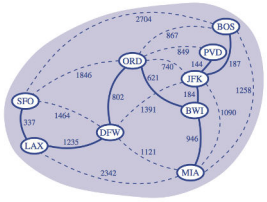
Minimum Spanning Trees



1

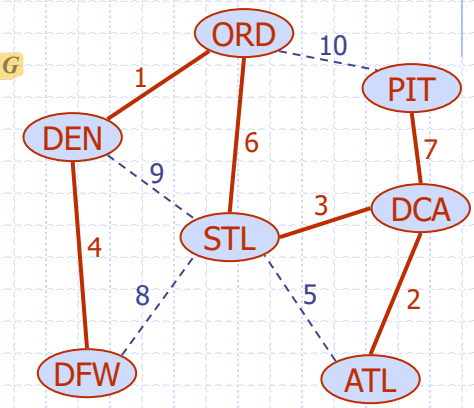
Minimum Spanning Trees

Spanning subgraph
- Subgraph of a graph G containing all the vertices of G

Spanning tree
- Spanning subgraph that is itself a tree

Minimum spanning tree (MST)
- Spanning tree of a weighted graph with minimum total edge weight

Applications
- Communications networks
- Transportation networks



Non ha loop ed è minimo nella somma dei pesi degli archi.

2

Non significa scegliere il percorso più breve da un nodo all'altro.
Non necessariamente allargo subito scoprendo minimo se prendo distanze minime.
Voglio una media complessiva bassa dei tratti il percorso che posso seguire

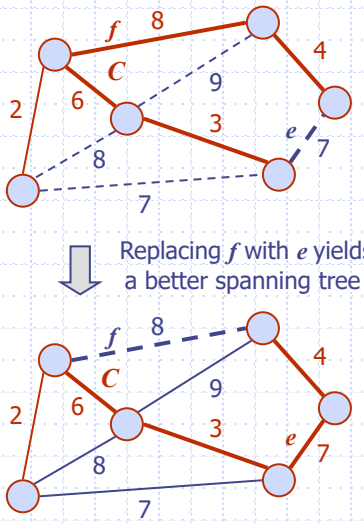
Cycle Property

Cycle Property:

- Let T be a minimum spanning tree of a weighted graph G
- Let e be an edge of G that is not in T and C let be the cycle formed by e with T
- For every edge f of C, $weight(f) \leq weight(e)$

Proof:

- By contradiction
- If $weight(f) > weight(e)$ we can get a spanning tree of smaller weight by replacing e with f



3

NON PUÒ ACCADERE CHE POSSO MIGLIORARE PARTIZIONE

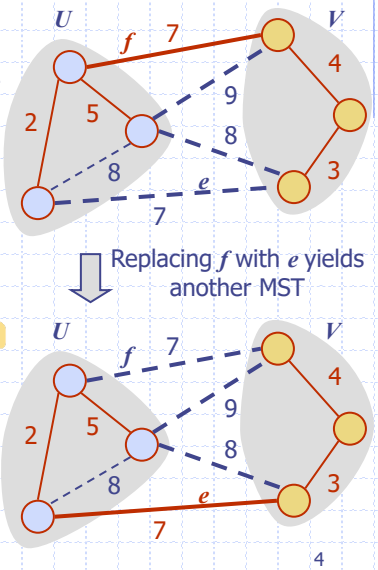
Partition Property

Partition Property:

- Consider a partition of the vertices of G into subsets U and V
- Let e be an edge of minimum weight across the partition
- There is a minimum spanning tree of G containing edge e

Proof:

- Let T be an MST of G
- If T does not contain e, consider the cycle C formed by e with T and let f be an edge of C across the partition
- By the cycle property, $weight(f) \leq weight(e)$
- Thus, $weight(f) = weight(e)$
- We obtain another MST by replacing f with e



4

f deve essere tra le partizioni.

PARTIZIONE: Sottografo che contiene sottoset dei vertici di partenza ma che per me è già un albero ricoprente minimo per quella partizione. Esiste un altro che collega le 2 partizioni? Arco più piccolo è quello che non forma albero ricoprente minimo.

Qui vado a salvare nel vertice il peso dell'arco più piccolo della nuvola collegato a quel vertice

Prim-Jarnik's Algorithm

- Similar to Dijkstra's algorithm
- We pick an arbitrary vertex s and we grow the MST as a cloud of vertices, starting from s
- We store with each vertex v label $d(v)$ representing the smallest weight of an edge connecting v to a vertex in the cloud
- At each step:
  - We add to the cloud the vertex u outside the cloud with the smallest distance label
  - We update the labels of the vertices adjacent to u

5

Prim-Jarnik Pseudo-code

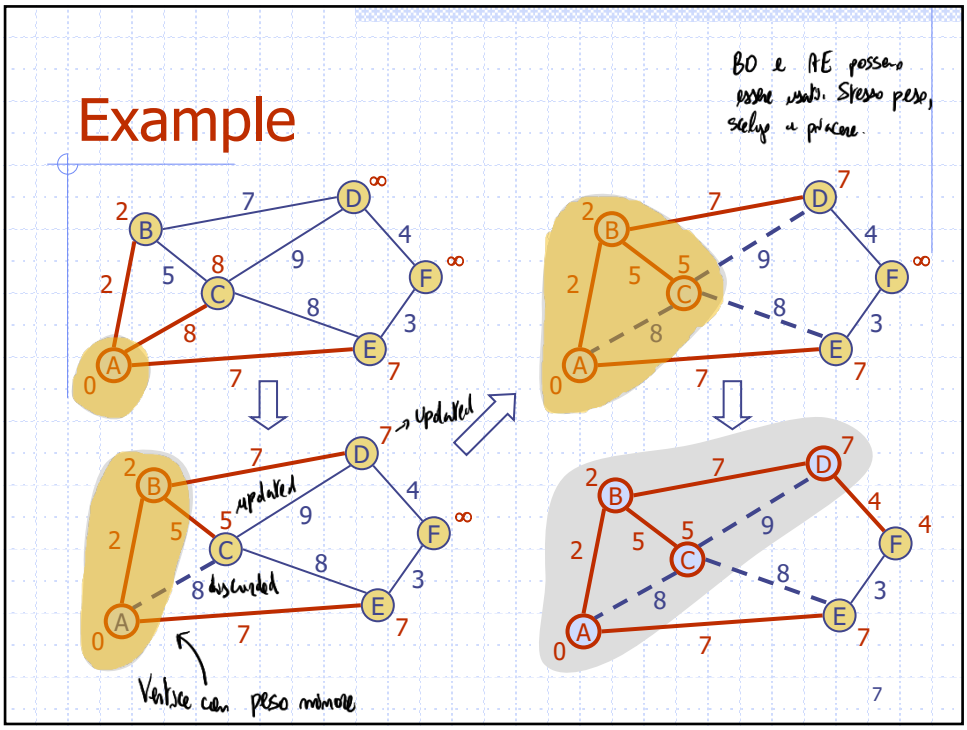
Algorithm PrimJarnik(G):
  Input: An undirected, weighted, connected graph G with n vertices and m edges
  Output: A minimum spanning tree T for G
  Pick any vertex s of G
  $D[s] = 0$
  for each vertex $v \neq s$ **do**
    $D[v] = \infty$
  Initialize $T = \emptyset$.
  Initialize a priority queue Q with an entry $(D[v], (v, \text{None}))$ for each vertex v, where $D[v]$ is the key in the priority queue, and (v, None) is the associated value.
  while Q is not empty **do**
    $(u, e) =$ value returned by Q.remove_min()
    Connect vertex u to T using edge e.
    for each edge $e' = (u, v)$ such that v is in Q **do**
      {check if edge (u, v) better connects v to T}
      if $w(u, v) < D[v]$ **then**
        $D[v] = w(u, v)$
        Change the key of vertex v in Q to $D[v]$.
        Change the value of vertex v in Q to (v, e').
  return the tree T

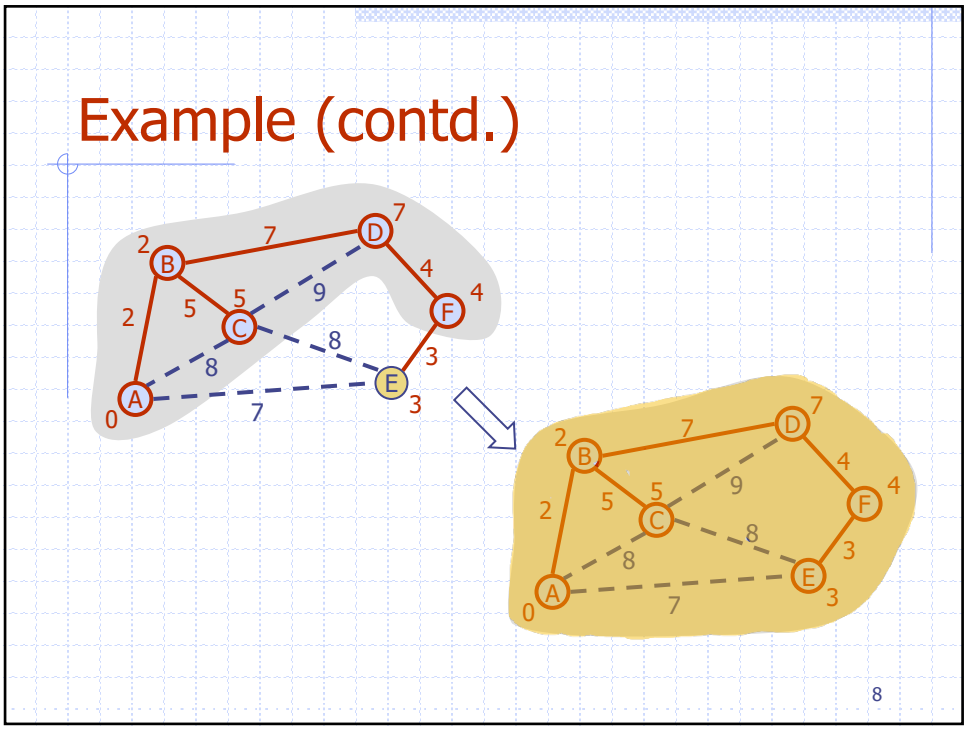
etichetta elemento: vertice-arco che mi fa raggiungere quel vertice

weight → Diffrenza: peso che ho usato finora il nodo rispetto a peso del 1° arco che mi ha permesso di raggiungerlo

Quando finisce coda ho aggiunto tutti nodi e archi connessi

6


Example
BD e AE possono essere usati. Stesso peso, scelgo a piacere.
updated
discarded
Vertice con peso minore
7


Example (contd.)
8

Analysis

- Graph operations
  - We cycle through the incident edges once for each vertex
- Label operations
  - We set/get the distance, parent and locator labels of vertex z $O(\deg(z))$ times
  - Setting/getting a label takes $O(1)$ time
- Priority queue operations
  - Each vertex is inserted once into and removed once from the priority queue, where each insertion or removal takes $O(\log n)$ time
  - The key of a vertex w in the priority queue is modified at most $\deg(w)$ times, where each key change takes $O(\log n)$ time
- Prim-Jarnik's algorithm runs in $O((n+m)\log n)$ time provided the graph is represented by the adjacency list structure
  - Recall that $\sum_v \deg(v) = 2m$
- The running time is $O(m \log n)$ since the graph is connected

Devo fare n set di adiacenti
n set sa anche pr due se fa parte
pr ogni nodo, anche isolati.
Somma dei gradi è somma è 2m
Se grafo è connesso somma è 2m.
Altrimenti è minore.

Proporzionale a n perché
sui nodi la devo fare
almeno una volta. Ma posso
ridurla a 2m, perché
operazione devo compierla al
più grado di z volte per
ogni nodo.

Kruskal's Approach

- Maintain a partition of the vertices into clusters
  - Initially, single-vertex clusters
  - Keep an MST for each cluster
  - Merge "closest" clusters and their MSTs
- A priority queue stores the edges outside clusters
  - Key: weight
  - Element: edge
- At the end of the algorithm
  - One cluster and one MST

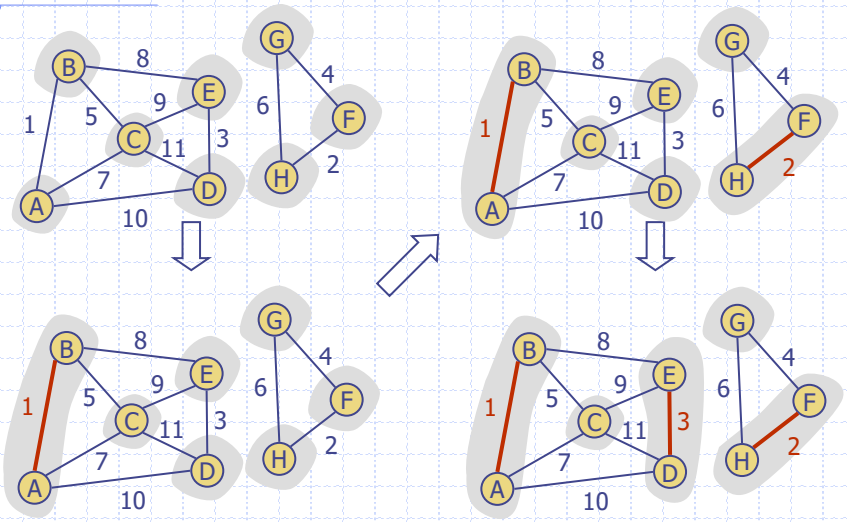
10

Kruskal's Algorithm

Algorithm Kruskal(G):
Input: A simple connected weighted graph G with n vertices and m edges
Output: A minimum spanning tree T for G
for each vertex v in G **do**
Define an elementary cluster $C(v) = \{v\}$.
Initialize a priority queue Q to contain all edges in G, using the weights as keys.
$T = \emptyset$ {T will ultimately contain the edges of the MST}
while T has fewer than $n-1$ edges **do**
(u,v) = value returned by Q.remove_min()
Let $C(u)$ be the cluster containing u, and let $C(v)$ be the cluster containing v.
if $C(u) \neq C(v)$ **then**
Add edge (u,v) to T.
Merge $C(u)$ and $C(v)$ into one cluster.
return tree T

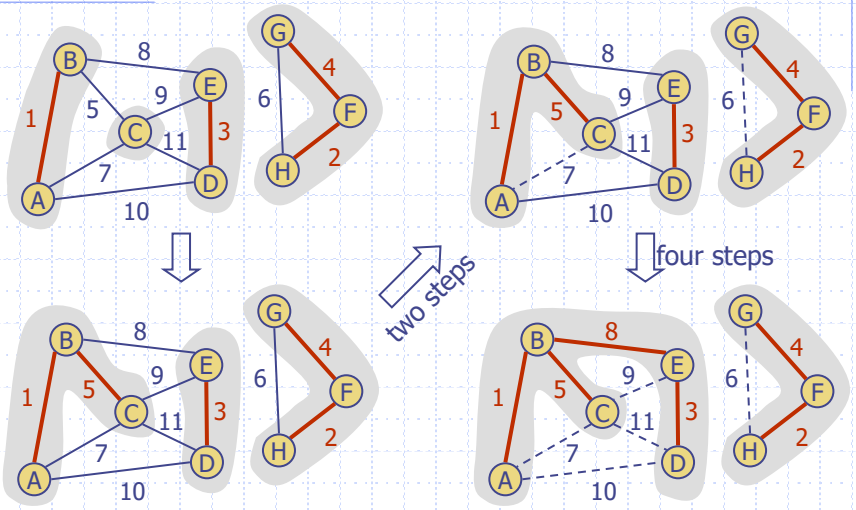
11

Example



12

Example (contd.)



13

Mi fermo quando nella coda a priorità non ci sono più archi.

Problema sono i cluster: non so strutture dati. Potremmo usare Grafo? Non male.

Data Structure for Kruskal's Algorithm

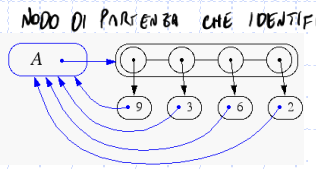
- The algorithm maintains a forest of trees
- A priority queue extracts the edges by increasing weight
- An edge is accepted it if connects distinct trees
- We need a data structure that maintains a partition, i.e., a collection of disjoint sets, with operations:
  - makeSet(u): create a set consisting of u
  - find(u): return the set storing u
  - union(A, B): replace sets A and B with their union

Costruttore Kind of

14

Se ho arco che è già in un cluster, devo trovare cluster che contiene arco

List-based Partition



NODO DI PARTENZA CHE IDENTIFICA

- Each set is stored in a sequence
- Each element has a reference back to the set
  - operation find(u) takes O(1) time, and returns the set of which u is a member. *
  - in operation union(A,B), we move the elements of the smaller set to the sequence of the larger set and update their references ⇒ prendo oggetti più piccolo e aggiorno.
  - the time for operation union(A,B) is min(|A|, |B|)
- Whenever an element is processed, it goes into a set of size at least double, hence each element is processed at most log n times
- Total time needed to do n unions and finds is O(n log n)

15

CLUSTER è NON CONTIENE DATI

Nella partizione non ci sono archi. E e CLUSTER? ENDPOINT SONO NELLA STESSA PARTIZIONE?

Ho connessione 1 vertice e il successivo, ma anche al collegamento al cluster di riferimento. Perché? Se voglio sapere se vertice in lista vicino se ho collegamento a quello specifico cluster. ⇒ O(1)

→ Tutte le volte che unisco raddoppio, unione eseguita in tempo log, ma n volte: aggiungere *

* Quando creo cluster lo salvo in modo (A, vertice)
Vertice inserito nel cluster finisce in cluster che è almeno il doppio del cluster di partenza.

Partition-Based Implementation

- Partition-based version of Kruskal's Algorithm
  - Cluster merges as unions
  - Cluster locations as finds
- Running time $O((n+m)\log n)$ Complessivamente con.
  - Priority Queue operations: $O(m \log n)$
  - Union-Find operations: $O(n \log n)$

Quello calcolato prima sempre se connesso

16

* n-1 elementi e li devo provare: inserire: n volte

Devo fare log n unioni ma complessivamente n volte, per gli n vertici nella struttura.

Java Implementation

```
1  /** Computes a minimum spanning tree of graph g using Kruskal's algorithm. */
2  public static <V> PositionalList<Edge<Integer>> MST(Graph<V,Integer> g) {
3    // tree is where we will store result as it is computed
4    PositionalList<Edge<Integer>> tree = new LinkedPositionalList<>();
5    // pq entries are edges of graph, with weights as keys
6    PriorityQueue<Integer, Edge<Integer>> pq = new HeapPriorityQueue<>();
7    // union-find forest of components of the graph
8    Partition<Vertex<V>> forest = new Partition<>();
9    // map each vertex to the forest position
10   Map<Vertex<V>,Position<Vertex<V>>> positions = new ProbeHashMap<>();
11
12   for (Vertex<V> v : g.vertices())
13     positions.put(v, forest.makeGroup(v));
14
15   for (Edge<Integer> e : g.edges())
16     pq.insert(e.getElement(), e);
17
```

17

Java Implementation, 2

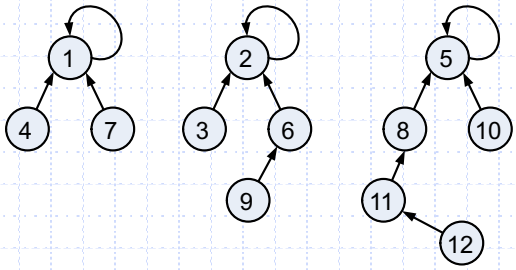
```
18   int size = g.numVertices();
19   // while tree not spanning and unprocessed edges remain...
20   while (tree.size() != size - 1 && !pq.isEmpty()) {
21     Entry<Integer, Edge<Integer>> entry = pq.removeMin();
22     Edge<Integer> edge = entry.getValue();
23     Vertex<V>[] endpoints = g.endVertices(edge);
24     Position<Vertex<V>> a = forest.find(positions.get(endpoints[0]));
25     Position<Vertex<V>> b = forest.find(positions.get(endpoints[1]));
26     if (a != b) {
27       tree.addLast(edge);
28       forest.union(a,b);
29     }
30   }
31
32   return tree;
33 }
```

18

↳ Nodo rappresenta cluster se punta a sé stesso.
Radice = primo nodo del cluster. Nodo che lo identifica.

Tree-based Implementation

- Each element is stored in a node, which contains a pointer to a set name
- A node v whose set pointer points back to v is also a set name
- Each set is a tree, rooted at a node with a self-referencing set pointer
- For example: The sets "1", "2", and "5":



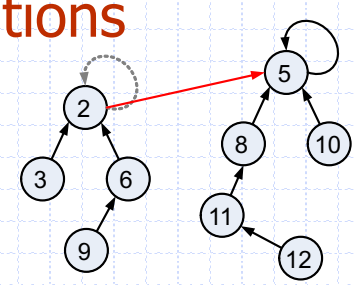
19

Dato un nodo qualsiasi, a quale cluster appartiene, come lo dico per nodo g? Devo muovere il puntatore e arrivare in radice.

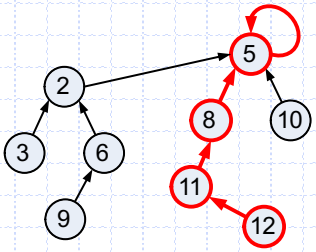
Union-Find Operations

- To do a union, simply make the root of one tree point to the root of the other

Very easy. O(1)



- To do a find, follow set-name pointers from the starting node until reaching a node whose set-name pointer refers back to itself



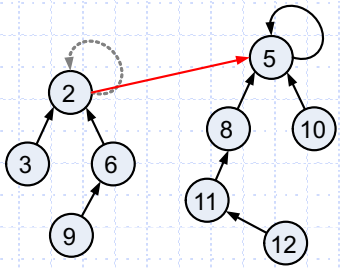
20

Perdo nel find. Per ridurre numero di passaggi, si aggancia albero più piccolo a più grande. 12 avrei dovuto fare 1 passo in più.

m log n si sposta dalla union alla find. (h dell'albero circa log n)

Union-Find Heuristic 1

- Union by size:
  - When performing a union, make the root of smaller tree point to the root of the larger
- Implies O(n log n) time for performing n union-find operations:
  - Each time we follow a pointer, we are going to a subtree of size at least double the size of the previous subtree
  - Thus, we will follow at most O(log n) pointers for any find.



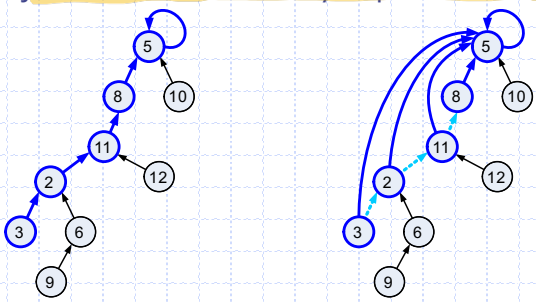
21

Best practice. Mediamente risparmio tempo.

Union-Find Heuristic 2

Riduco complessivamente il tempo.

- Path compression:
  - After performing a find, compress all the pointers on the path just traversed so that they all point to the root



Find della find: percorre tutto albero da inizio alla fine.
Si fa compressione.
Parto da nodo che voglio interrogare e metto in memoria tutti i 22
nodi visitati fino alla radice. Dopo tutti cancello gli collegamenti
fra i nodi e li faccio puntare alla radice.
Ora faccio tutto in tempo costante. Perdendo in generale.
Perché non lo faccio durante la union? La union stessa la fa n-1
volte, la ricerca non più fatta n-1 volte. Se ad un certo punto ho già usato n-1 archi. Posso dover fare n-1 ricerche e
non m.

Durante la union so che non dovrò cercare tutti i nodi.
Se faccio dimensione union, parecchie di quelle ricorsive non le farò.

Java Implementation

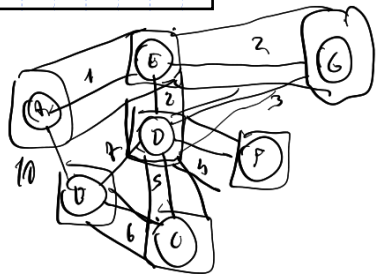
```
1  /** A Union-Find structure for maintaining disjoint sets. */
2  public class Partition<E> {
3    //--------------- nested Locator class -------------
4    private class Locator<E> implements Position<E> {
5      public E element;
6      public int size;
7      public Locator<E> parent;
8      public Locator(E elem) {
9        element = elem;
10       size = 1;
11       parent = this;              // convention for a cluster leader
12     }
13     public E getElement( ) { return element; }
14   } //--------- end of nested Locator class ---------
15   /** Makes a new cluster containing element e and returns its position. */
16   public Position<E> makeCluster(E e) {
17     return new Locator<E>(e);
18   }
```

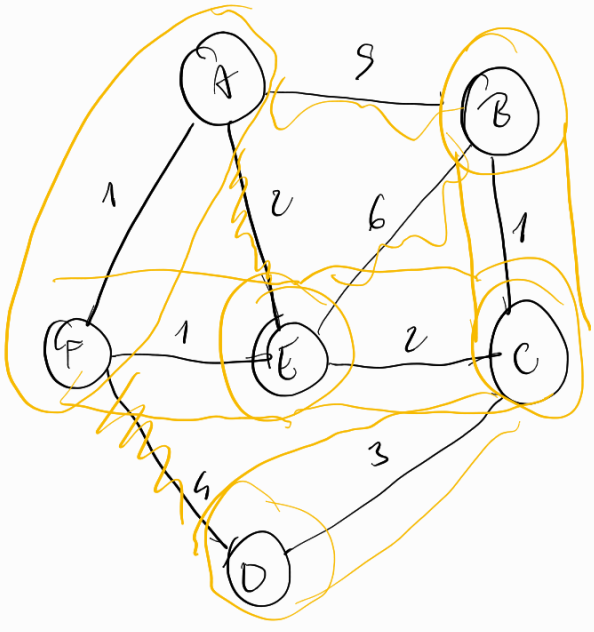
23

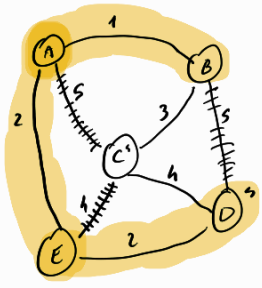
Java Implementation, 2

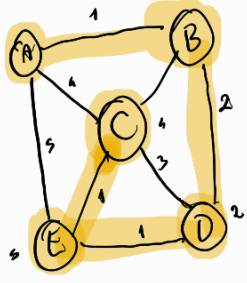
```
19   /**
20    * Finds the cluster containing the element identified by Position p
21    * and returns the Position of the cluster's leader.
22    */
23   public Position<E> find(Position<E> p) {
24     Locator<E> loc = validate(p);
25     if (loc.parent != loc)
26       loc.parent = (Locator<E>) find(loc.parent); // overwrite parent after recursion
27     return loc.parent;
28   }
29   /** Merges the clusters containing elements with positions p and q (if distinct). */
30   public void union(Position<E> p, Position<E> q) {
31     Locator<E> a = (Locator<E>) find(p);
32     Locator<E> b = (Locator<E>) find(q);
33     if (a != b)
34       if (a.size > b.size) {
35         b.parent = a;
36         a.size += b.size;
37       } else {
38         a.parent = b;
39         b.size += a.size;
40       }
41   }
42 }
```

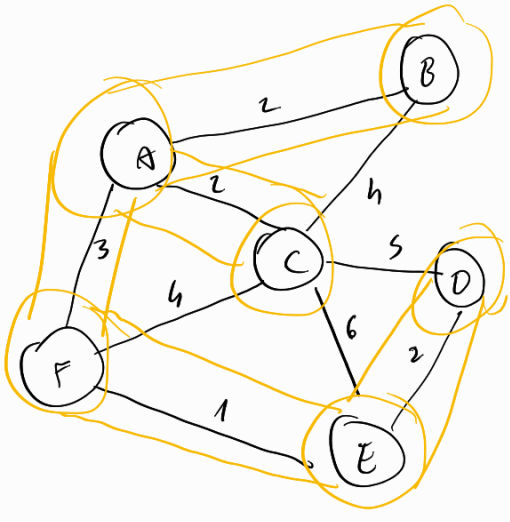
24

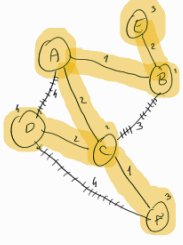


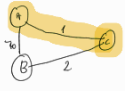

A
B
C
D
E
F
9
1
2
6
1
1
2
3
4

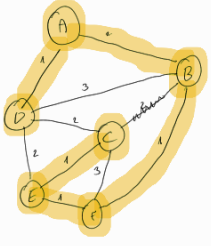

A
B
C
D
E
1
5
3
5
2
4
1
2
5

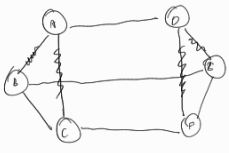

A
B
C
D
E
1
4
2
5
4
3
1
2
1
5


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


A
B
C
D
E
F


A
B
E
2


A
B
C
D
E
F


A
B
C
D
E
F