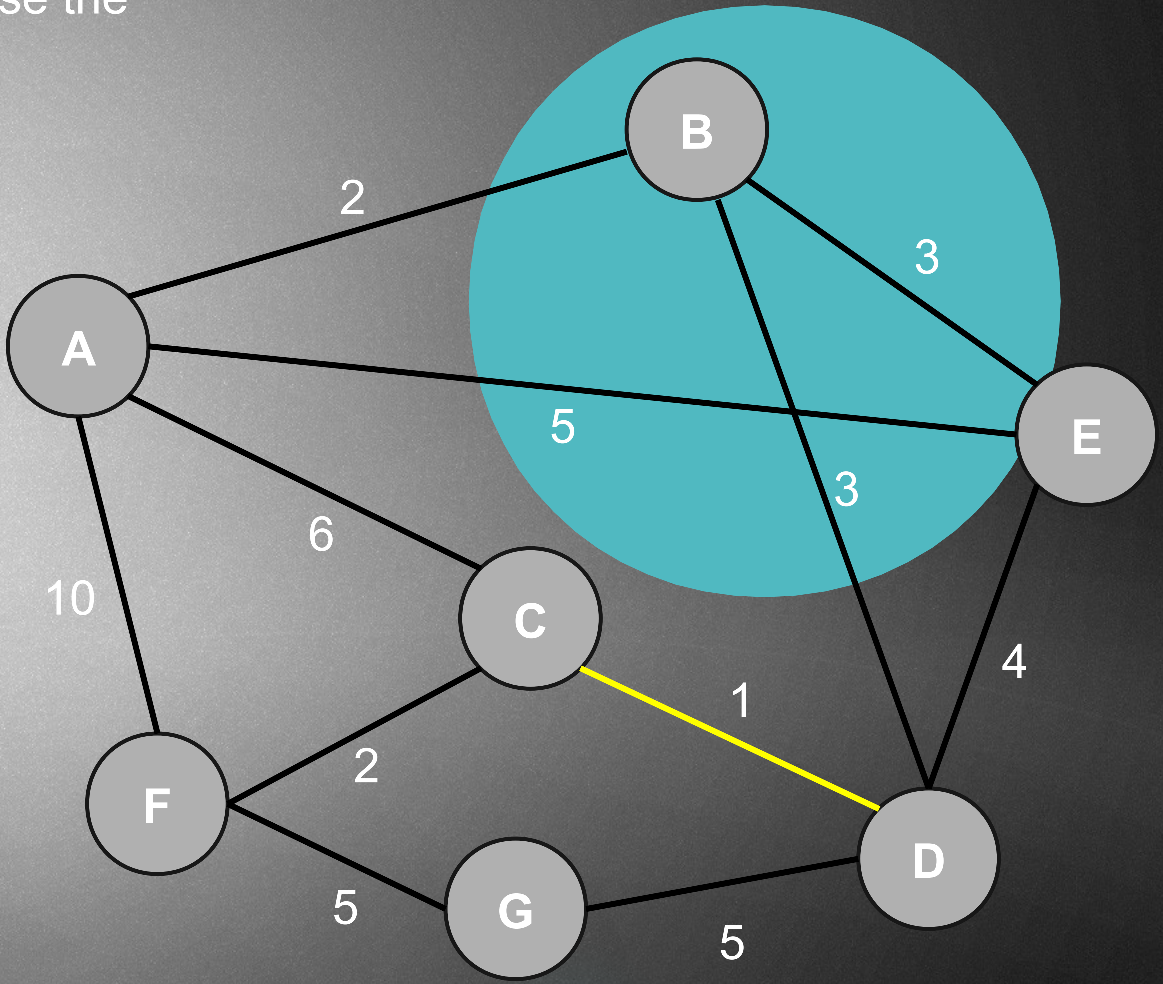




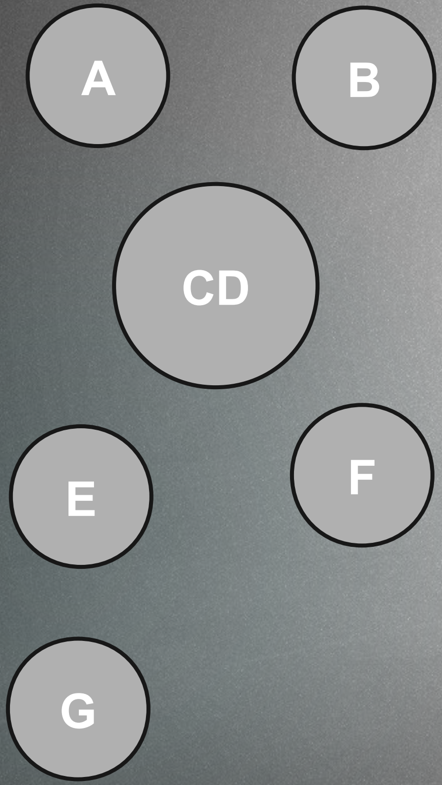
We have to sort the edges: 1, 2, 2, 3, 3, 4, 5, 5, 5, 6, 10

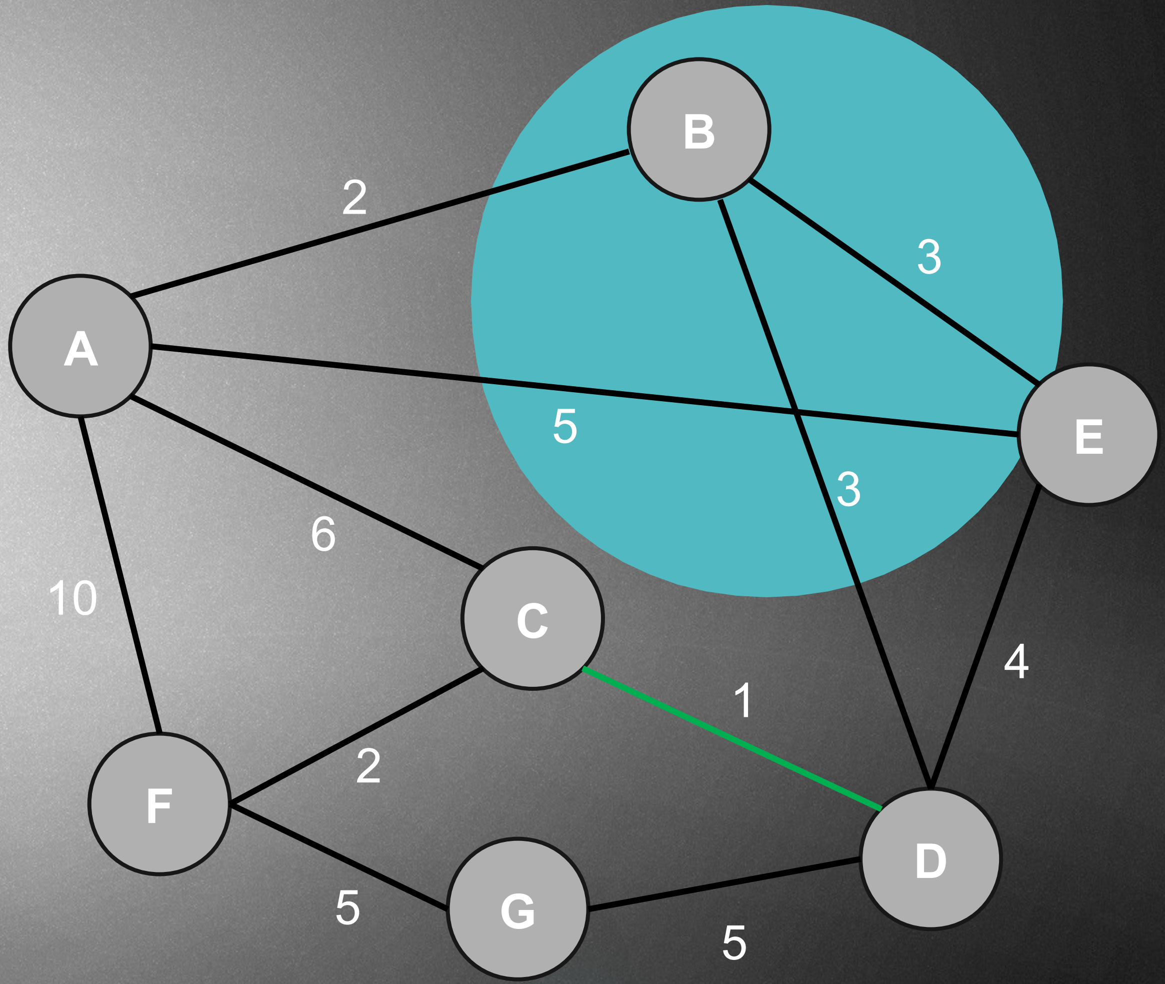
We can add the edge to the spanning tree because the the node are in distinct sets !!!



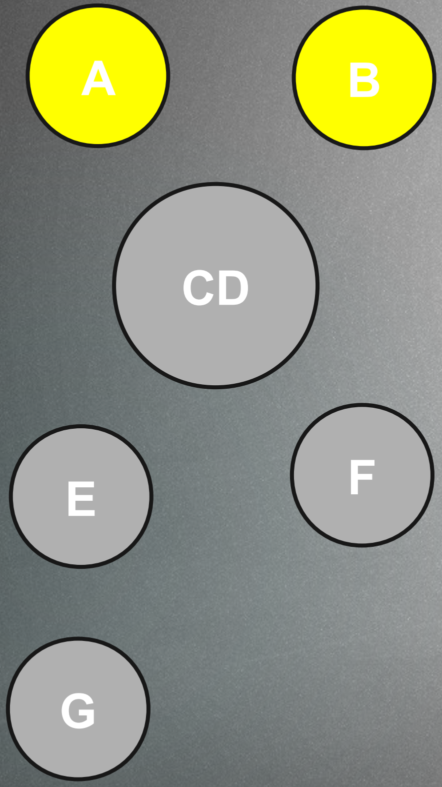


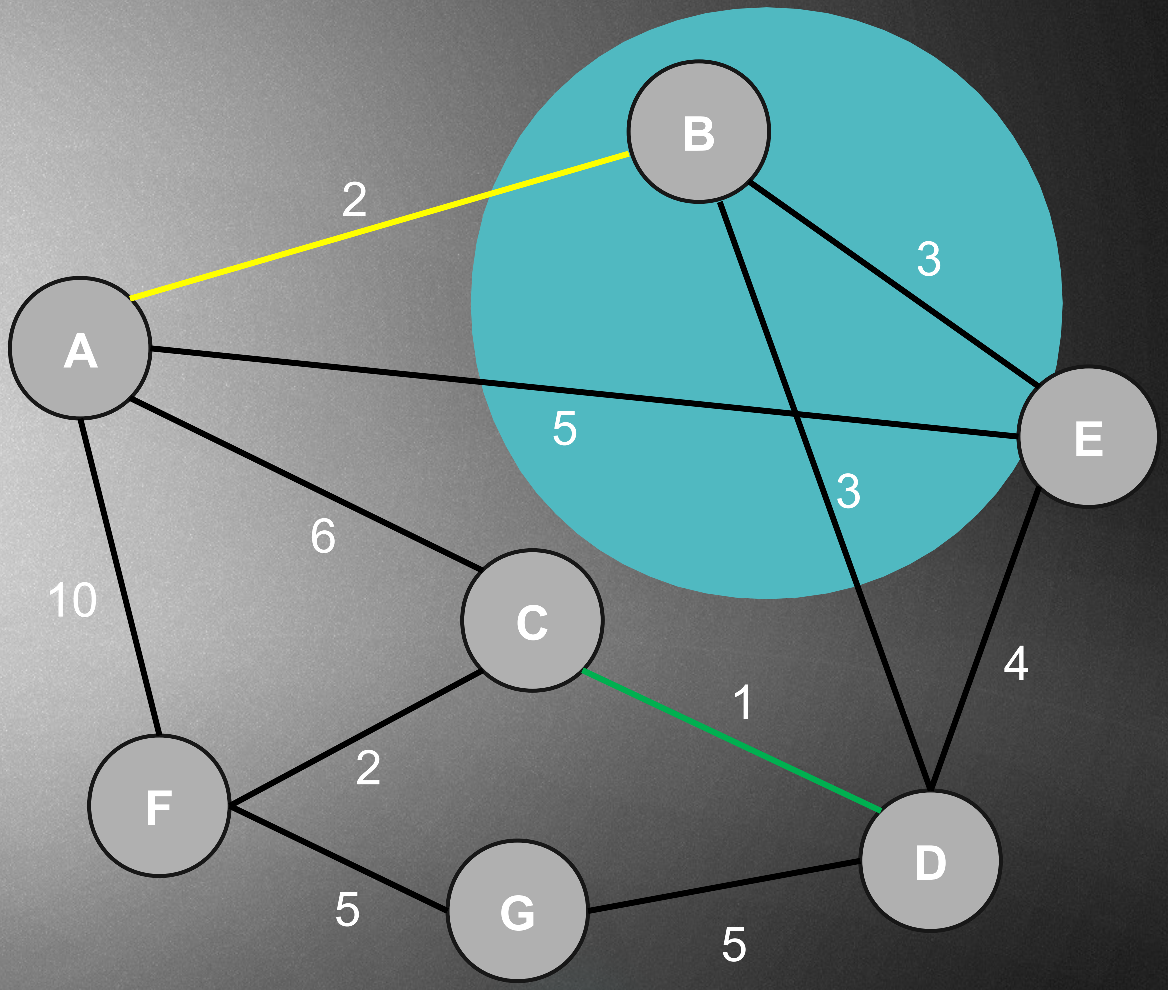
We have to sort the edges: 1, 2, 2, 3, 3, 4, 5, 5, 5, 6, 10



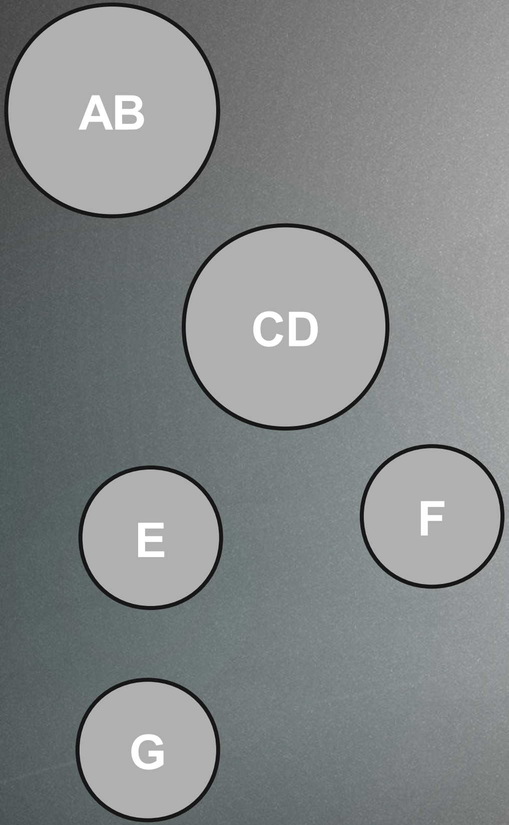


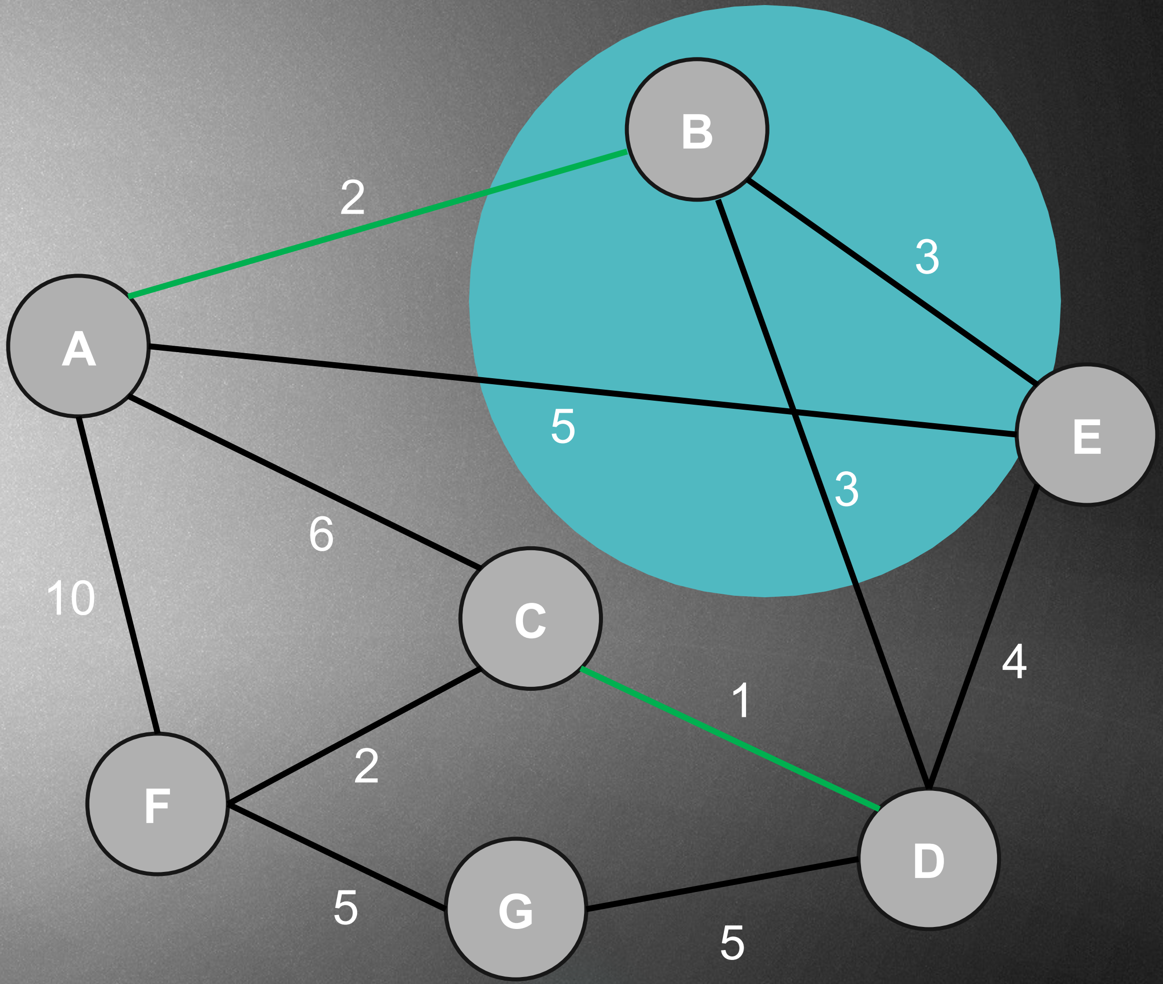
We have to sort the edges: 1, 2, 2, 3, 3, 4, 5, 5, 5, 6, 10



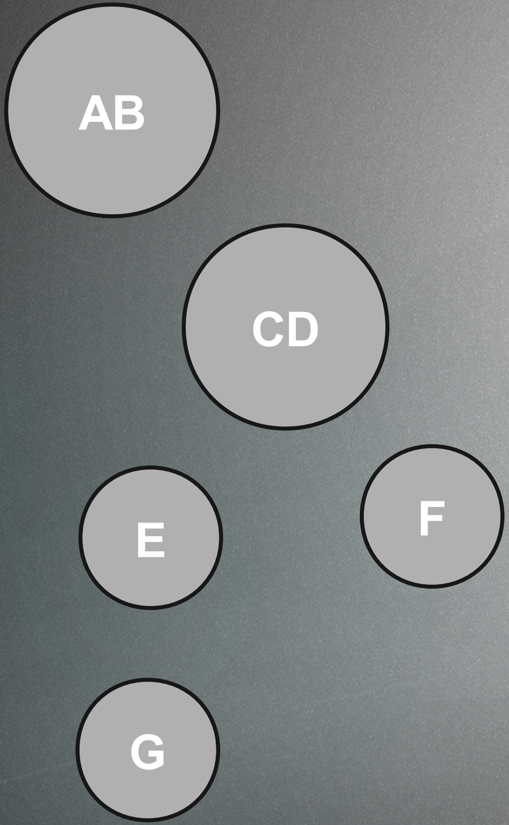


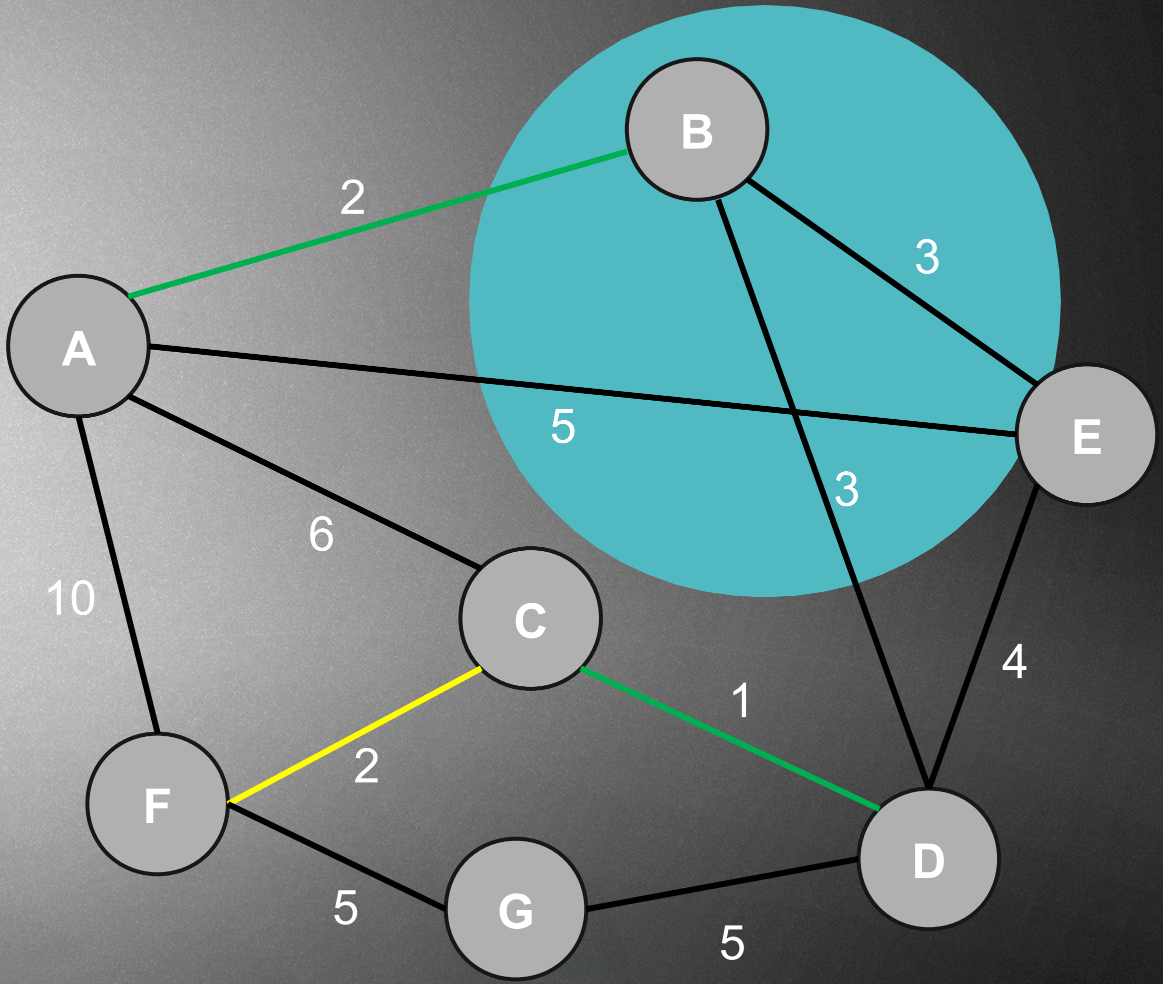
We have to sort the edges: 1, 2, 2, 3, 3, 4, 5, 5, 5, 6, 10



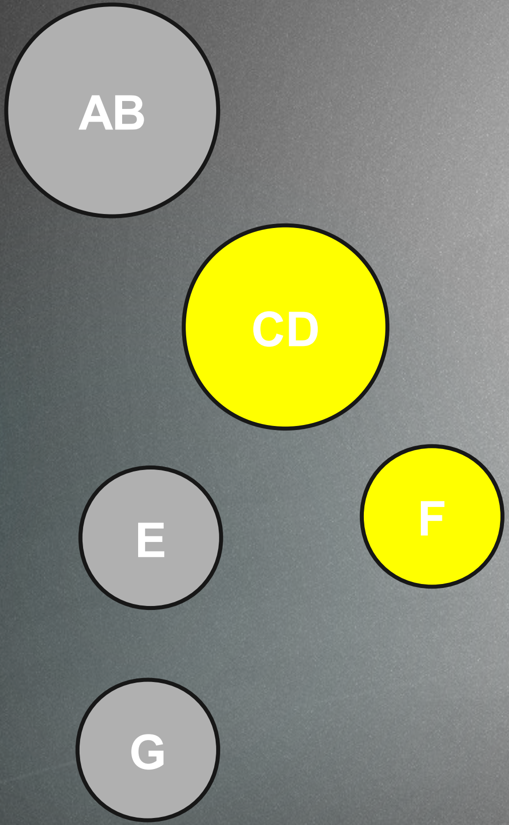


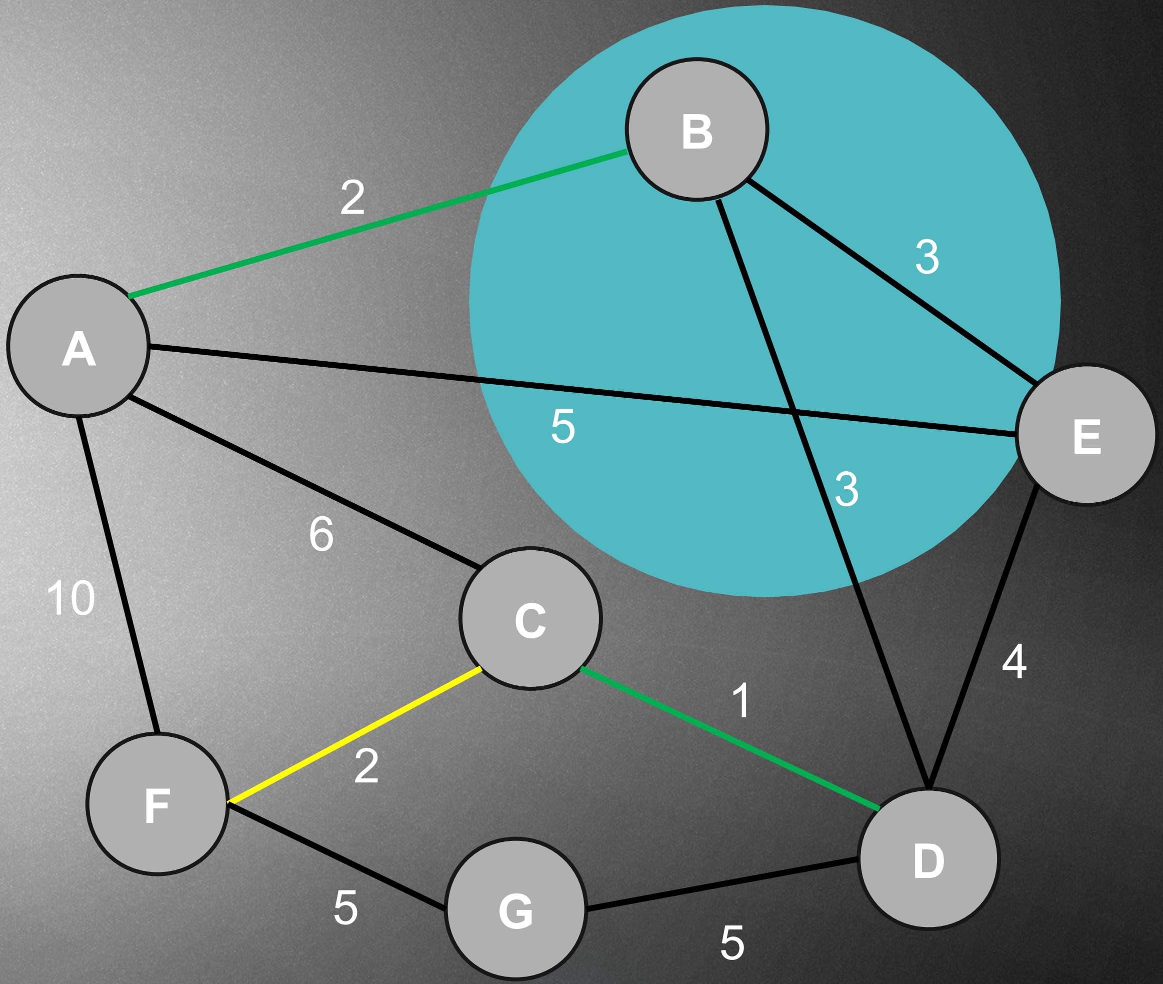
We have to sort the edges: 1, 2, 2, 3, 3, 4, 5, 5, 5, 6, 10



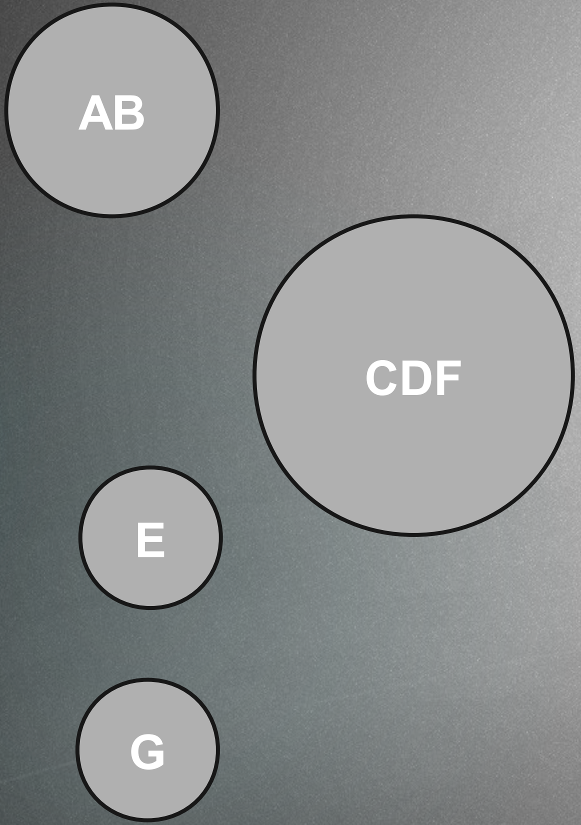


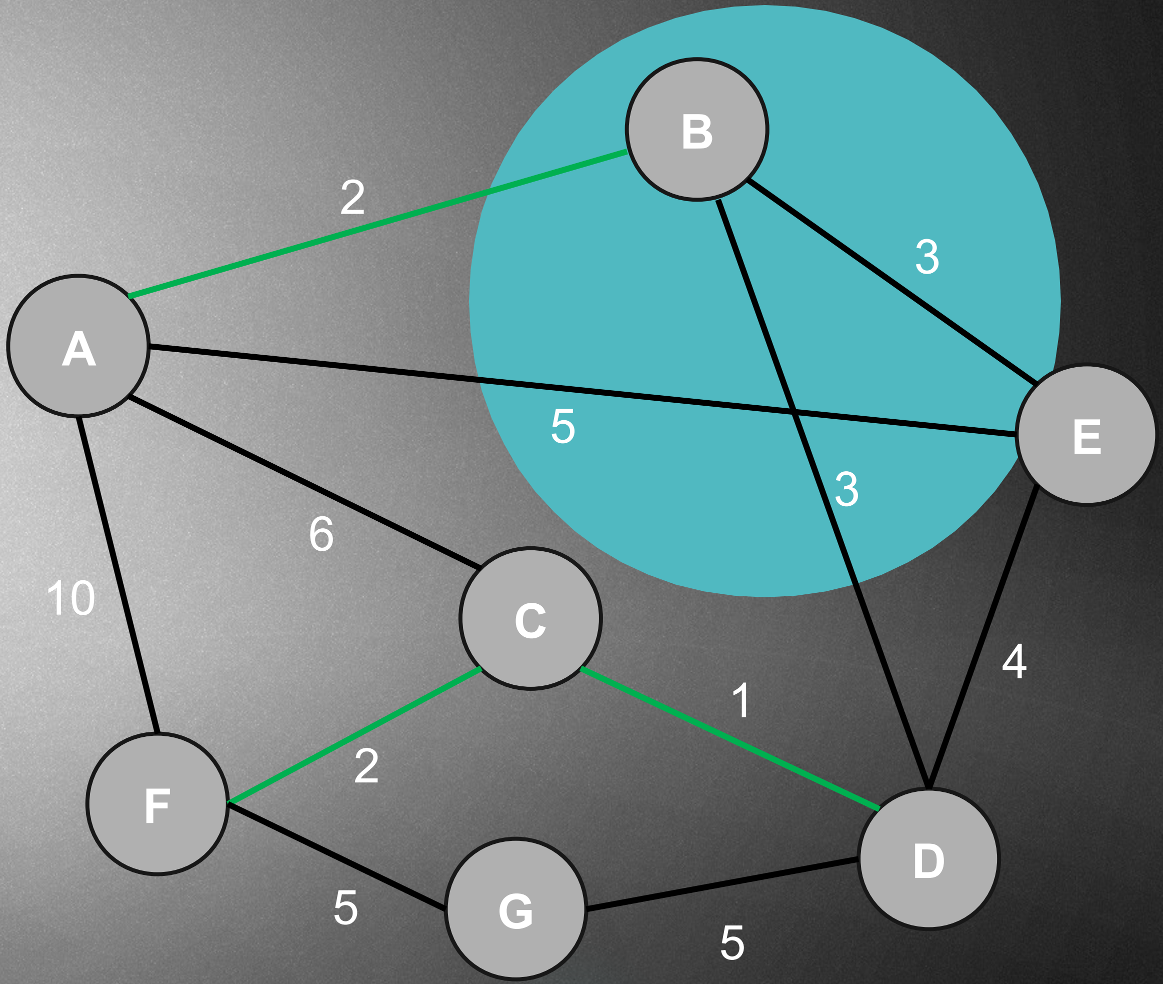
We have to sort the edges: 1, 2, 2, 3, 3, 4, 5, 5, 5, 6, 10



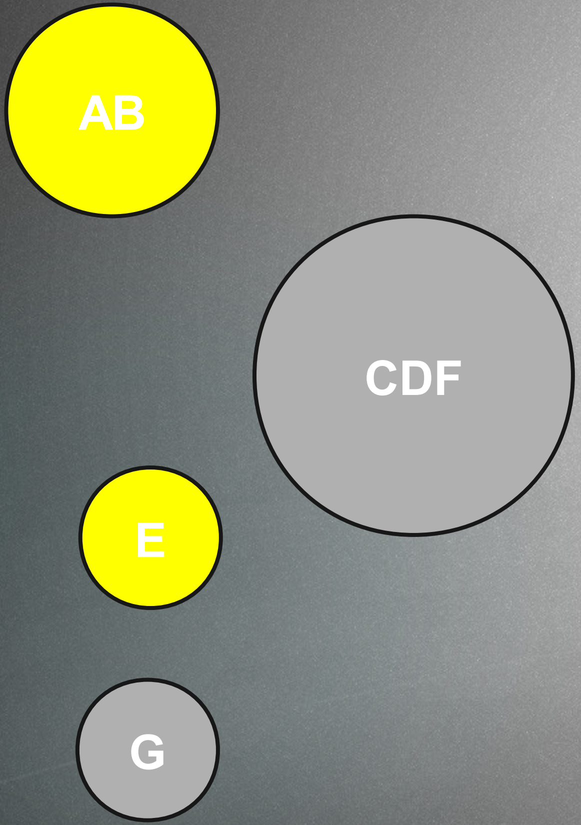


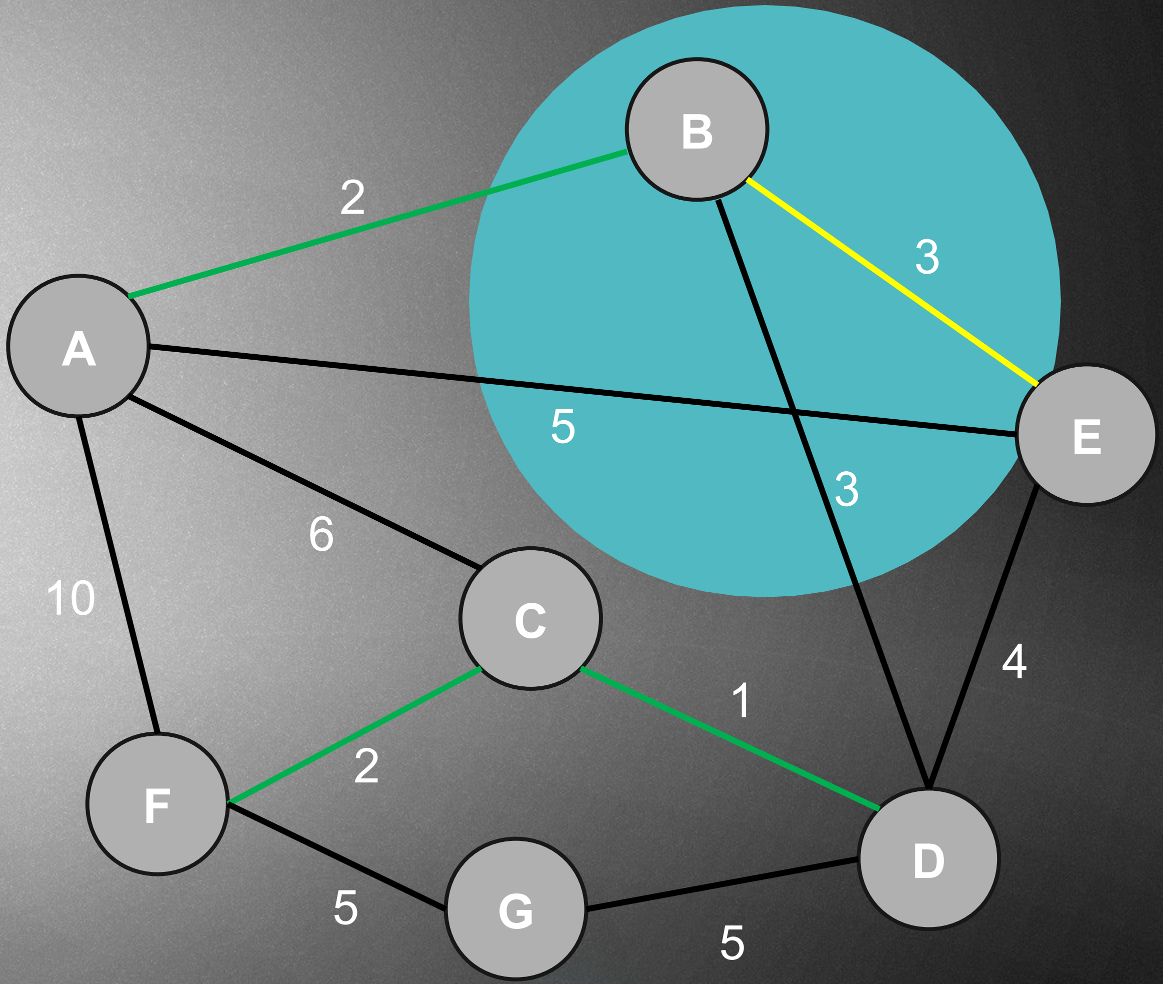
We have to sort the edges: 1, 2, 2, 3, 3, 4, 5, 5, 5, 6, 10



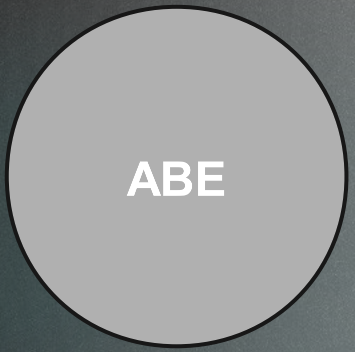


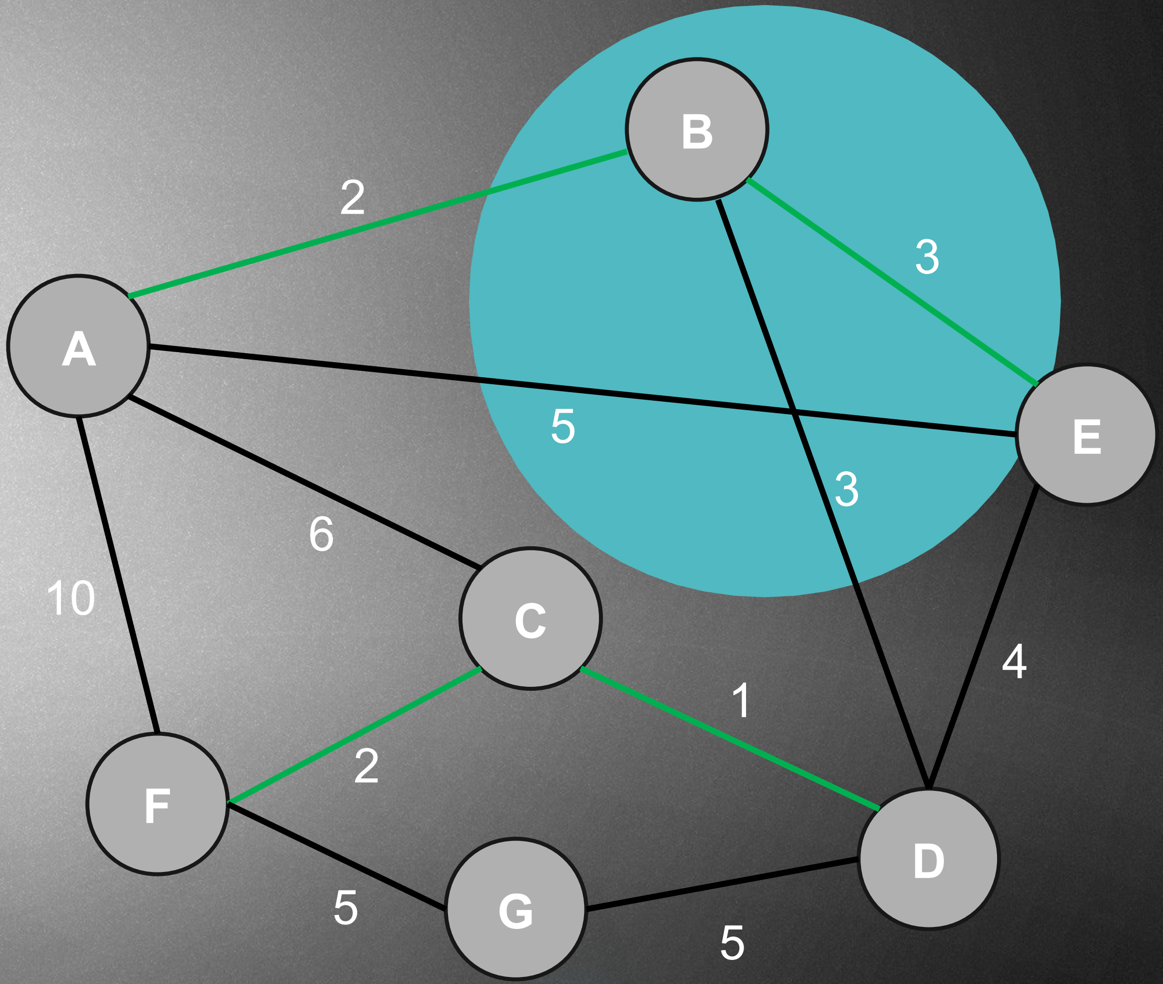
We have to sort the edges: 1, 2, 2, 3, 3, 4, 5, 5, 5, 6, 10



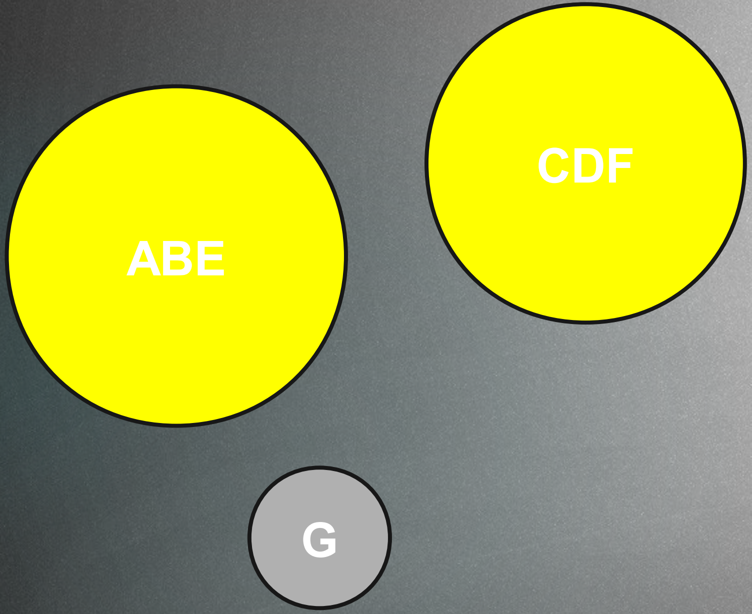


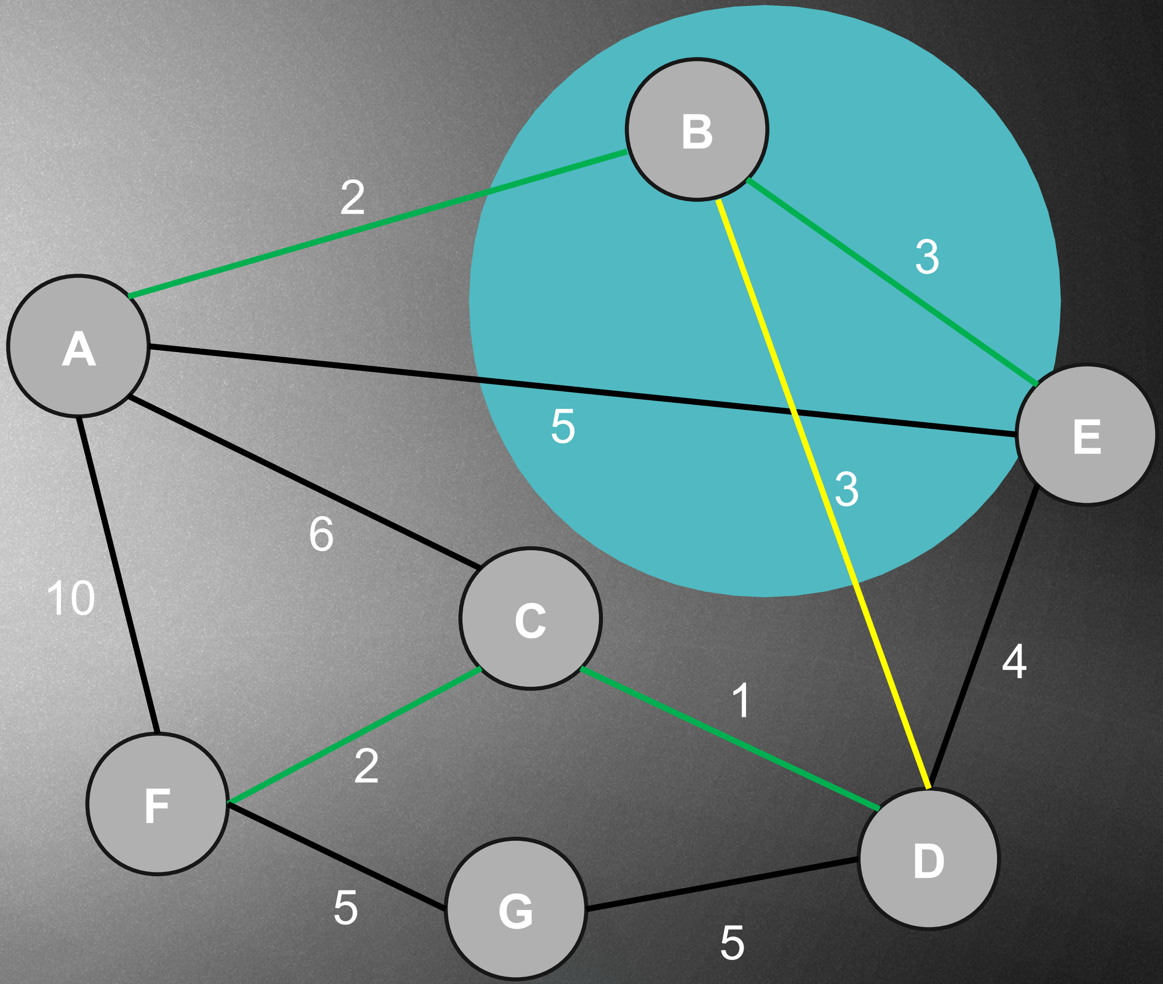
We have to sort the edges: 1, 2, 2, 3, 3, 4, 5, 5, 5, 6, 10



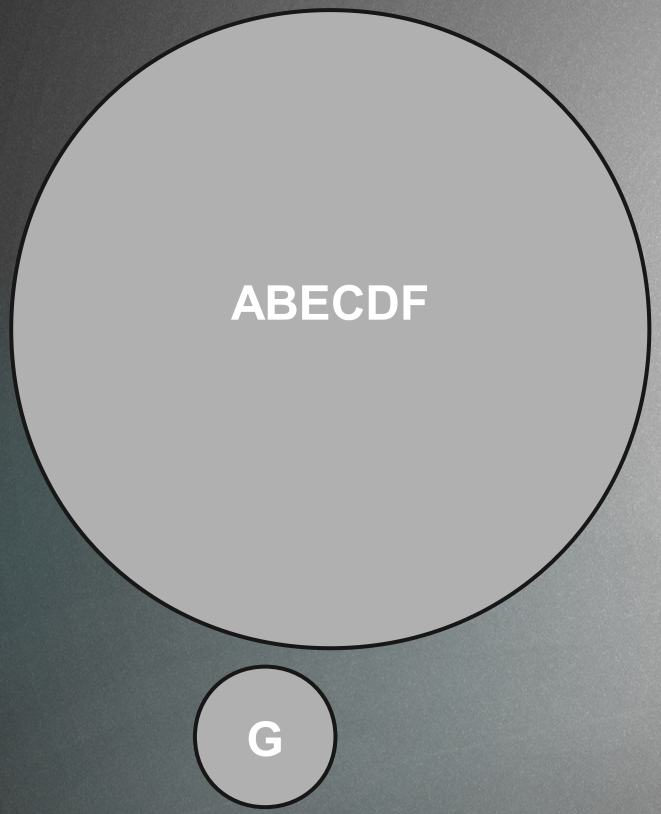


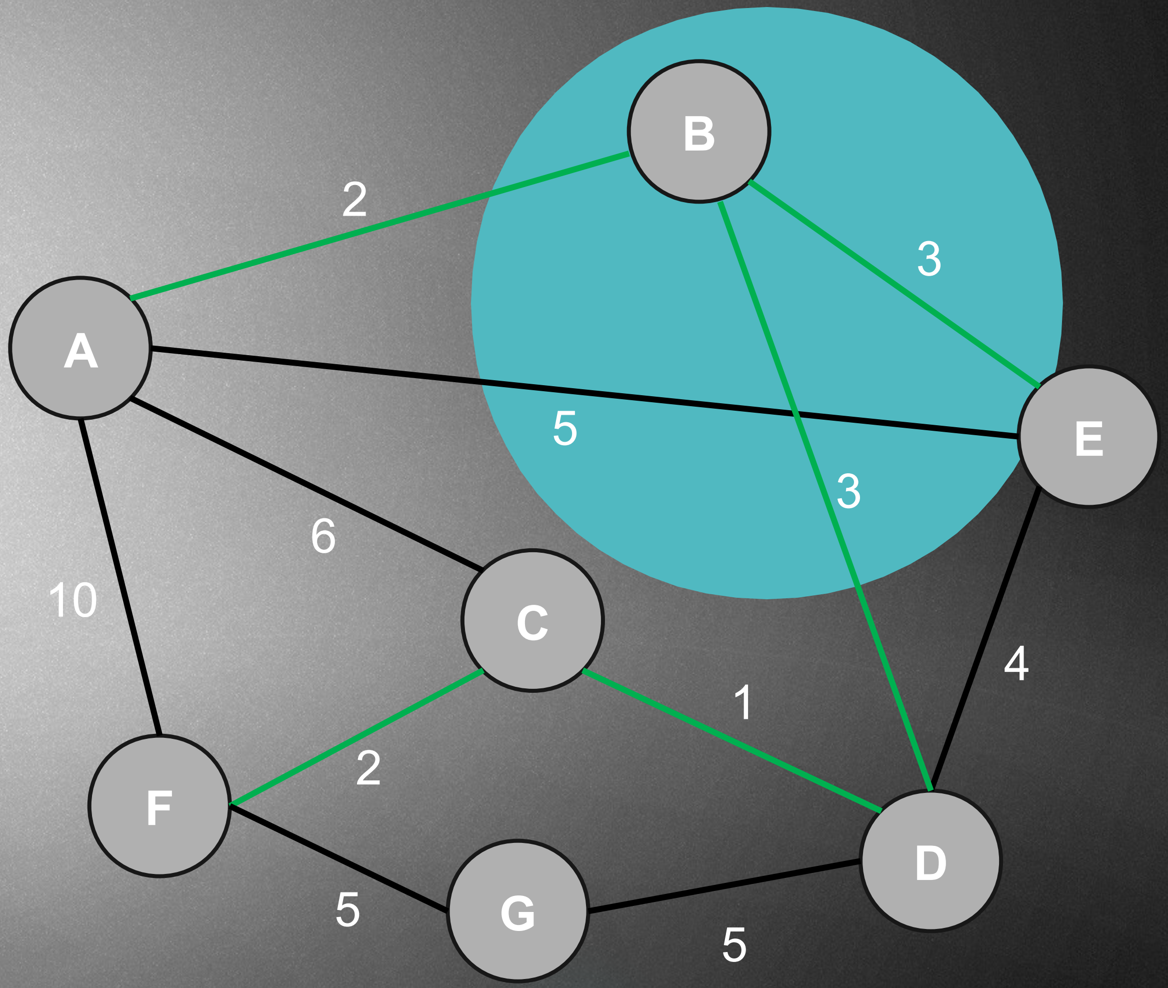
We have to sort the edges: 1, 2, 2, 3, 3, 4, 5, 5, 5, 6, 10



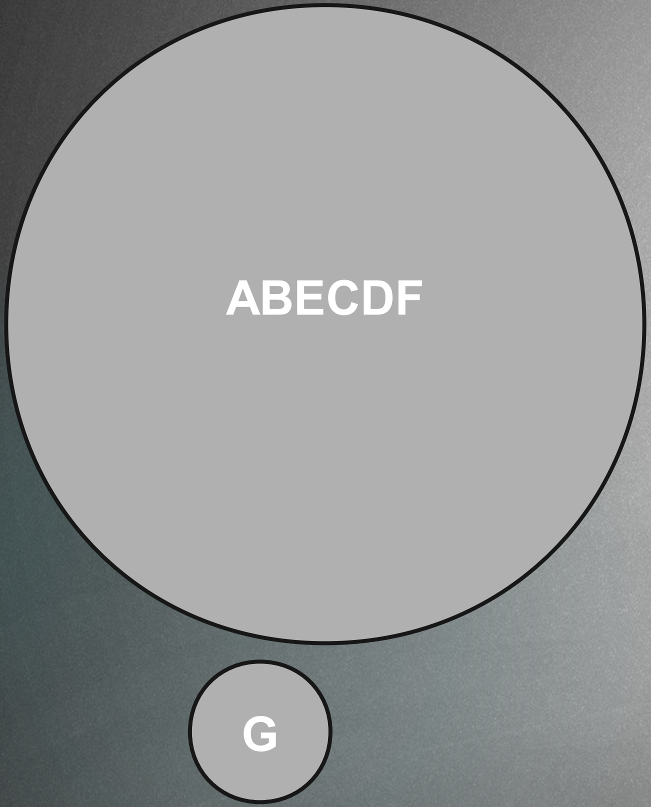


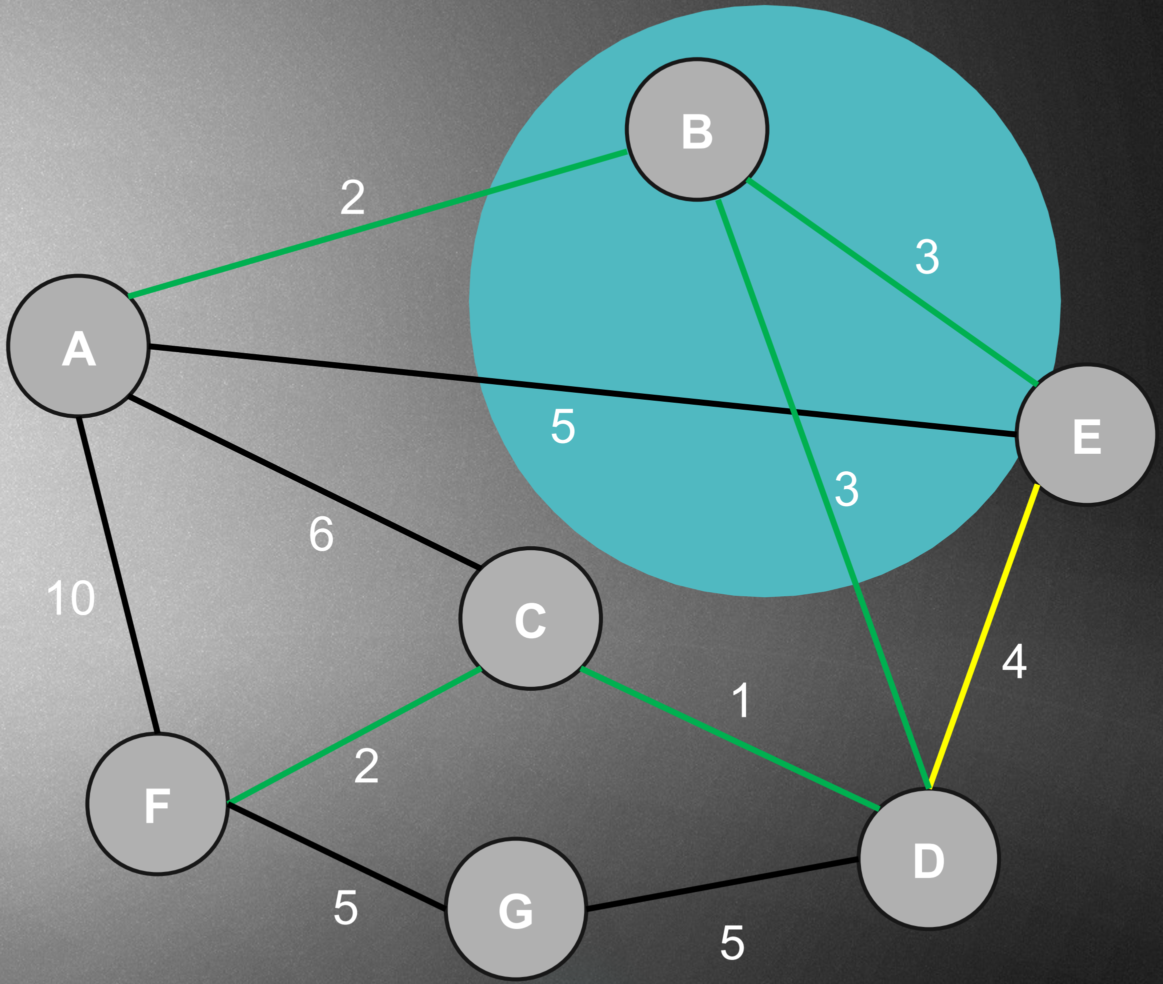
We have to sort the edges: 1, 2, 2, 3, 3, 4, 5, 5, 5, 6, 10



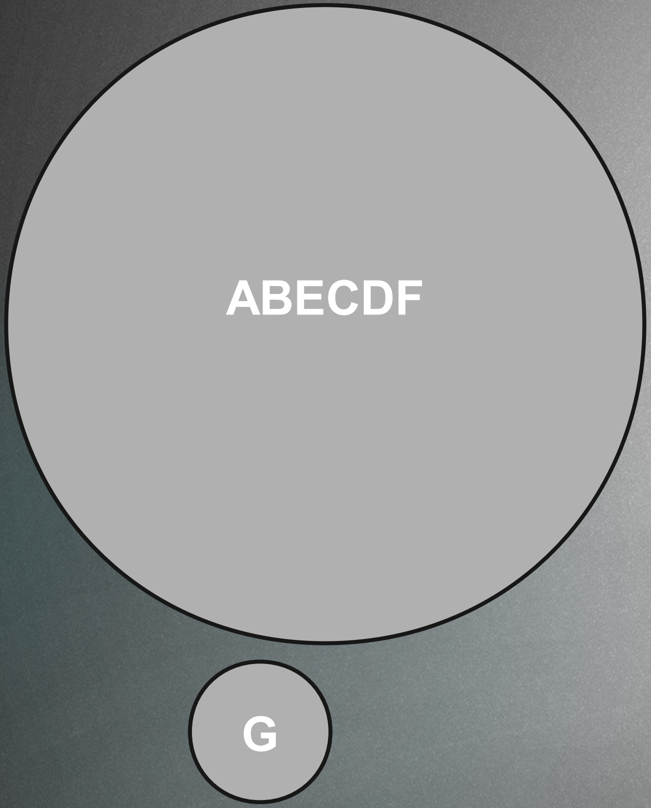


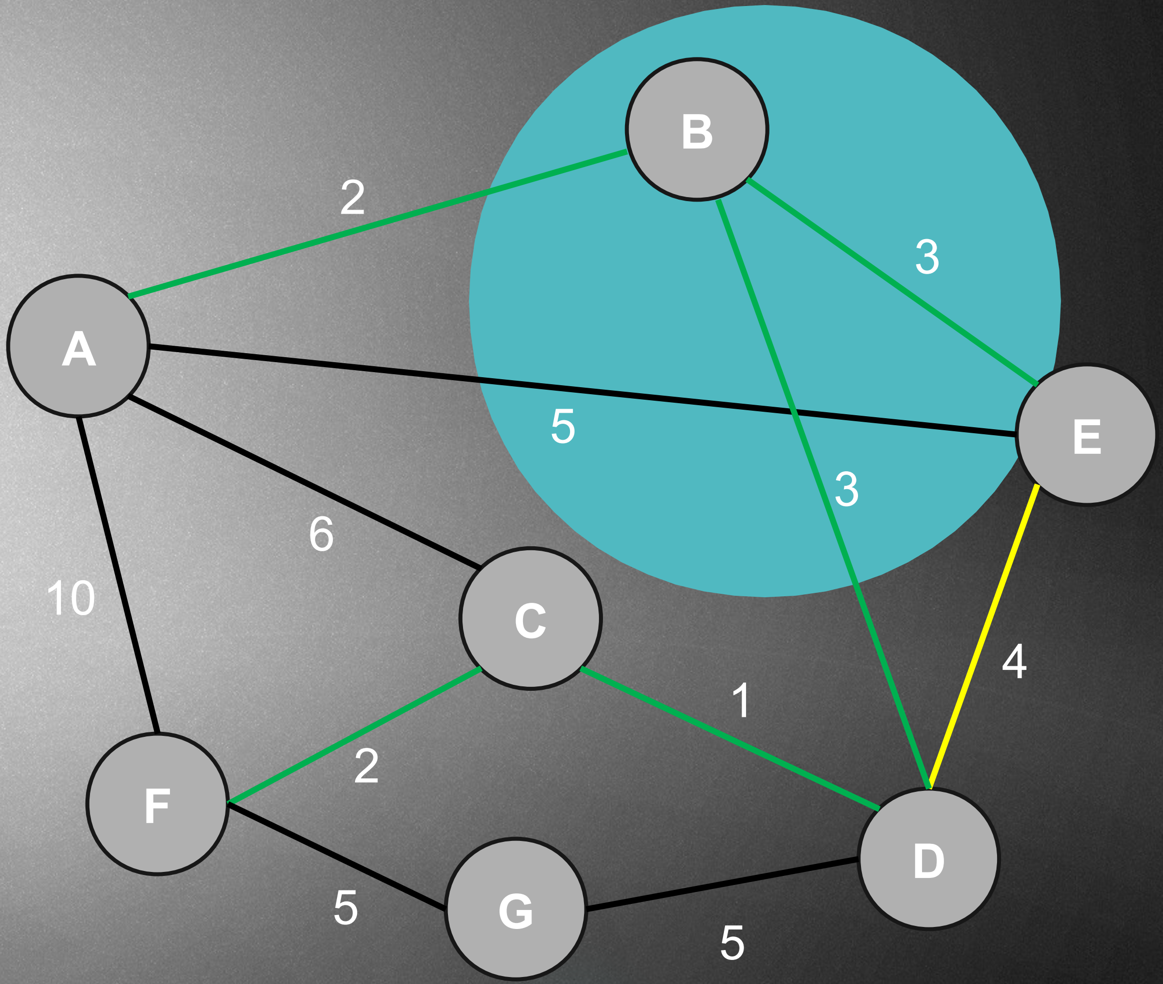
We have to sort the edges: 1, 2, 2, 3, 3, 4, 5, 5, 5, 6, 10



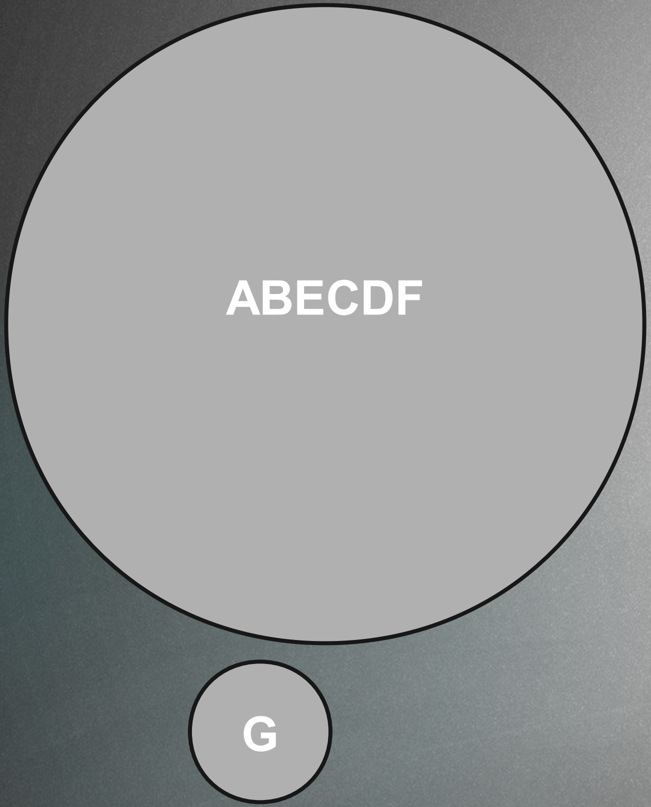


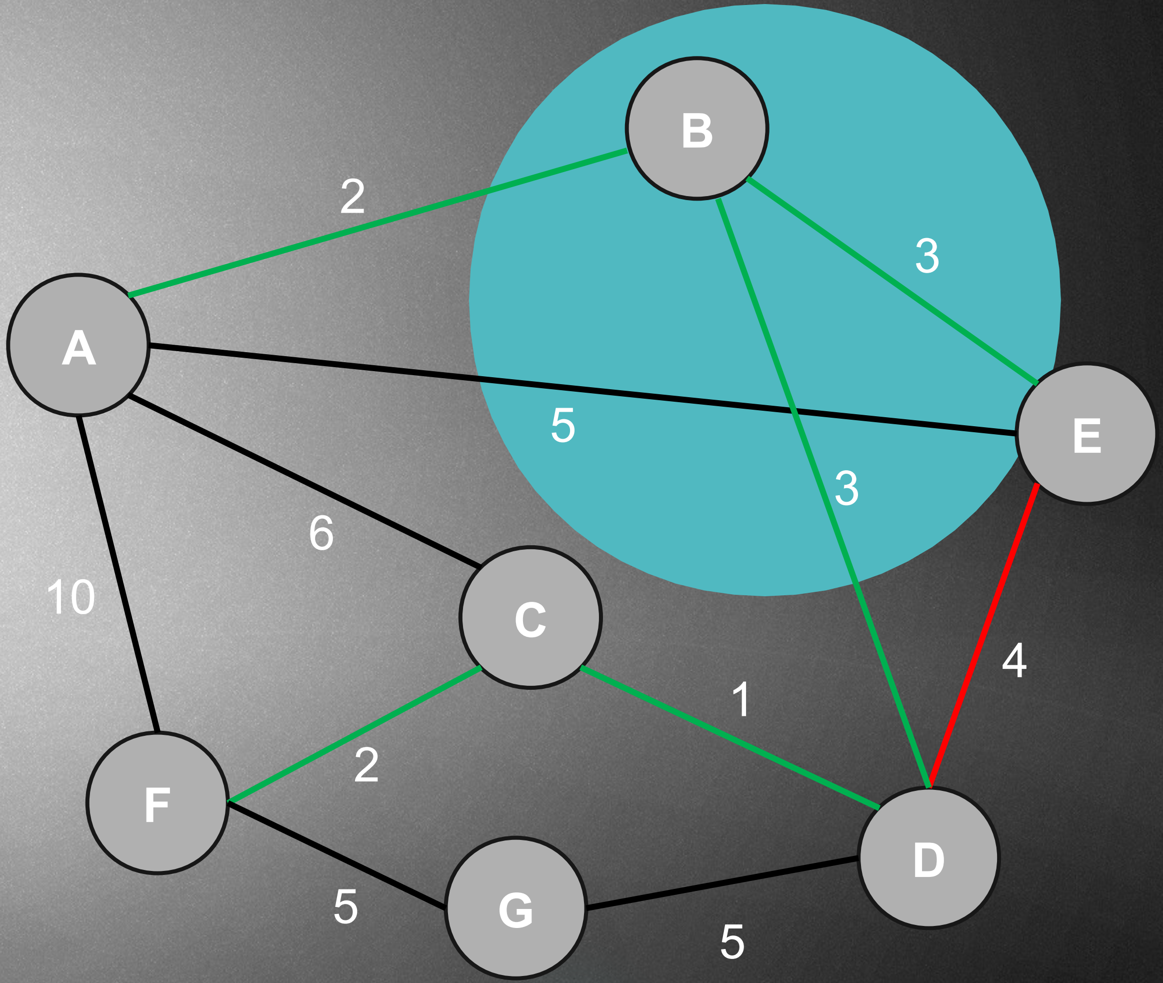
We have to sort the edges: 1, 2, 2, 3, 3, 4, 5, 5, 5, 6, 10



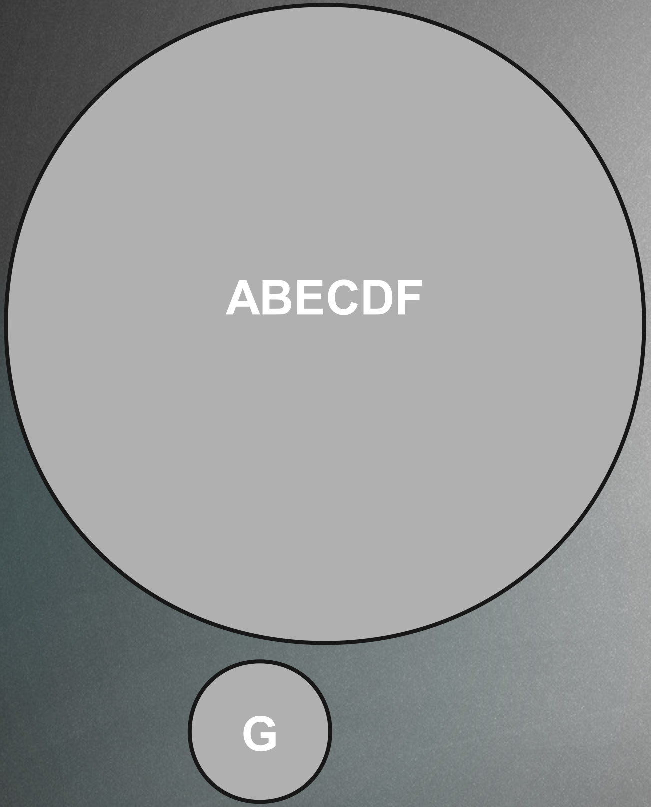


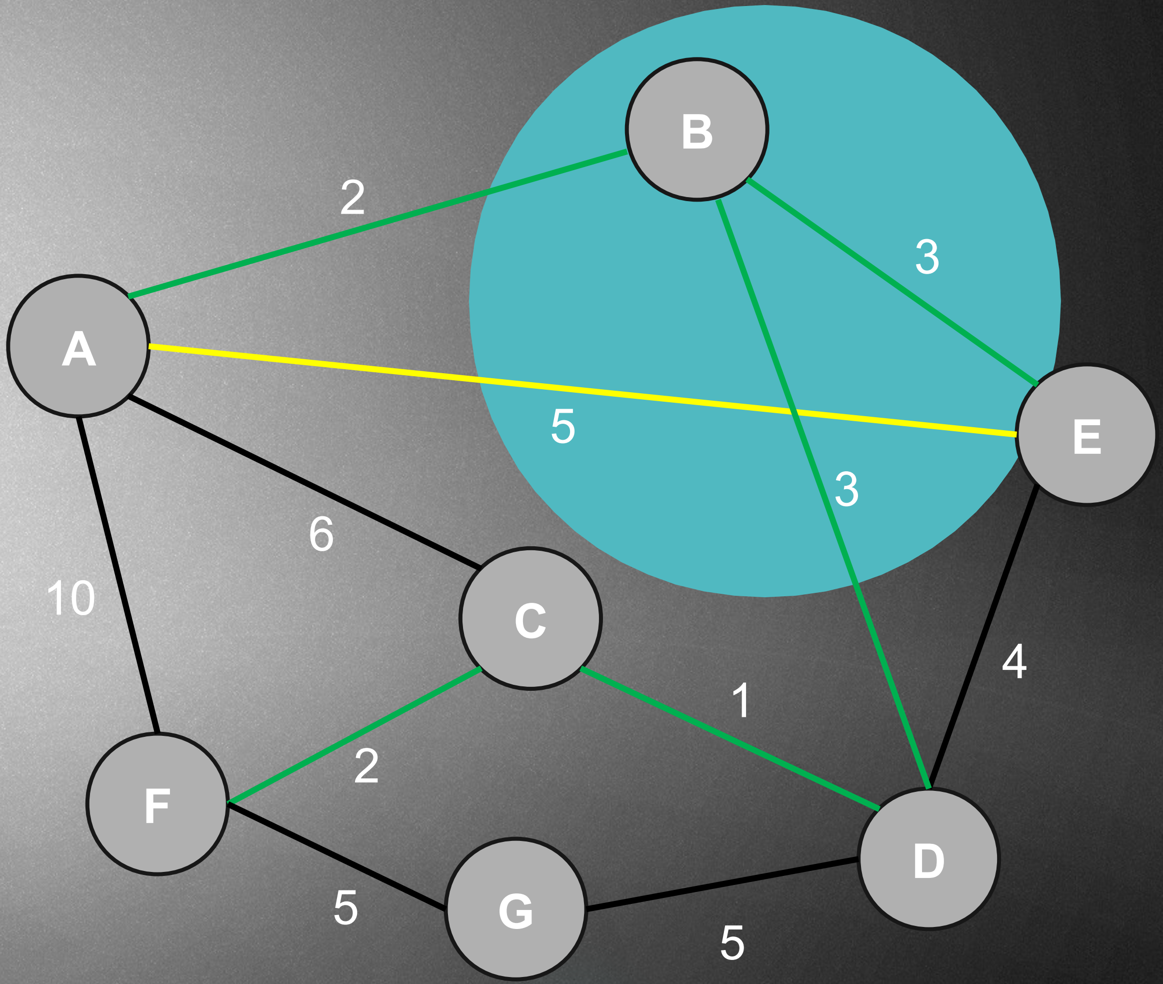
We have to sort the edges: 1, 2, 2, 3, 3, 4, 5, 5, 5, 6, 10



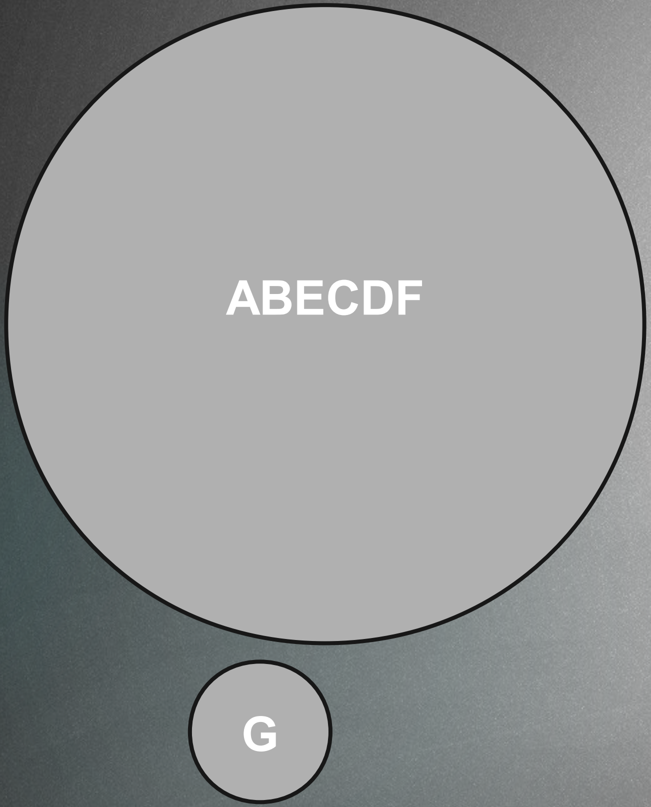


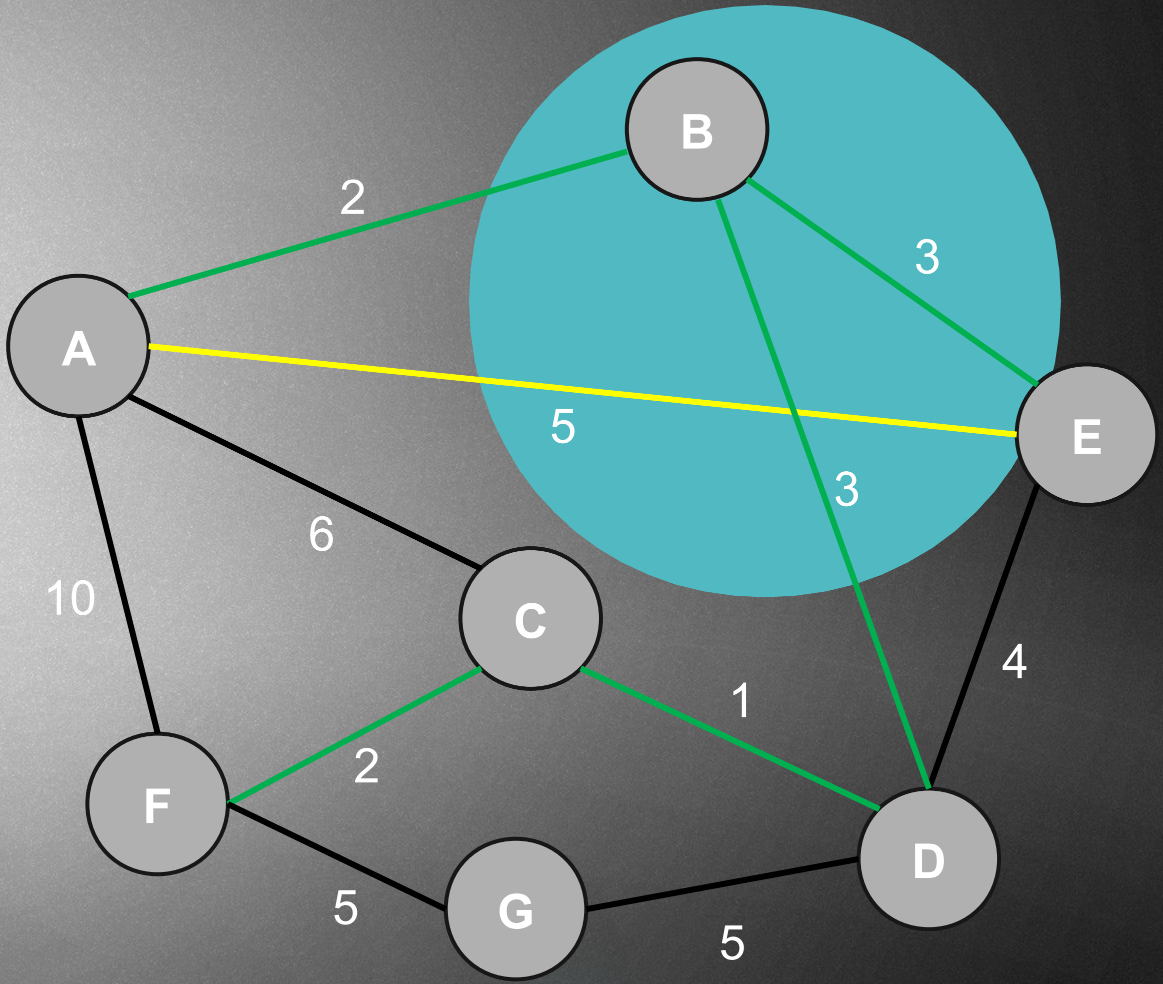
We have to sort the edges: 1, 2, 2, 3, 3, 4, 5, 5, 5, 6, 10



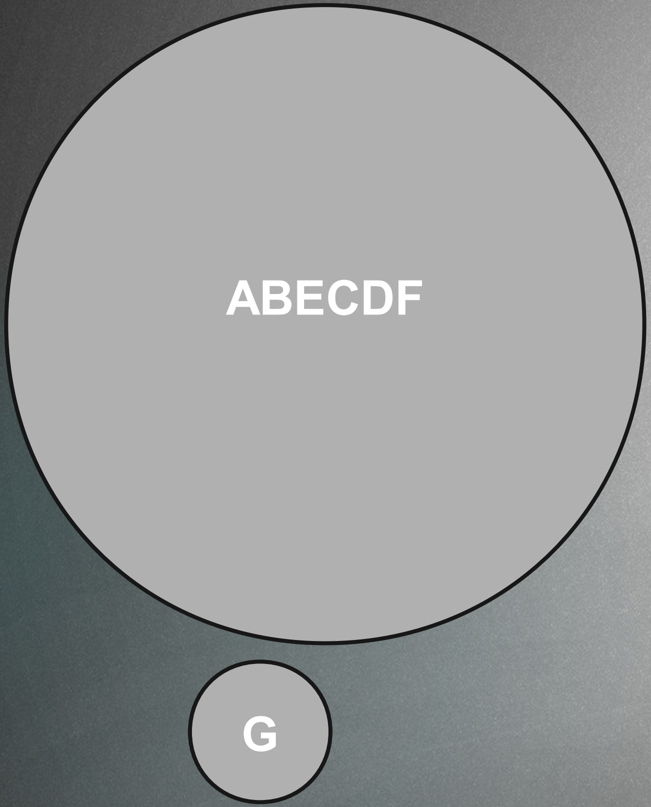


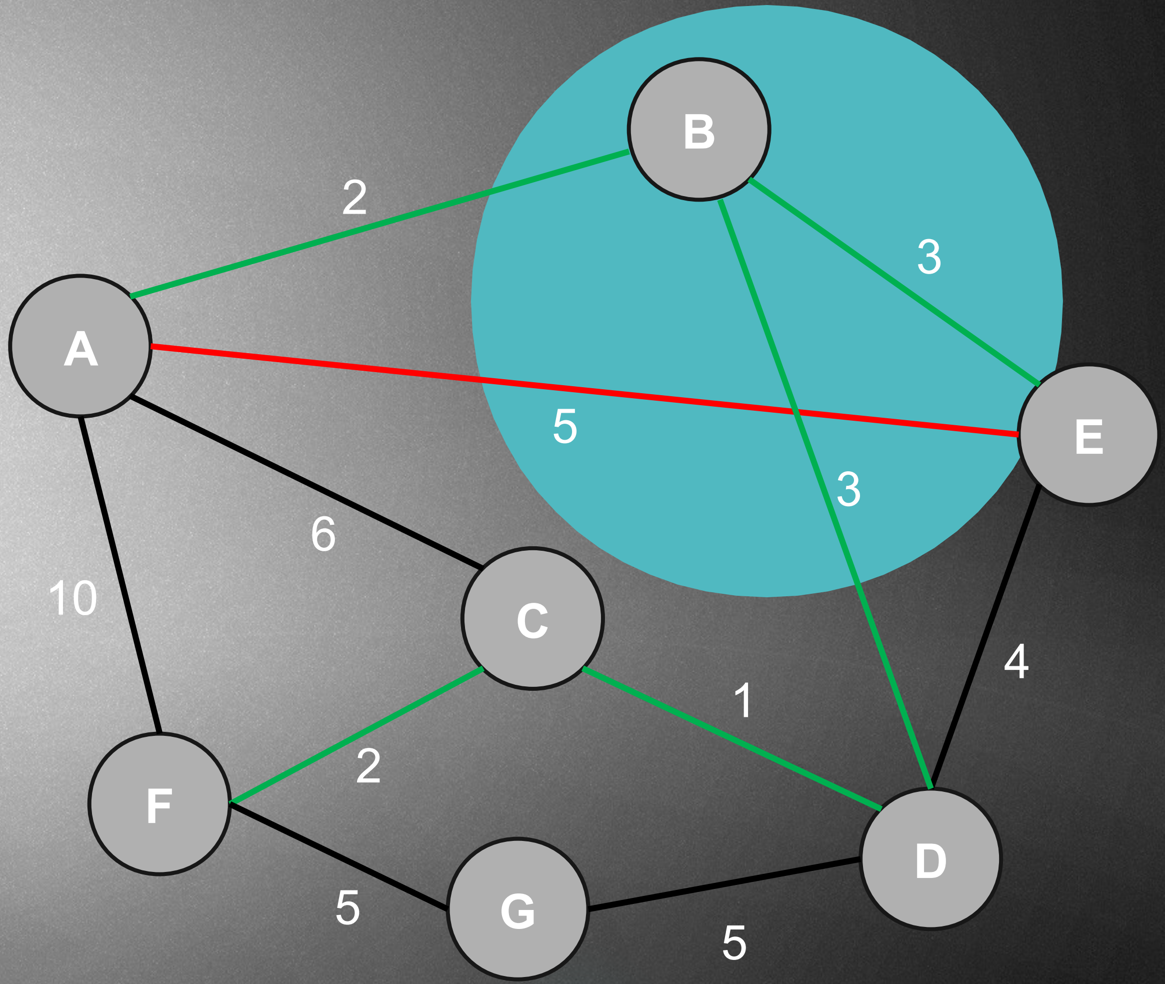
We have to sort the edges: 1, 2, 2, 3, 3, 4, 5, 5, 5, 6, 10



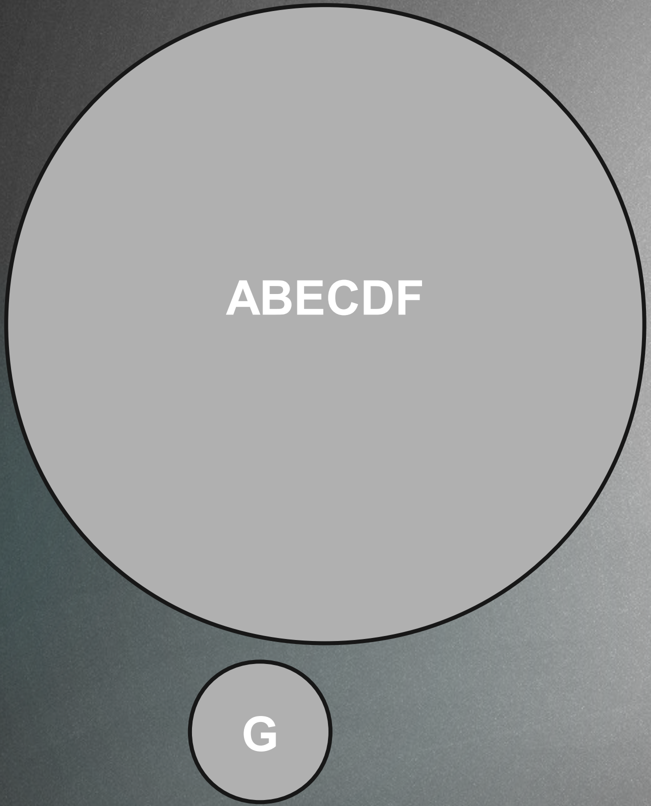


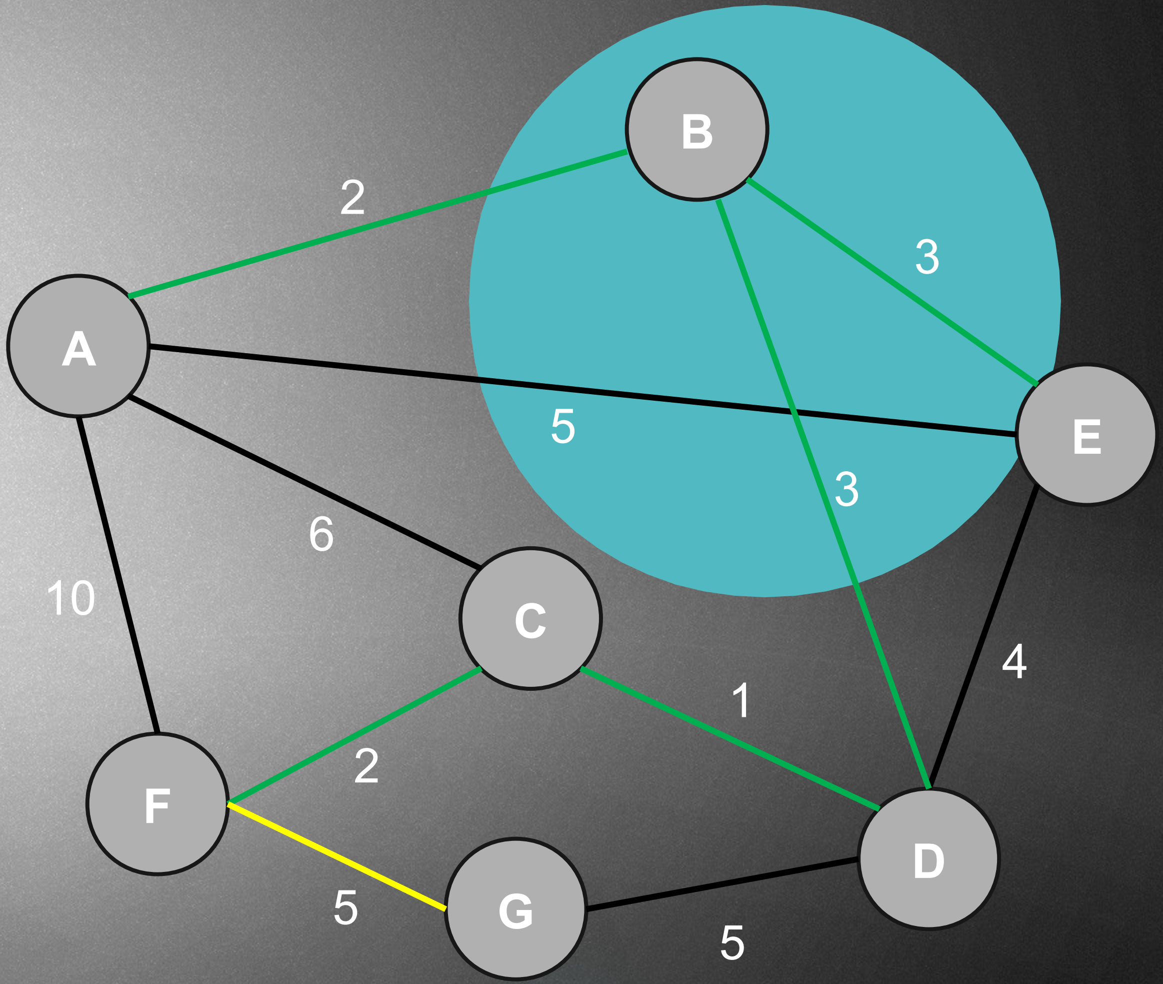
We have to sort the edges: 1, 2, 2, 3, 3, 4, 5, 5, 5, 6, 10



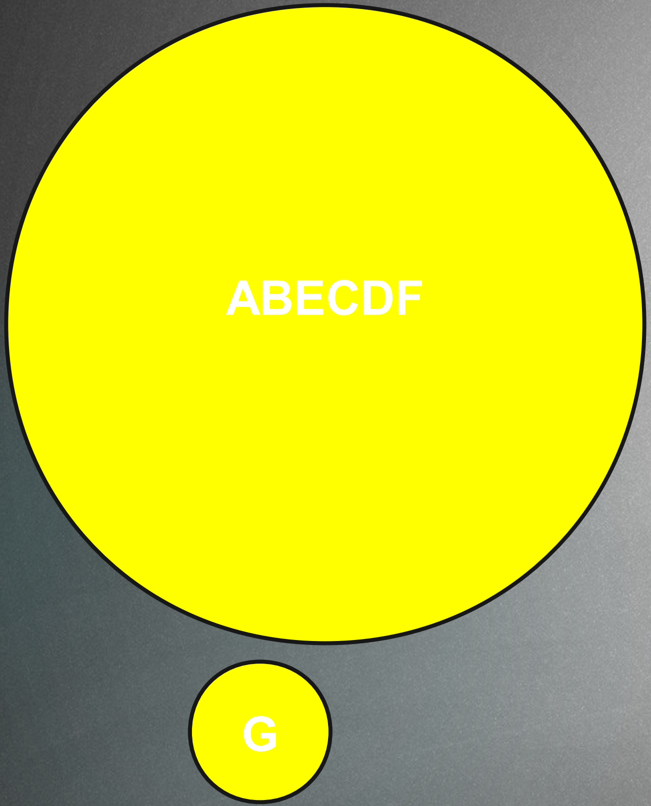


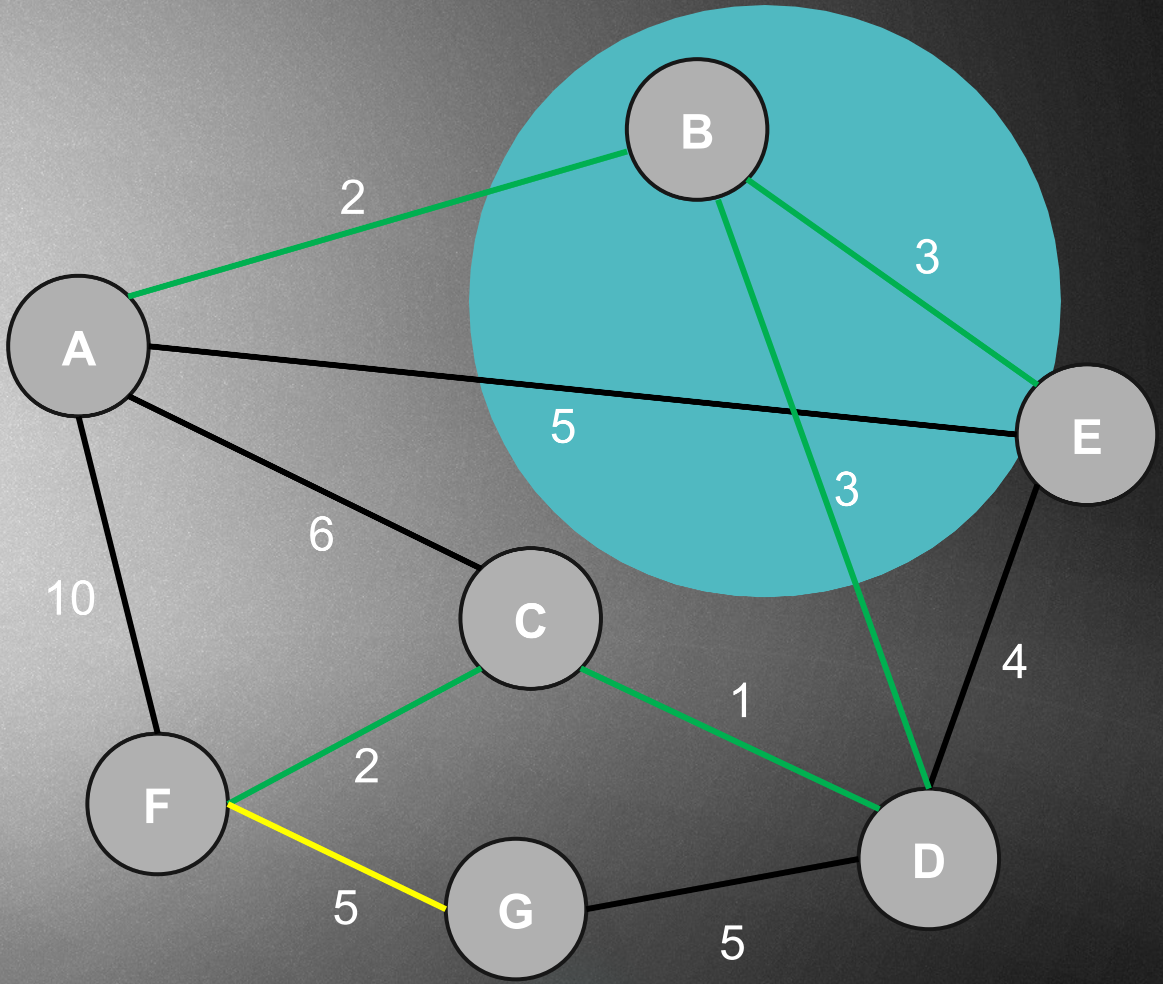
We have to sort the edges: 1, 2, 2, 3, 3, 4, 5, 5, 5, 6, 10



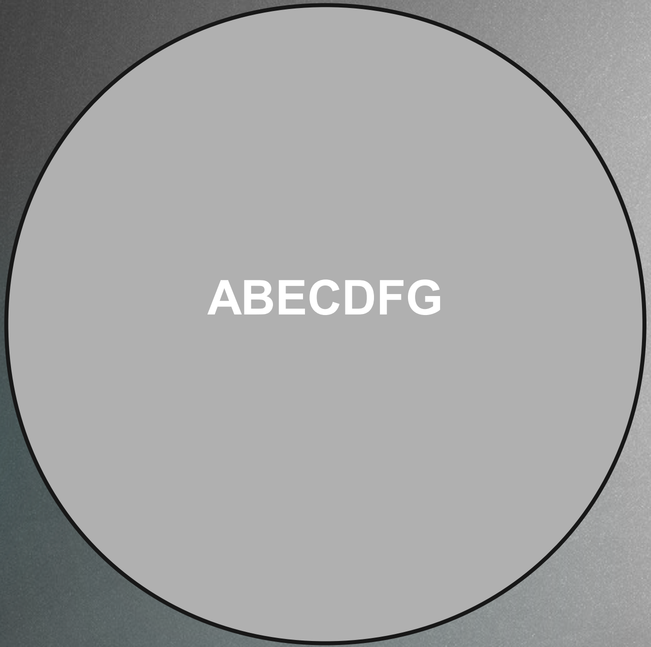


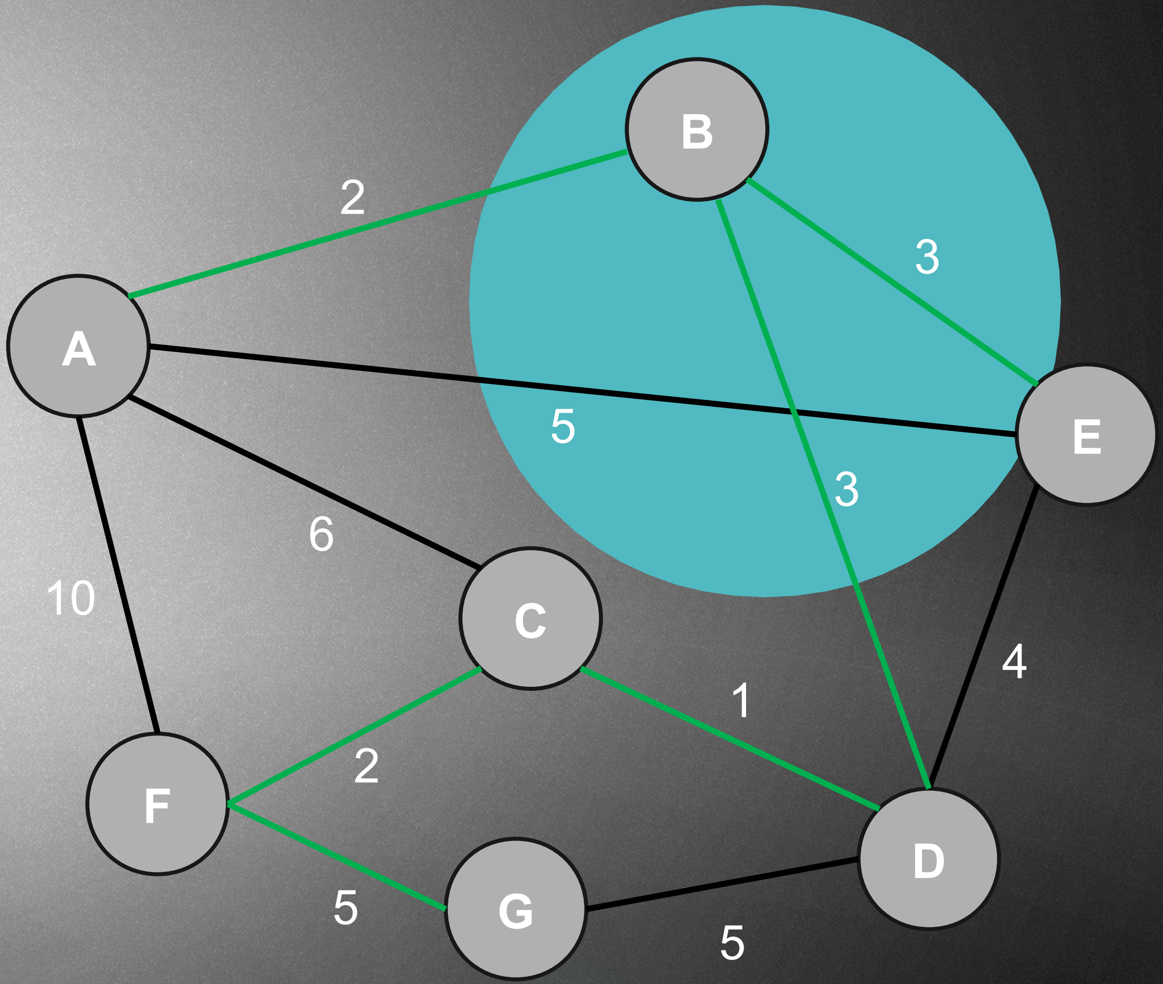
We have to sort the edges: 1, 2, 2, 3, 3, 4, 5, 5, 5, 6, 10



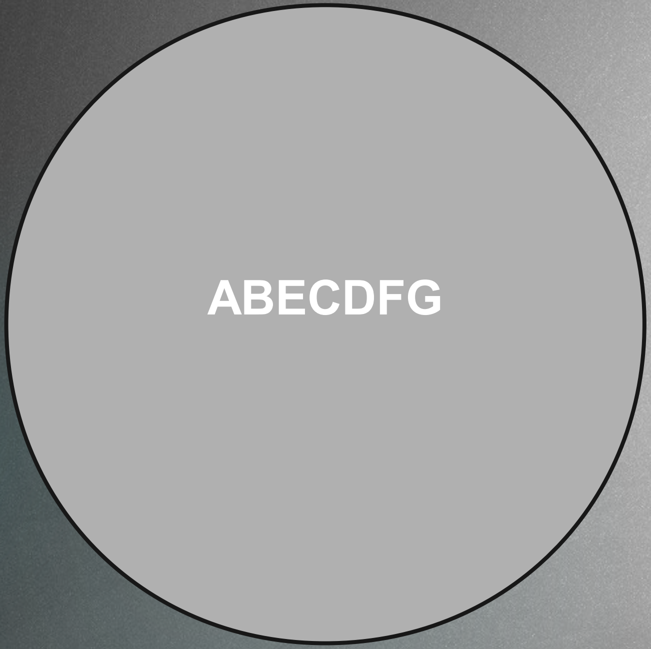


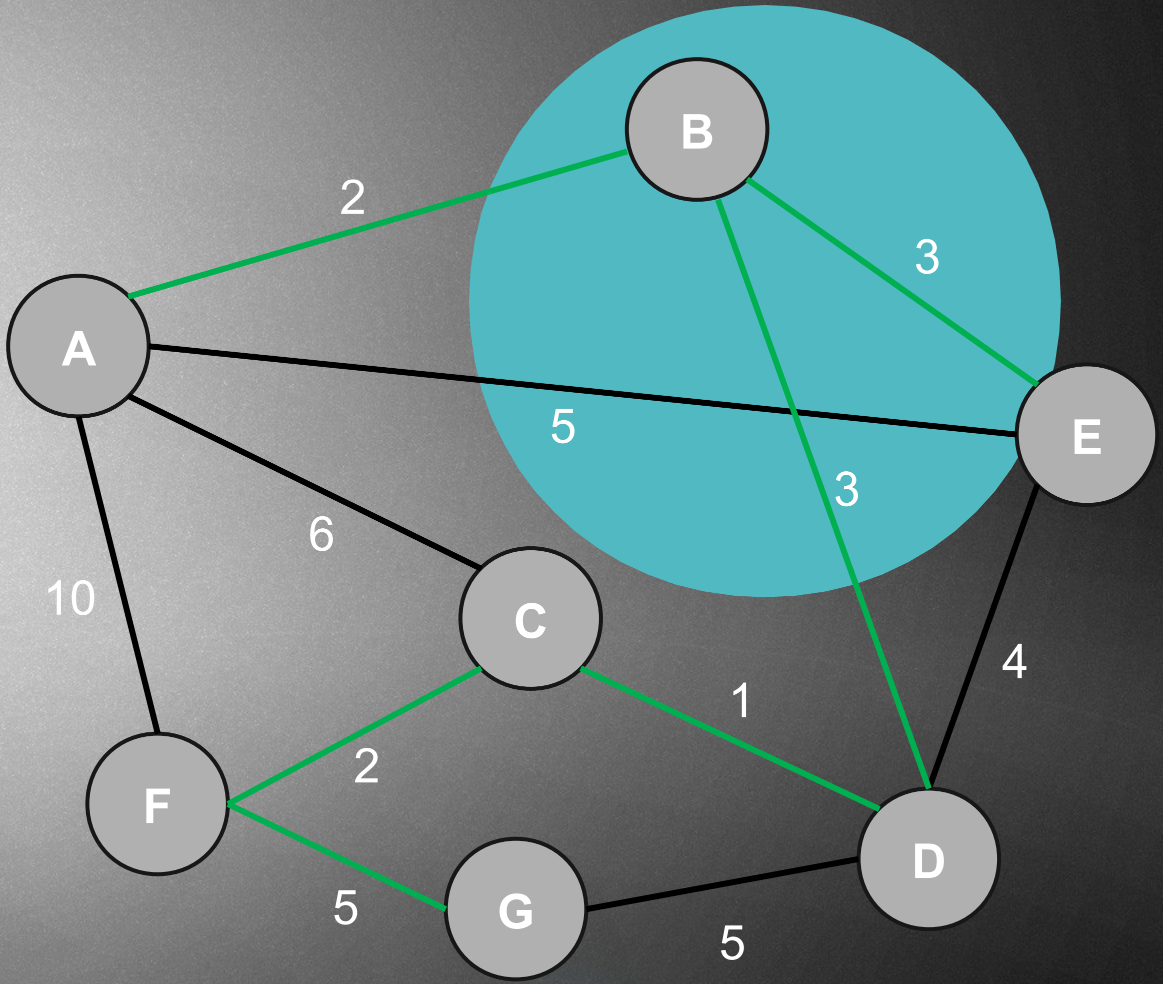
We have to sort the edges: 1, 2, 2, 3, 3, 4, 5, 5, 5, 6, 10





We have to sort the edges: 1, 2, 2, 3, 3, 4, 5, 5, 5, 6, 10





We have a single set so it is the end of the algorithm,
Minimal cost is: 2+3+3+1+2+5=16

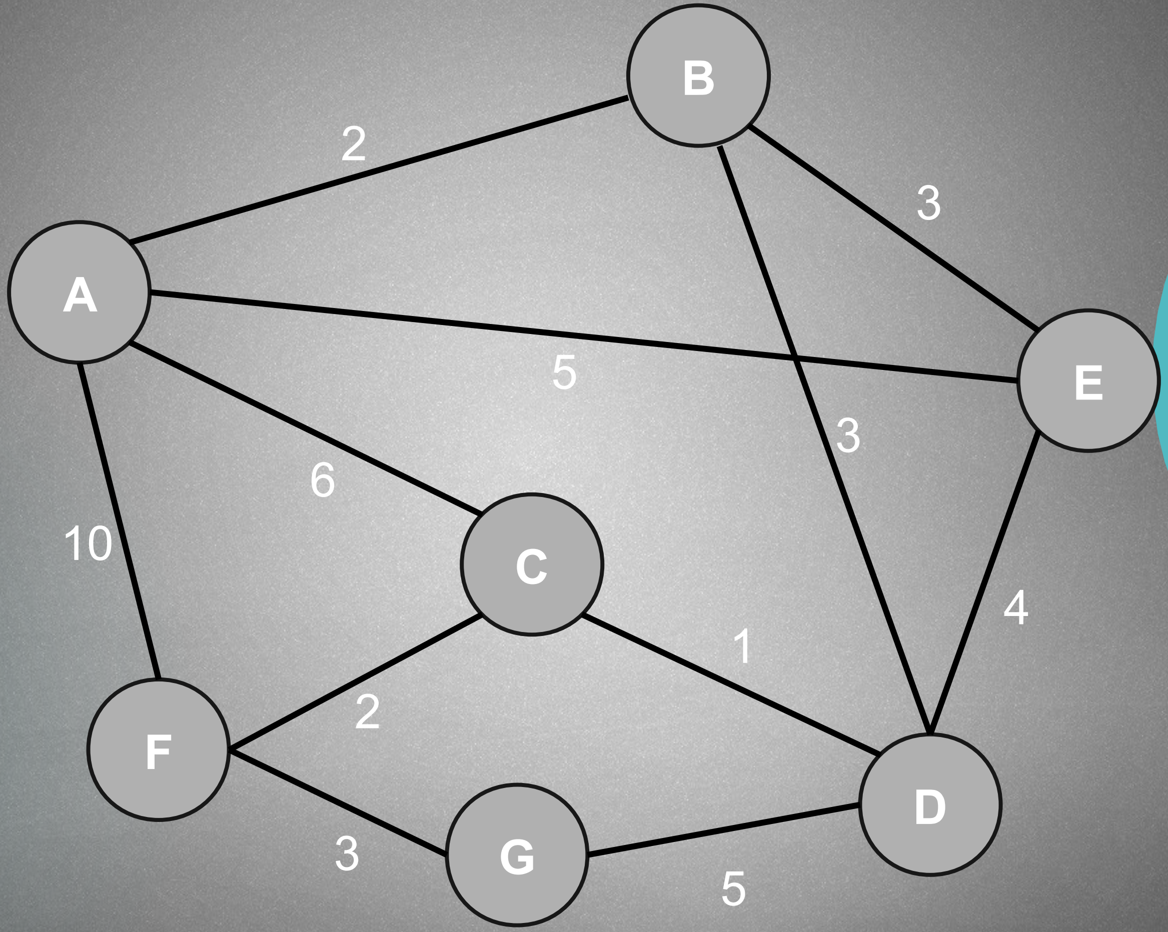
SPANNING TREES

PRIMS algorithm

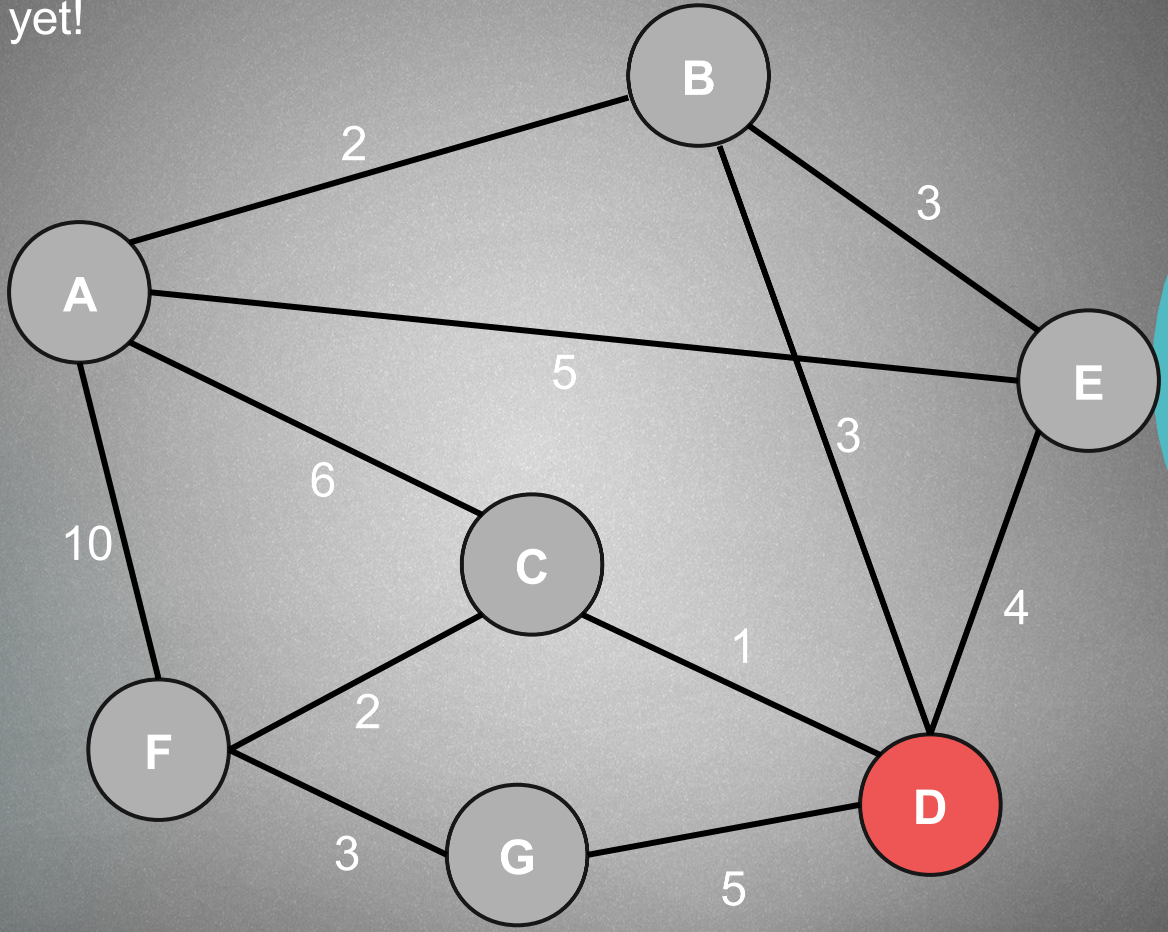
Prim-Jarnik algorithm

- In Kruskal implementation we build the spanning tree separately, adding the smallest edge to the spanning tree if there is no cycle
- In Prims algorithm we build the spanning tree from a given vertex, adding the smallest edge to the MST
- Kruskal → edge based   Prims → vertex based !!!
- There are two implementations: lazy and eager
- Lazy implementation: add the new neighbour edges to the heap without deleting its content
- Eager implementation: we keep updating the heap if the distance from a vertex to the MST has changed
- Average running time: **O(E*logE)** but we need additional memory space **O(E)**
- Worst case: **O(E*logV)**

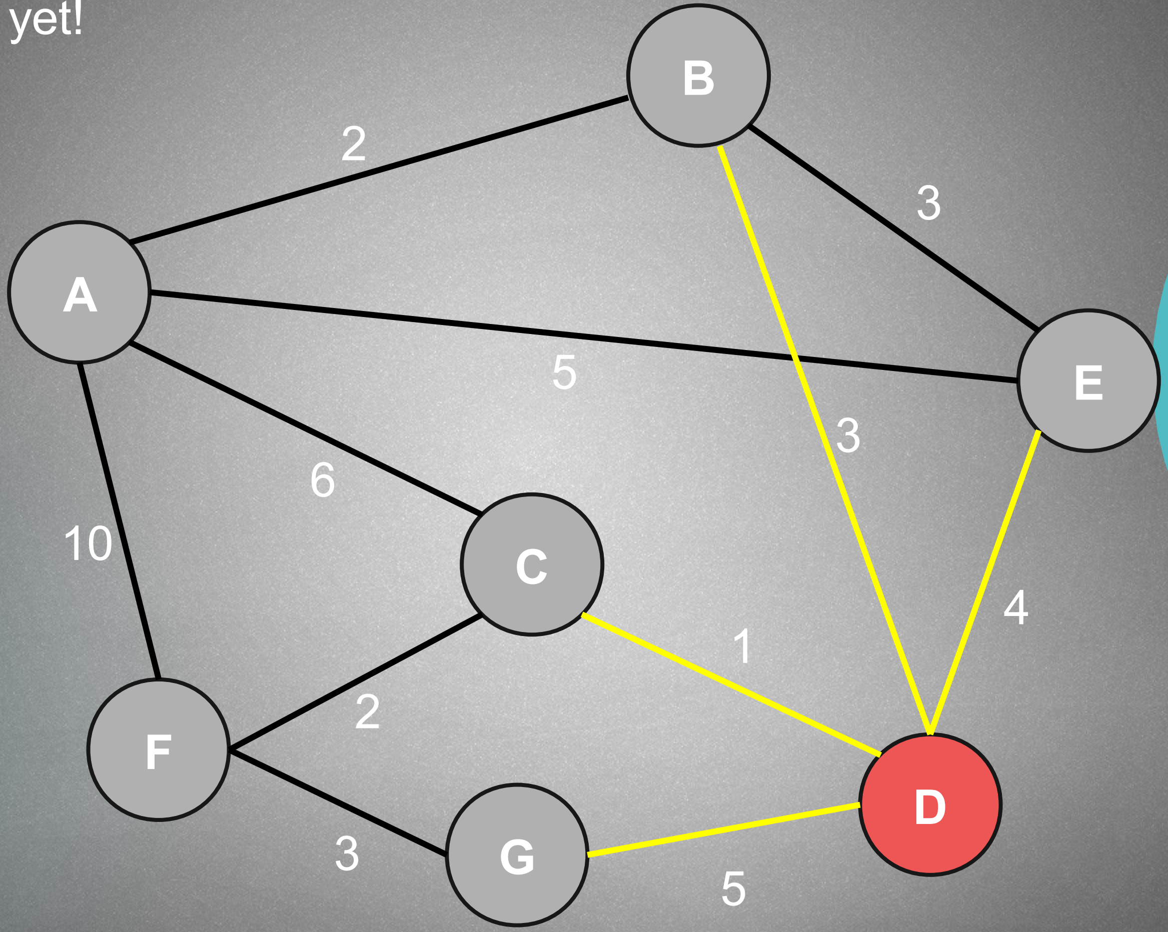
Start at a random vertex!



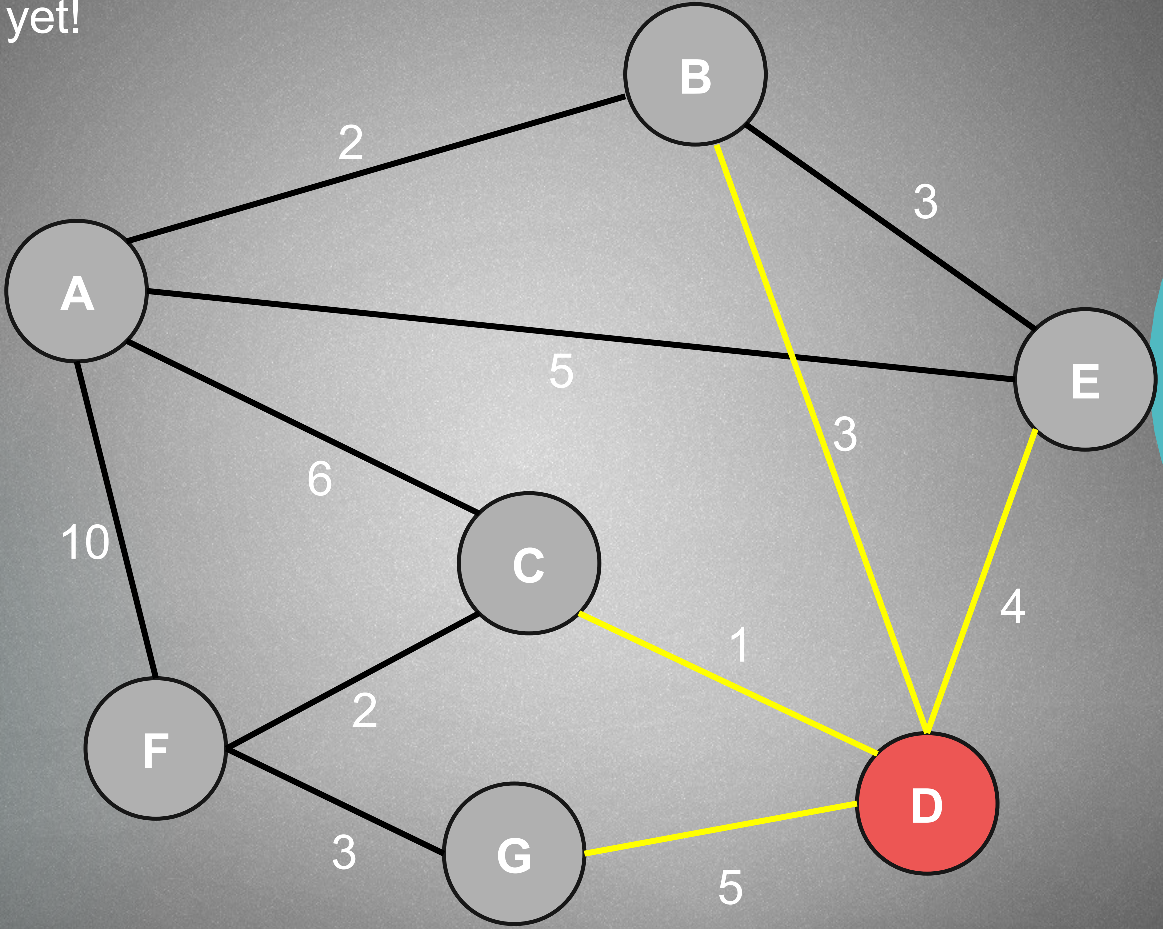
We consider edges to vertexes
we have not visited yet!



We consider edges to vertexes
we have not visited yet!



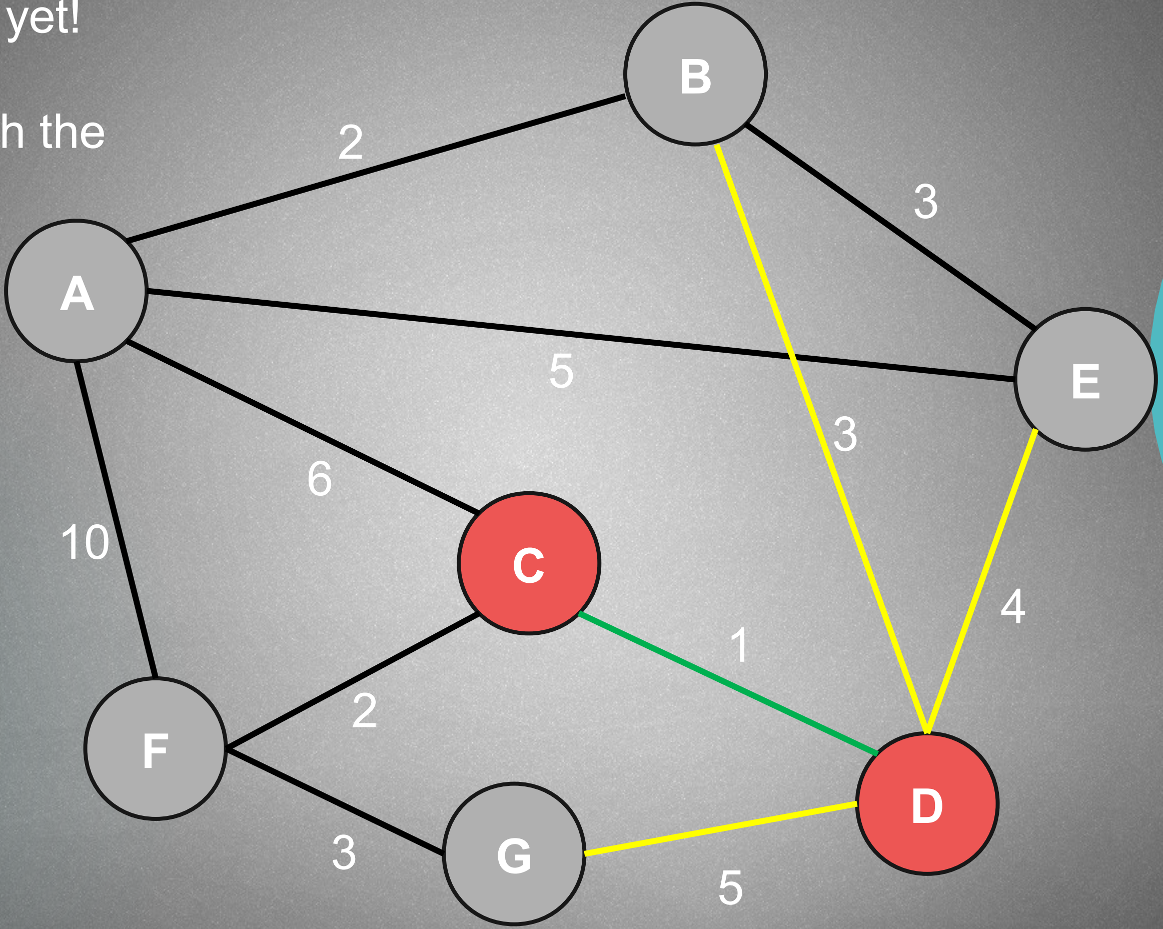
We consider edges to vertexes
we have not visited yet!



Heap content: G – 5 , C – 1 , B – 3 , E – 3

We consider edges to vertexes we have not visited yet!

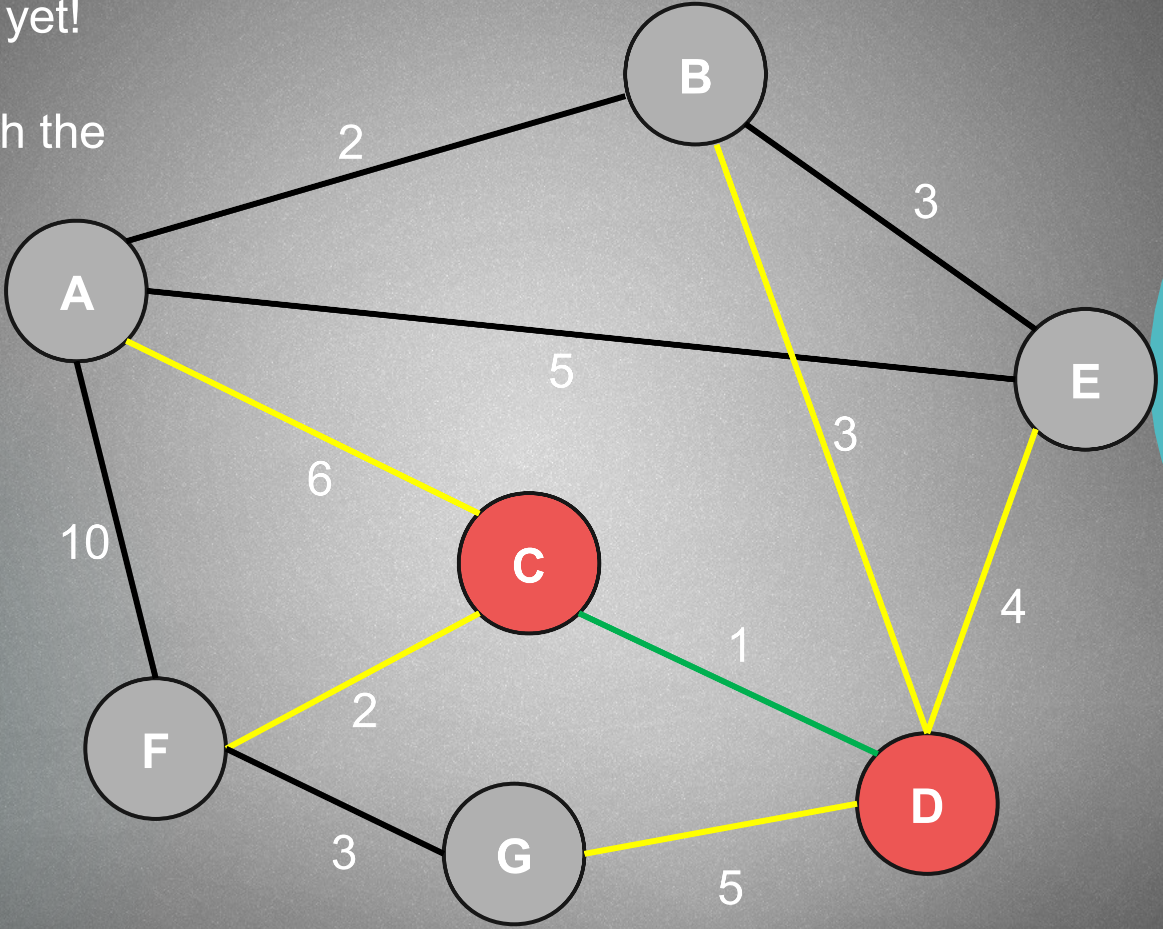
We pick the one with the lowest cost



Heap content: G – 5 , **C – 1** , B – 3 , E – 3

We consider edges to vertexes we have not visited yet!

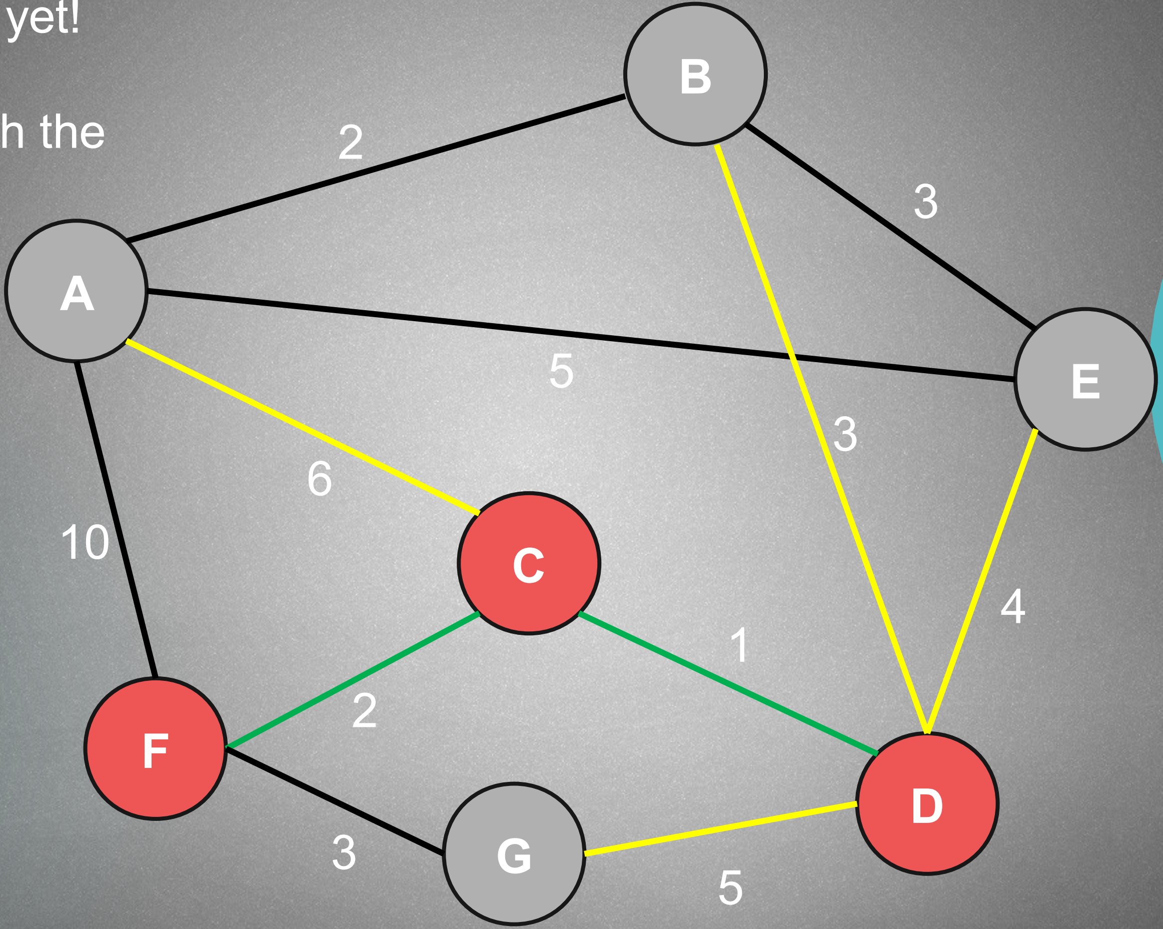
We pick the one with the lowest cost



Heap content: G – 5 , B – 3 , E – 3 , A – 6 , F – 2

We consider edges to vertexes we have not visited yet!

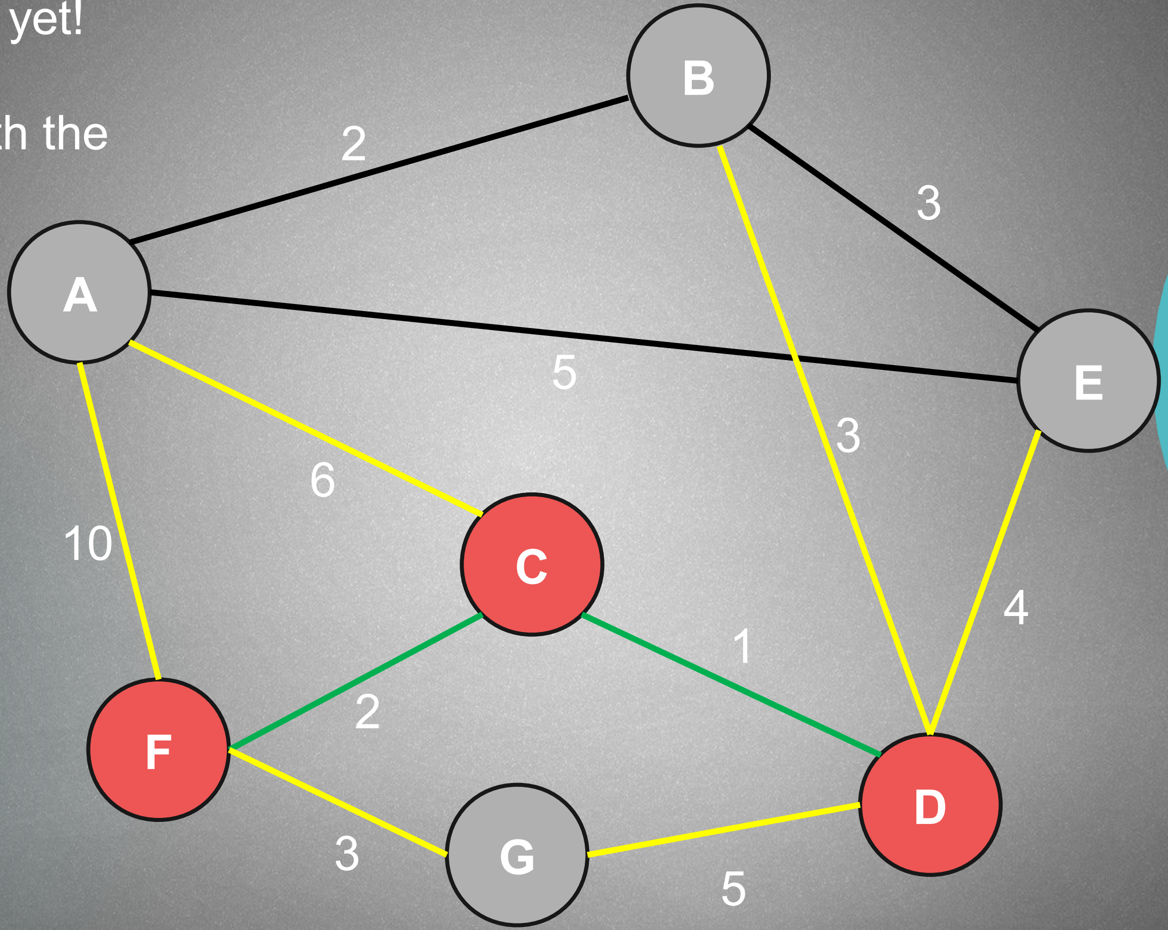
We pick the one with the lowest cost



Heap content: G – 5 , B – 3 , E – 3 , A – 6 , **F – 2**

We consider edges to vertexes we have not visited yet!

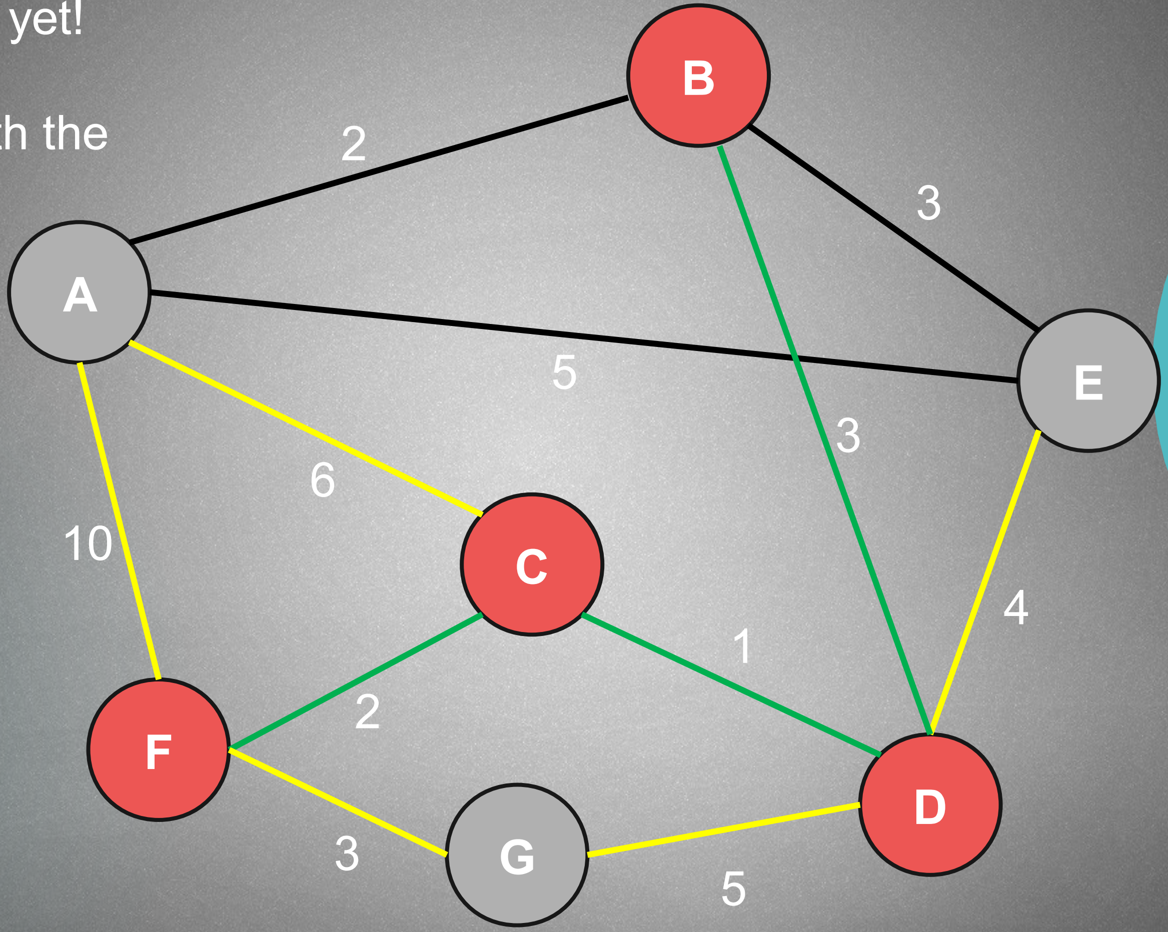
We pick the one with the lowest cost



Heap content: G – 5 , B – 3 , E – 3 , A – 6 , A – 10 , G – 3

We consider edges to vertexes we have not visited yet!

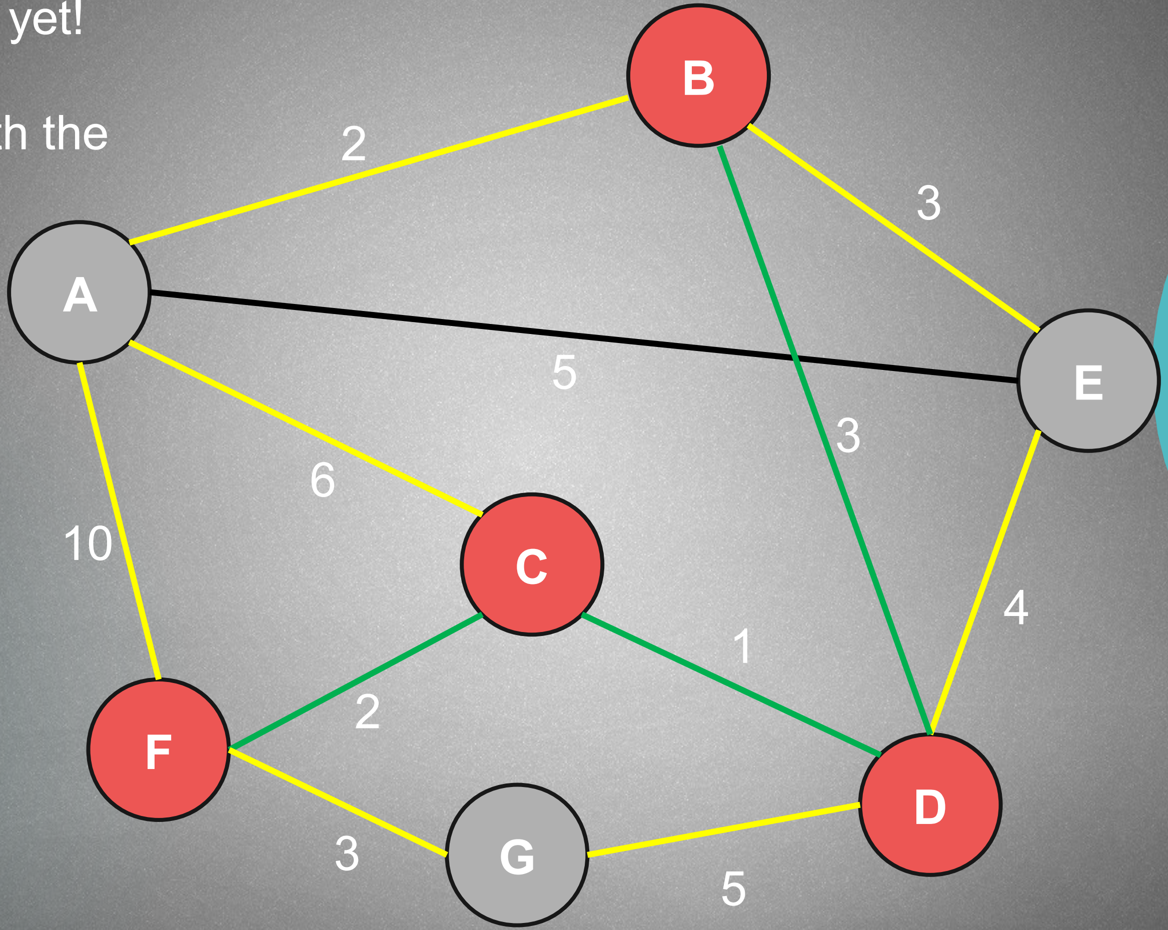
We pick the one with the lowest cost



Heap content: G – 5 , **B – 3** , E – 3 , A – 6 , A – 10 , G – 3

We consider edges to vertexes we have not visited yet!

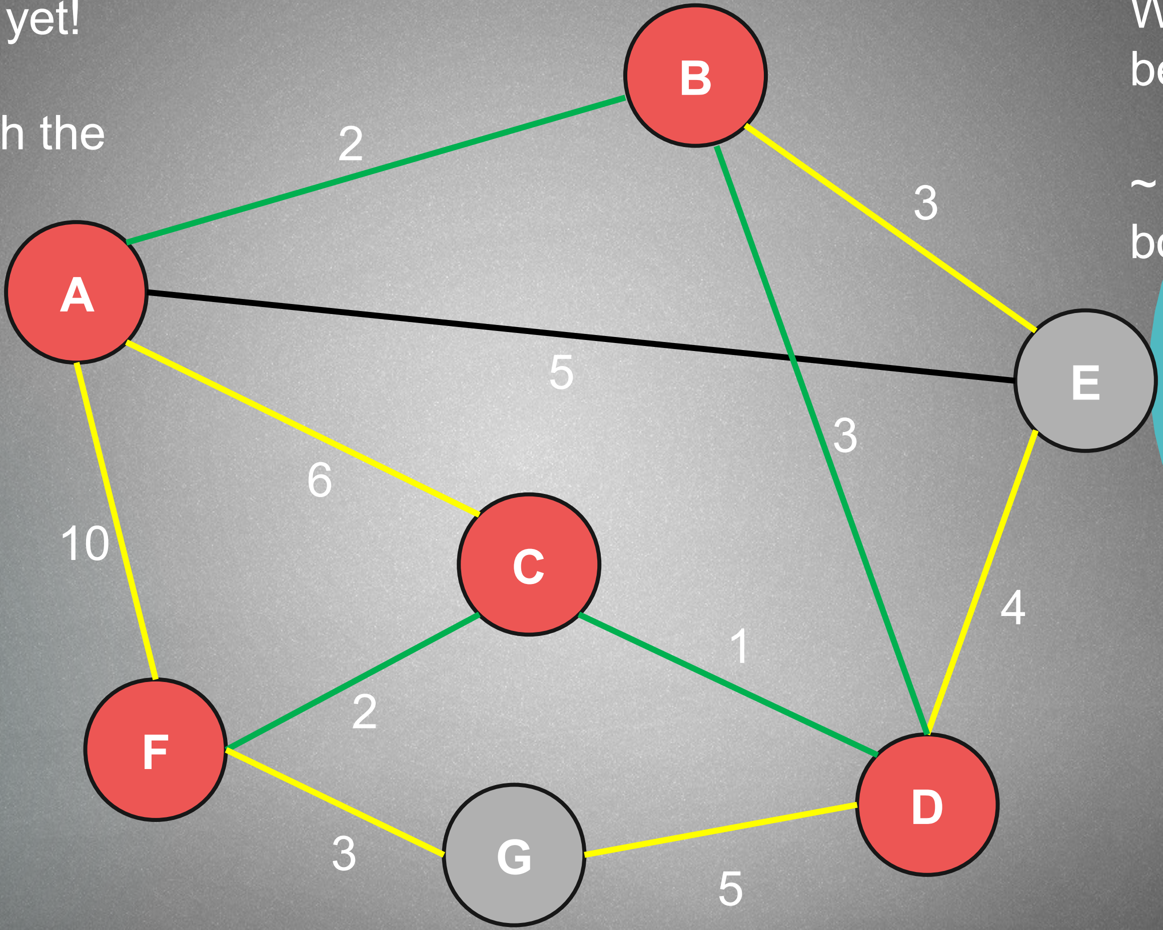
We pick the one with the lowest cost



Heap content: G – 5 , E – 3 , A – 6 , A – 10 , G – 3 , A – 2 , E – 3

We consider edges to vertexes we have not visited yet!

We pick the one with the lowest cost



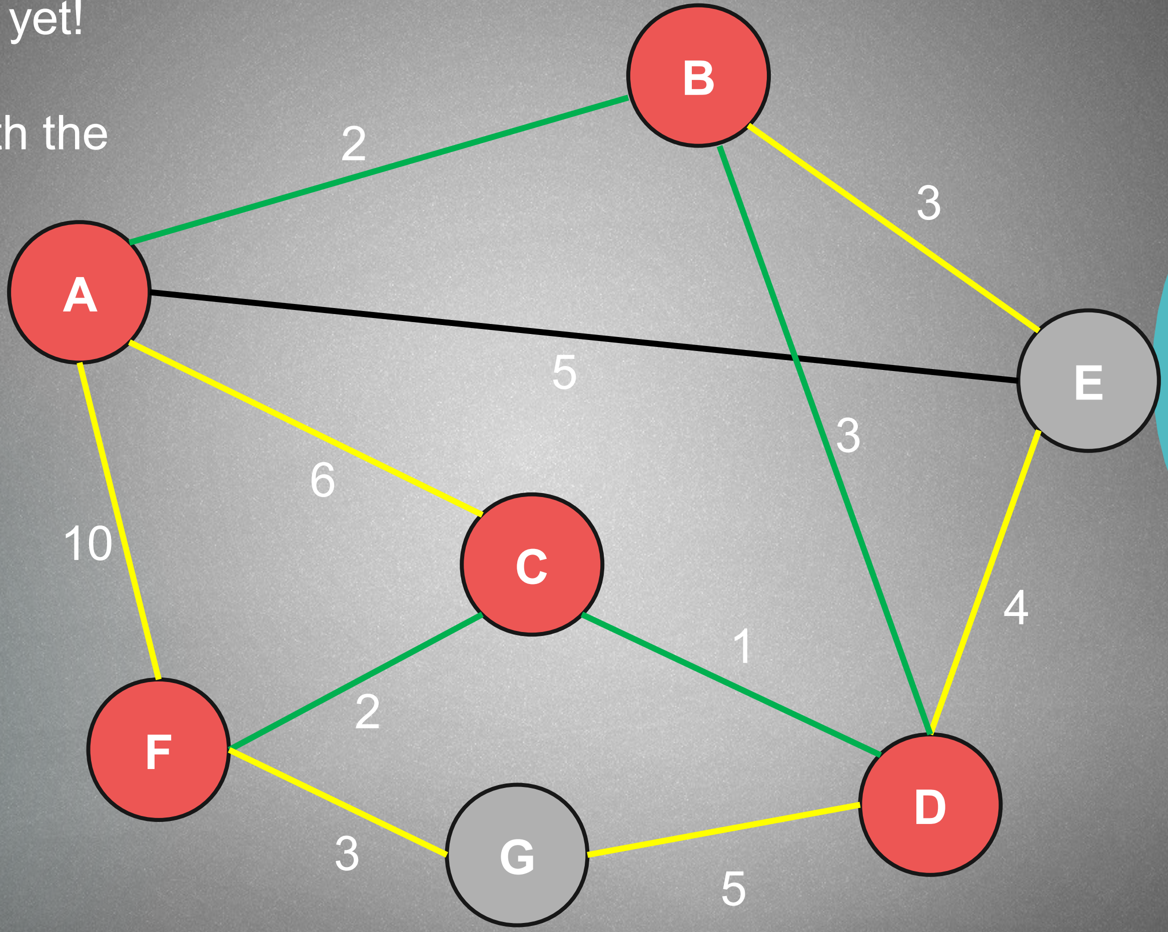
We do not consider edge between A and E

~ because we have considered both vertexes !!!

Heap content: G – 5 , E – 3 , A – 6 , A – 10 , G – 3 , A – 2 , E – 3

We consider edges to vertexes we have not visited yet!

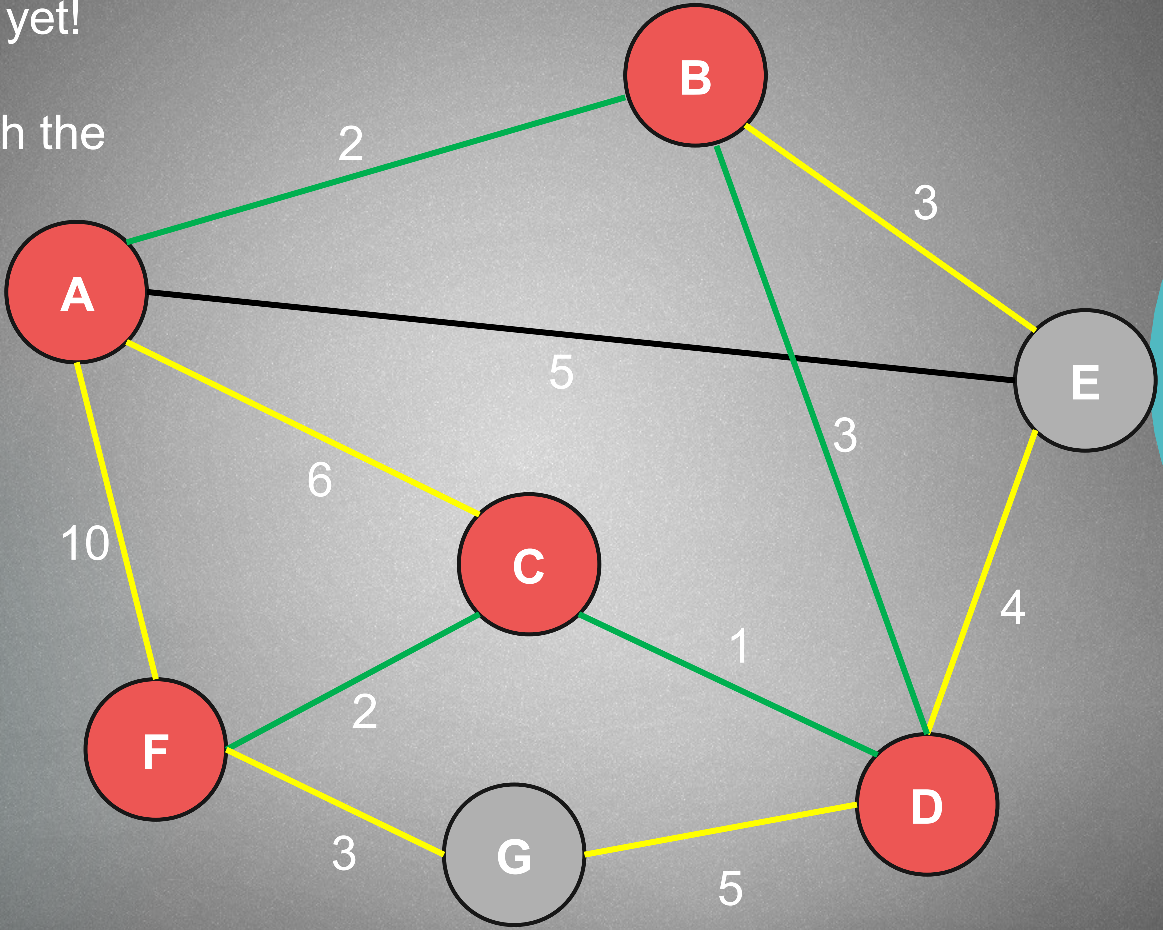
We pick the one with the lowest cost



Heap content: G – 5 , E – 3 , A – 6 , A – 10 , G – 3 , **A – 2** , E – 3

We consider edges to vertexes we have not visited yet!

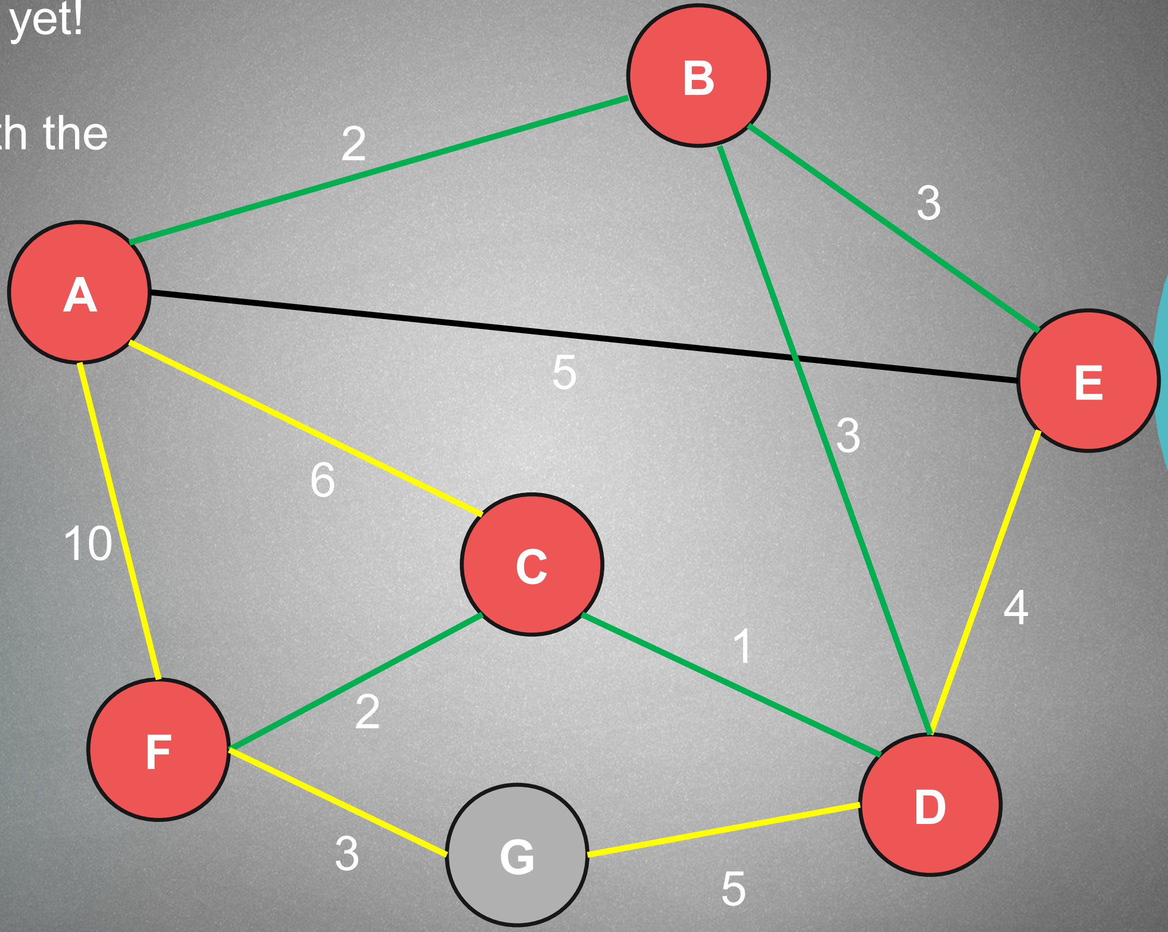
We pick the one with the lowest cost



Heap content: G – 5 , E – 3 , A – 6 , A – 10 , G – 3 , E – 3

We consider edges to vertexes we have not visited yet!

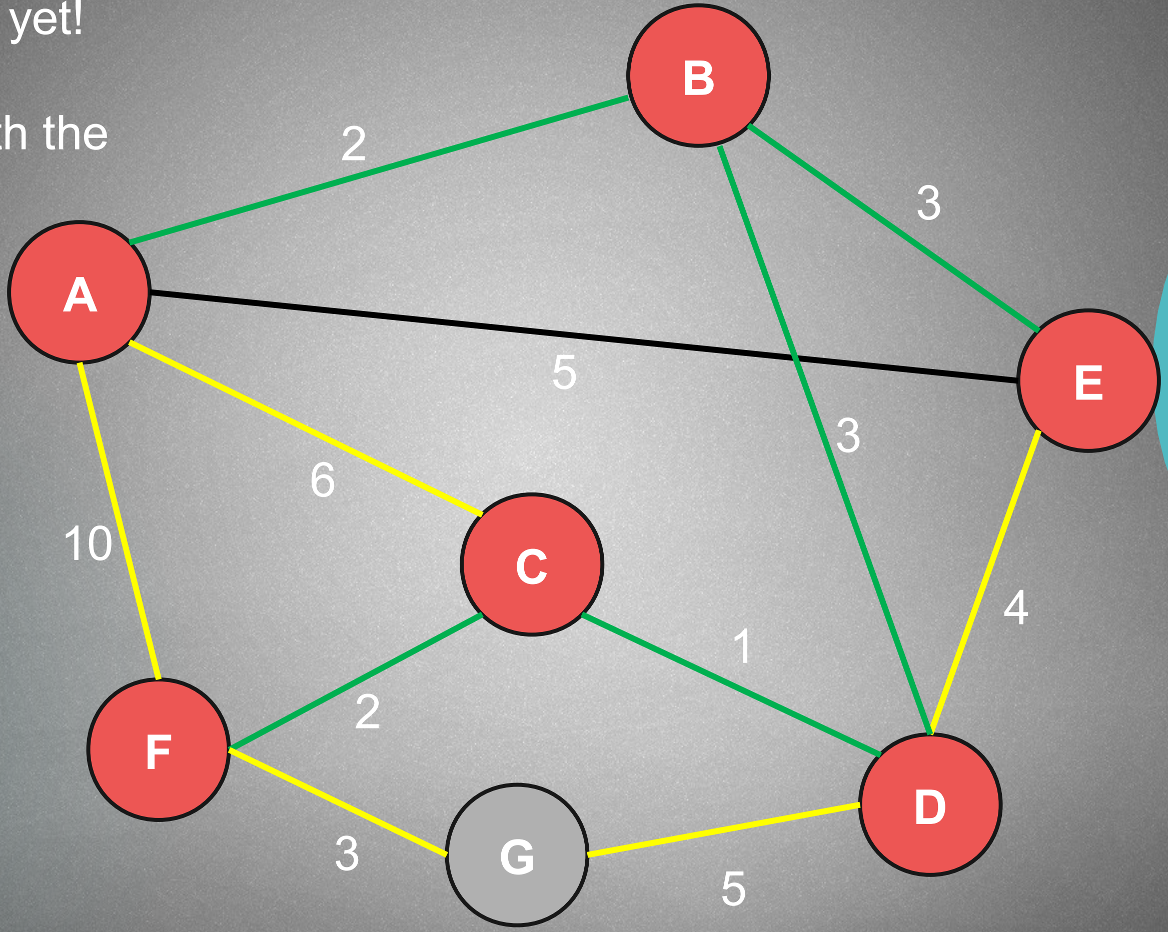
We pick the one with the lowest cost



Heap content: G – 5 , E – 3 , A – 6 , A – 10 , G – 3 , **E – 3**

We consider edges to vertexes we have not visited yet!

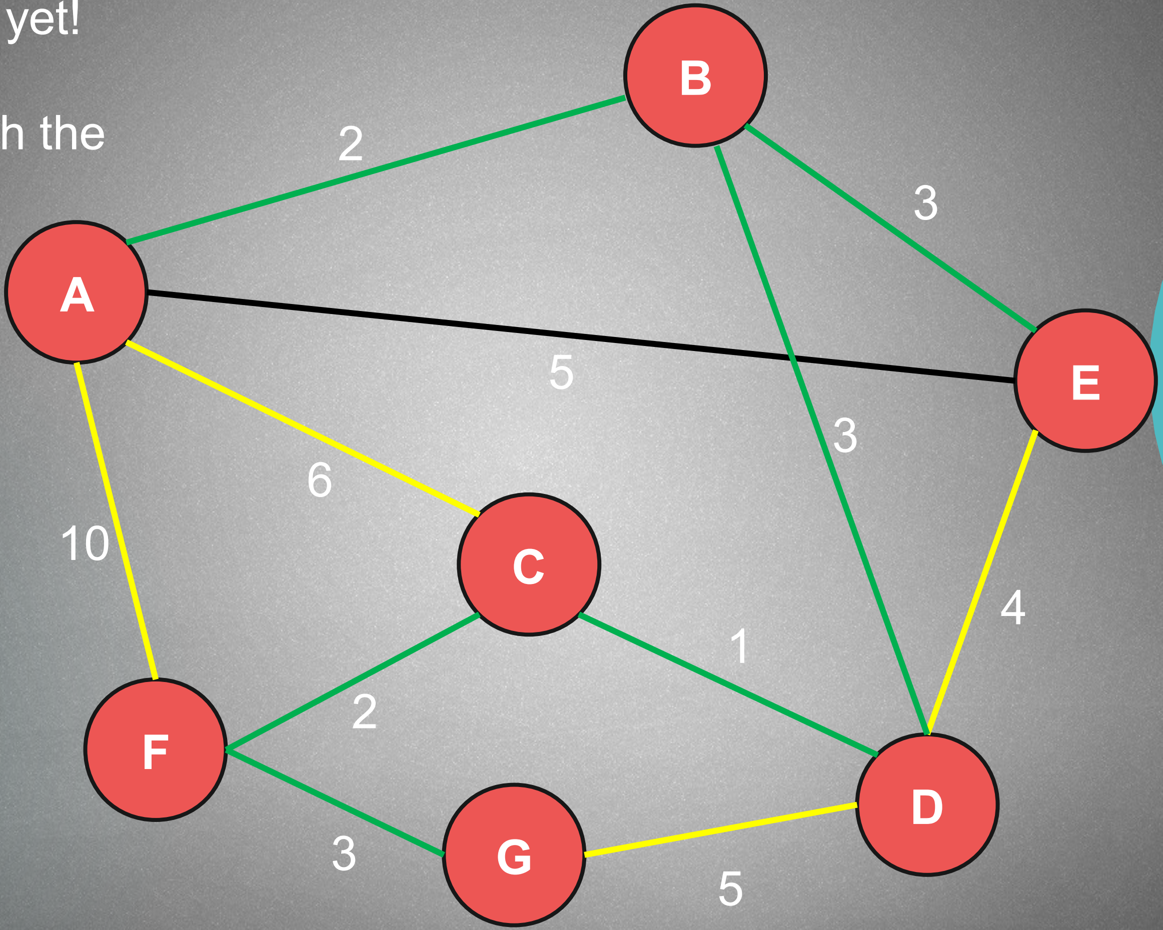
We pick the one with the lowest cost



Heap content: G – 5 , E – 3 , A – 6 , A – 10 , G – 3

We consider edges to vertexes we have not visited yet!

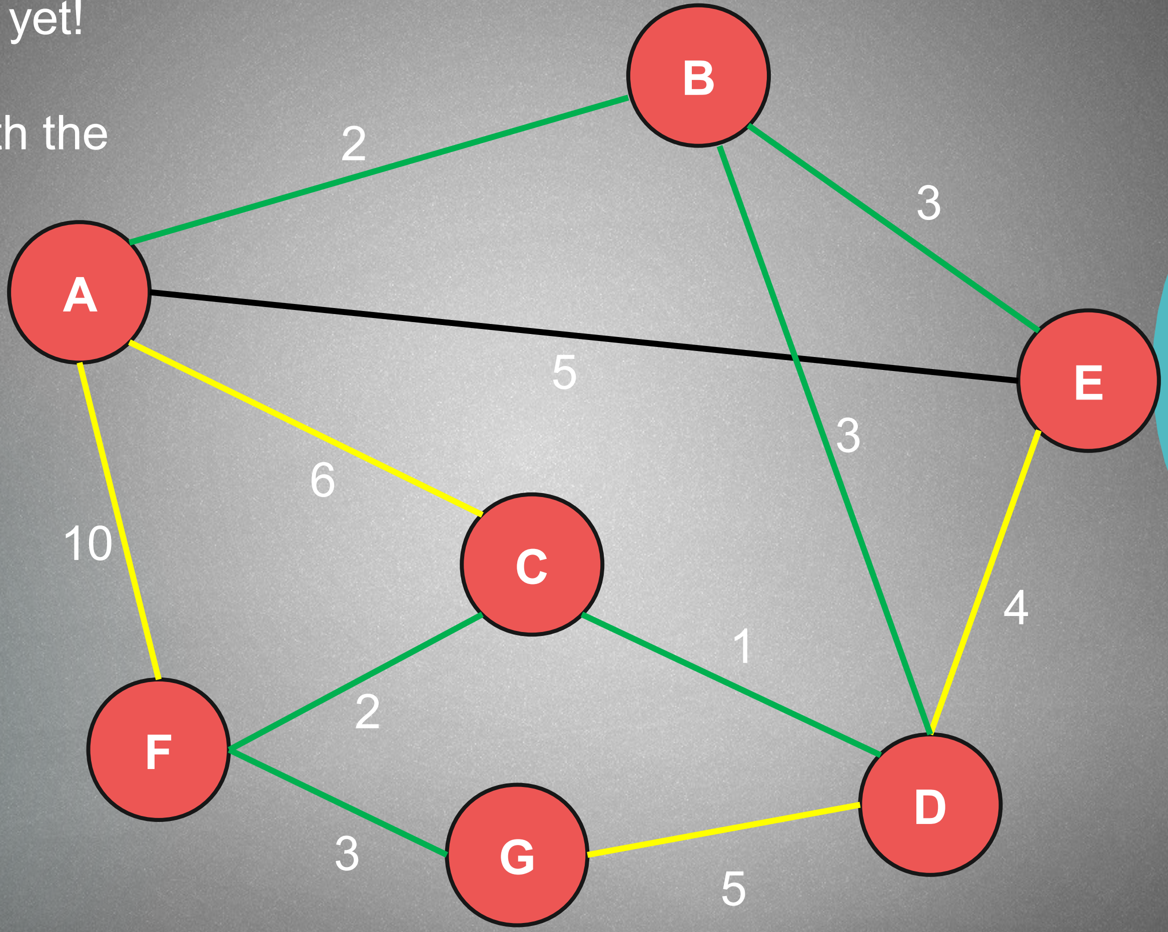
We pick the one with the lowest cost



Heap content: G – 5 , E – 3 , A – 6 , A – 10 , **G – 3**

We consider edges to vertexes we have not visited yet!

We pick the one with the lowest cost



We have visited all the nodes, so we are done! The minimum cost: 14

Prims VS Kruskal

- Prim's algorithm is significantly faster in the limit when you've got a really dense graph with many more edges than vertices
- Kruskal performs better in typical situations (sparse graphs) because it uses simpler data structures
- Kruskal can have better performance if the edges can be sorted in linear time or the edges are already sorted
- Prim's better if the number of edges to vertices is high (dense graphs)

Applications

- K means
- Telecommunications
- Recommendation systems

K-means clustering

- Important unsupervised machine learning technique
- We have a dataset: we want to find patterns !!!
- For examle: want to classify and cluster the points in the 2D plane
- We construct a minimal spanning tree –> leave the first k edge
- Result: we will have k clusters !!!