Università degli Studi dell'Aquila

Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

Università degli Studi dell'Aquila

Corso di Algoritmi e Strutture Dati con Laboratorio

GRAFI – II parte

La classe Network

- Anziché sviluppare otto classi (grafi e alberi, che possono essere rispettivamente orientati o non orientati, pesati o non pesati), svilupperemo soltanto una classe (**directed**) **Network** (Rif. **`Network.java`**)
- Le altre classi possono essere dichiarate per ereditarietà.

La classe Network

Esempi:

- Una rete non orientata è una rete orientata in cui ogni arco è “a due vie”.
- Un digrafo è una rete in cui ogni arco ha lo stesso peso (ad esempio 1.0).

La classe Network

```
public class Network<Vertex>
        implements Iterable<Vertex>
```

- Ricorda: L'interfaccia **`Iterable`** ha un metodo **`iterator()`** che restituisce un riferimento ad un'istanza di una classe che implementa l'interfaccia **`Iterator`**. I metodi **`hasNext()`** e **`next( )`** consentono l'uso del for avanzato.

Campi nella classe Network

- Dato un vertice v, quale informazione su v è rilevante?
1. Tutti i vertici w tali che la coppia <v, w> forma un arco
2. Il peso di ogni arco <v,w>
- Pertanto ad ogni vertice v associamo tutte le coppie <w, weight> tali che <v, w> è un arco di peso weight.
- Come memorizziamo tutte le coppie? Con

```
TreeMap<Vertex, Double>
```

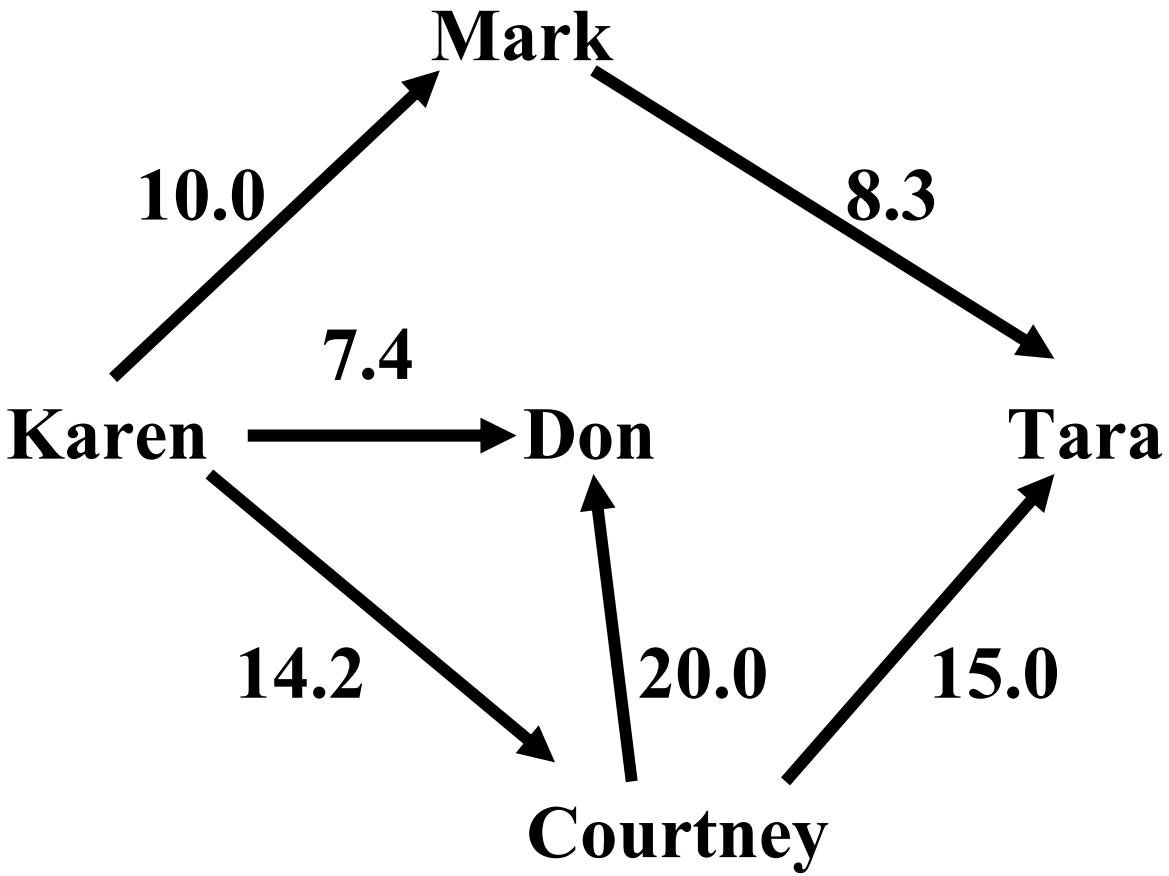
Campi nella classe Network

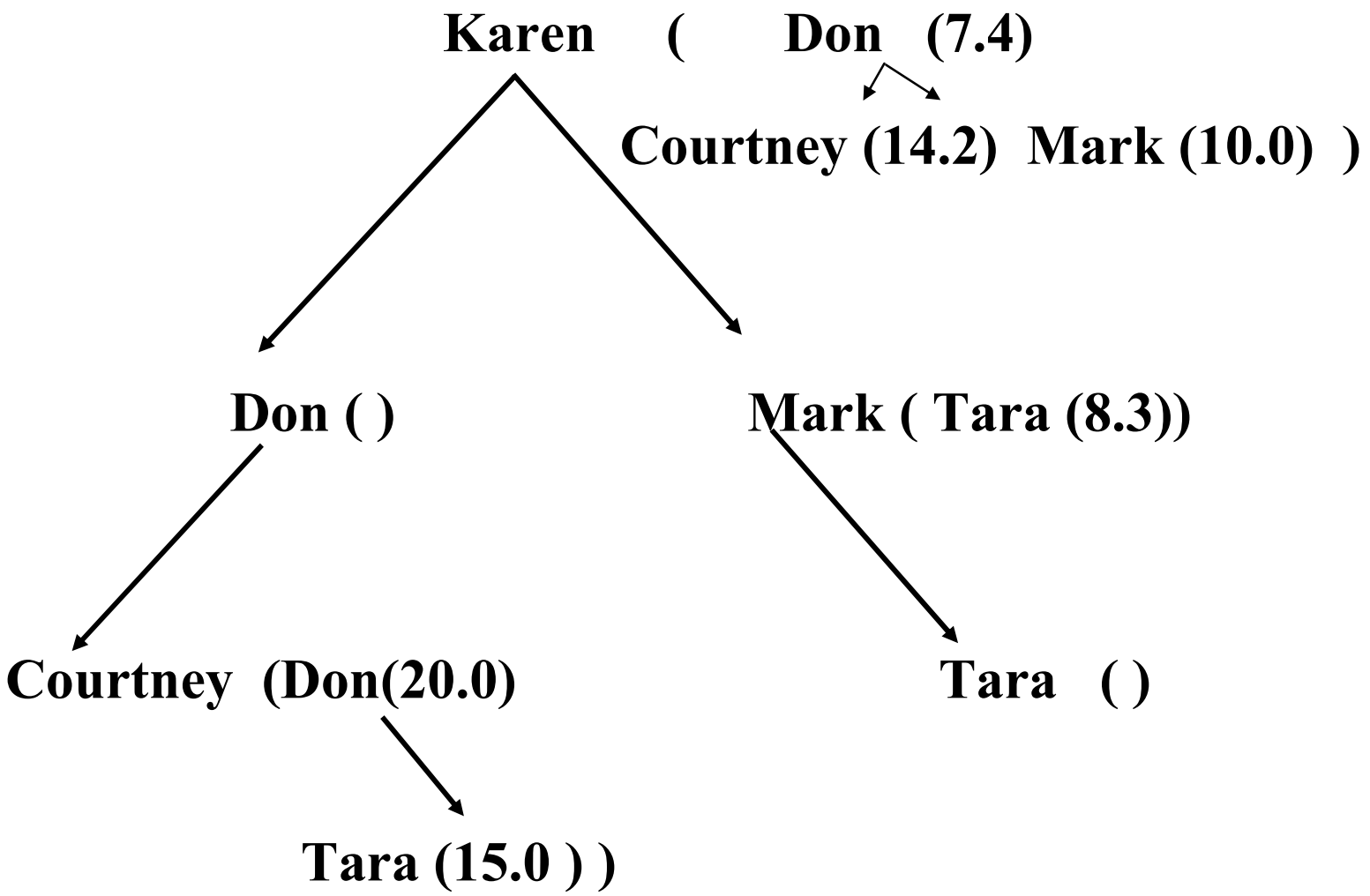
- Quale struttura possiamo usare per "associare" ogni vertice v al suo TreeMap?

  Un altro TreeMap !

- La classe Network ha un solo campo, che mappa ogni vertice v al TreeMap di coppie vertice-peso dei vicini di v:

```
protected TreeMap<Vertex,
   TreeMap<Vertex, Double>> adjacencyMap;
```


Mark
10.0
8.3
7.4
Karen
Don
Tara
14.2
20.0
15.0
Courtney


Karen (Don (7.4)
Courtney (14.2) Mark (10.0))
Don ( )
Mark (Tara (8.3))
Courtney (Don(20.0)
Tara ( )
Tara (15.0))

Connessione (forte)

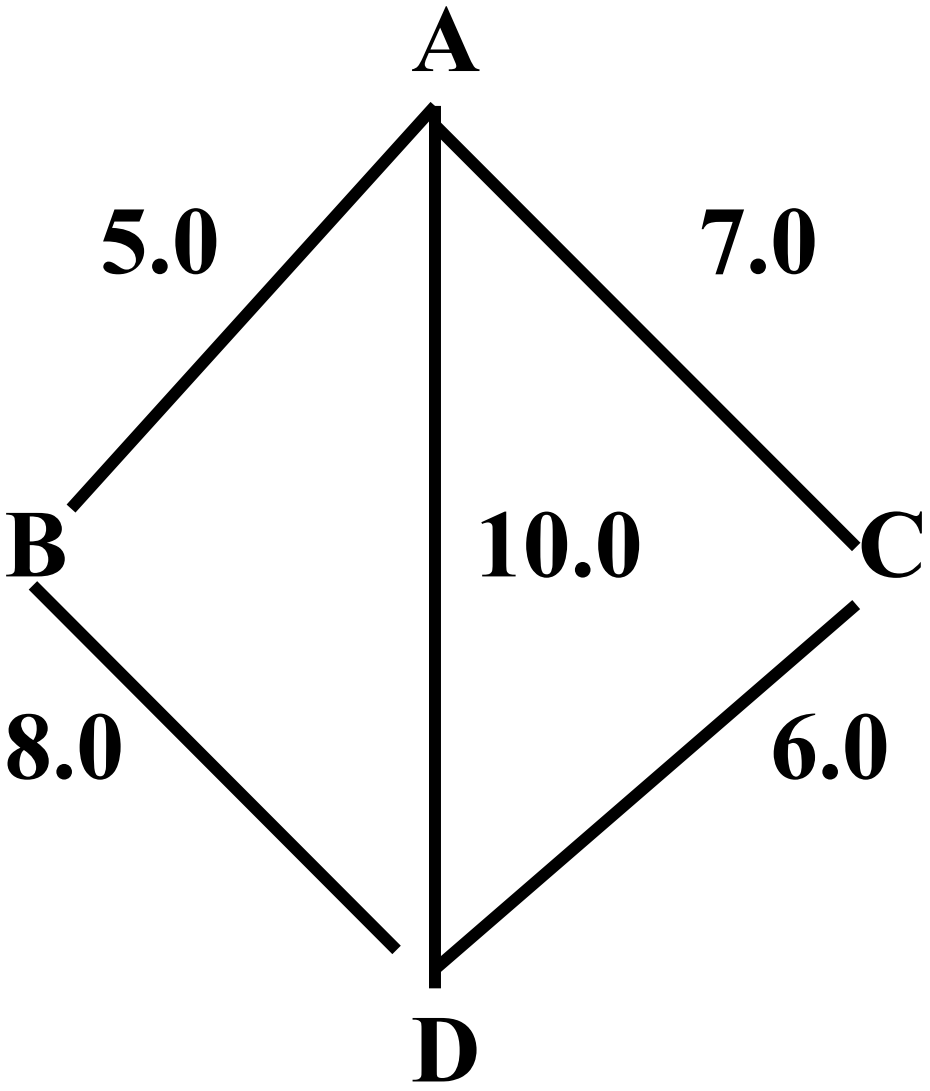
- Una scansione in profondità o in ampiezza può visitare tutti i vertici di grafo soltanto se è connesso
- Sia **`itr`** un iteratore che agisca su un digrafo: per ogni nodo **`v`** restituito da **`itr.next()`**, sia **`bfItr`** un iteratore in ampiezza che abbia **`v`** come vertice di partenza: il numero dei nodi raggiungibili da **`v`** (compreso **`v`**) deve essere uguale al numero di nodi del grafo.
- *Per un grafo non orientato l'algoritmo è più semplice!*

```
public boolean isConnected( )
{
 for ogni vertice v in questo digrafo
 {
  //conta il numero di vertici raggiungibili da v
  //Costruisci un BreadthFirstIterator, bfItr, che
  //parta da v
  int count = 0;
  while (bfItr.hasNext()){bfItr.next(); count++;}
  if (count < numero dei vertici in questo digrafo)
      return false;
  } //for
  return true;
} // algorithm for isConnected
```

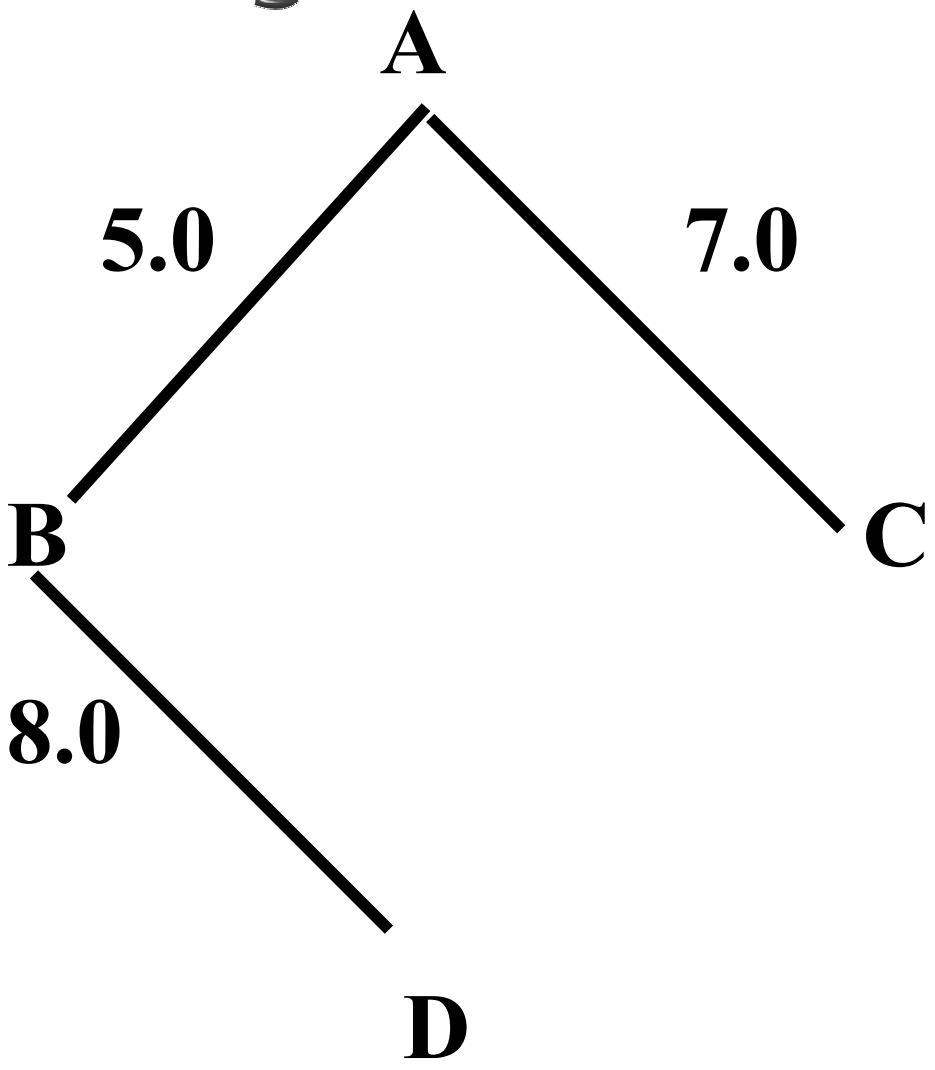
Ricerca del Minimo spanning Tree

- In una rete non orientata e connessa uno spanning tree è un albero pesato che contiene tutti i vertici della rete e alcuni tra i suoi archi (ed i corrispondenti pesi)
- Un **minimo spanning tree** (MST) è uno spanning tree in cui la somma di tutti i pesi non è maggiore della somma di tutti i pesi in qualsiasi altro spanning tree

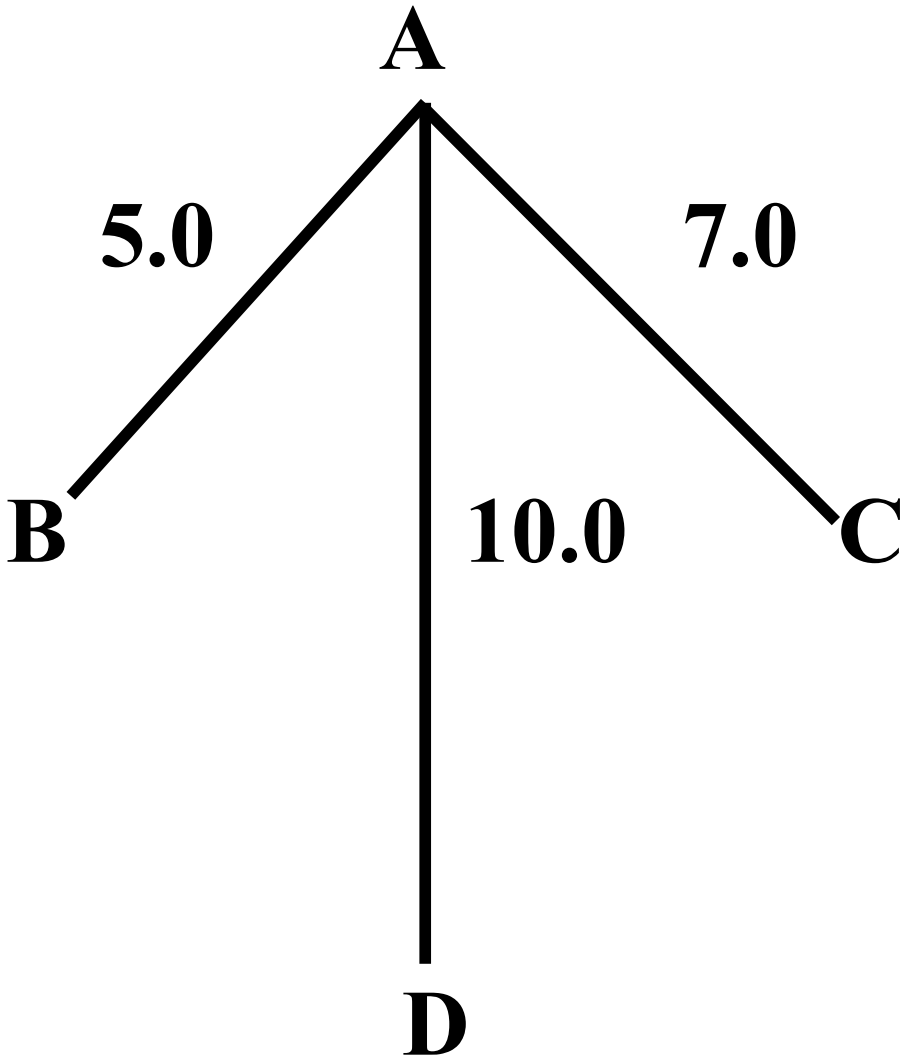
Esempio



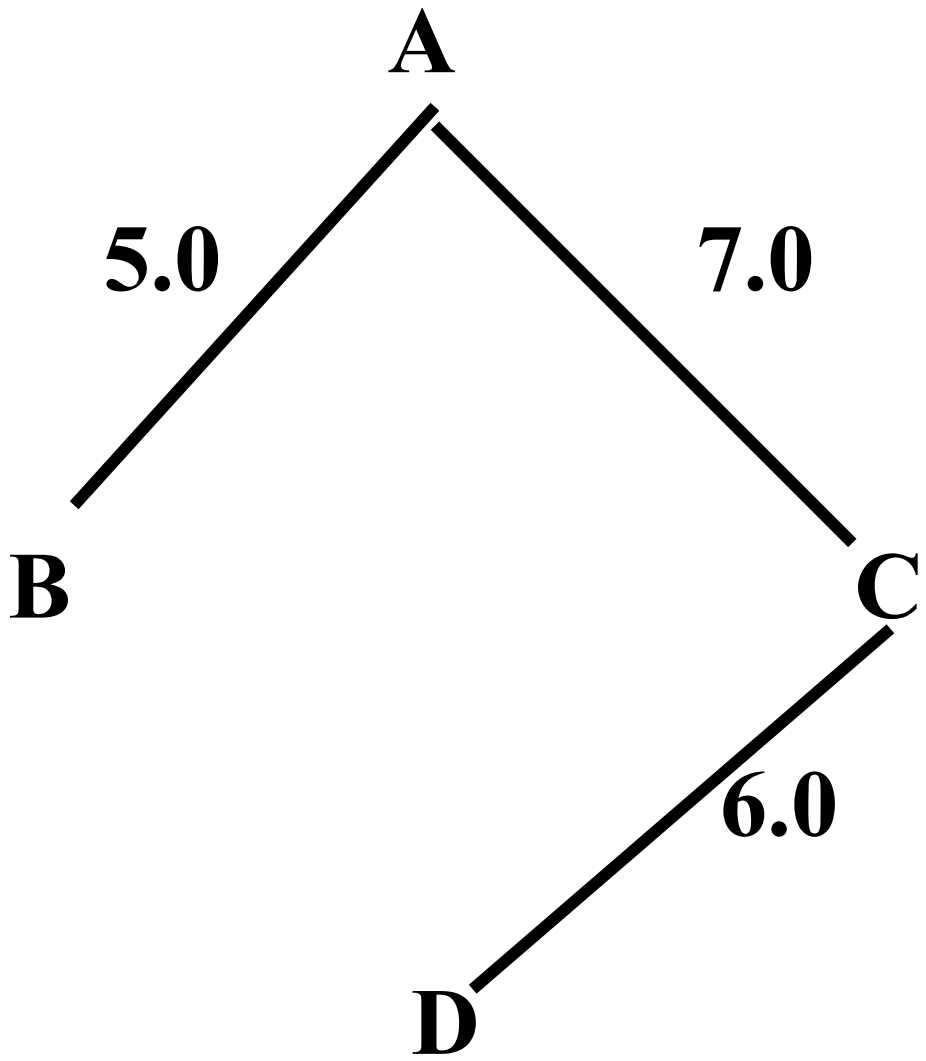
Uno spanning tree



Un altro spanning tree



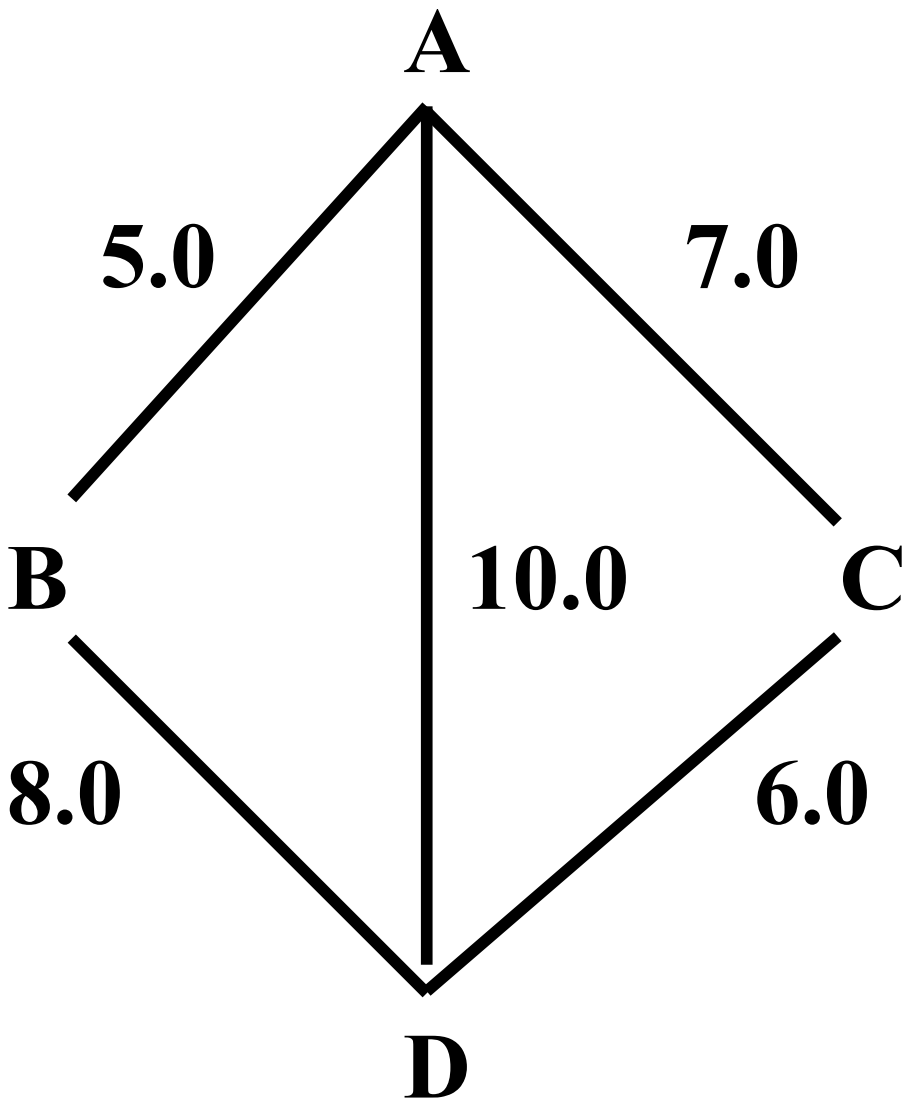
Uno spanning tree minimo



Nota: Un qualunque vertice può essere identificato come radice.

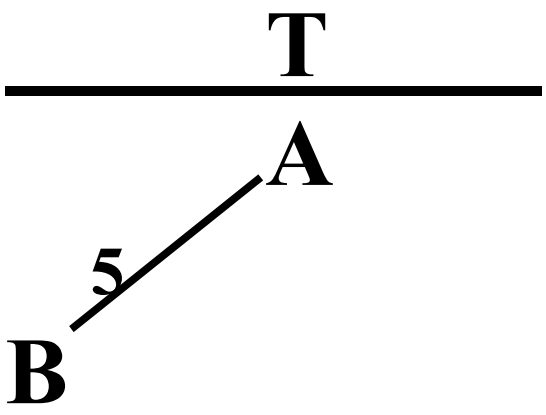
Algoritmo di Prim

- Si inizia con un albero vuoto T e si sceglie un qualunque nodo v della rete. Si aggiunge v a T.
- Per ogni nodo w vicino di v, si memorizza la tripla <v, w, weight(v,w)> in una collezione (*quale?*)
- Fino a quando T non copre tutti i nodi della rete, si estrae dalla raccolta la tripla <x,y,weight(x,y)> che ha il peso weight(x,y) minimo:
  - Se y non è in T si aggiunge y e l'arco (x,y) a T e si aggiungono nella raccolta le triple <y,z,weight(y,z)> tali che z non sia già in T
- Che tipo di raccolta serve? **Una coda con priorità**



Iniziamo con il nodo B.

T	Priority Queue
B	<B, A, 5> <B, D, 8>



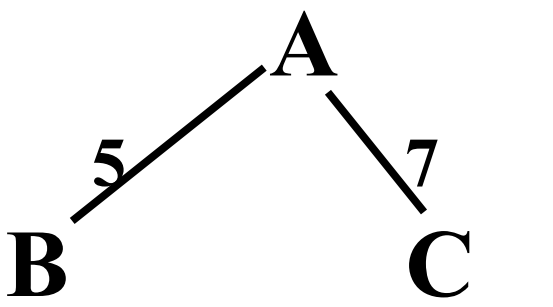
Priority Queue

<A, C, 7>

<B, D, 8>

<A, D, 10>

T



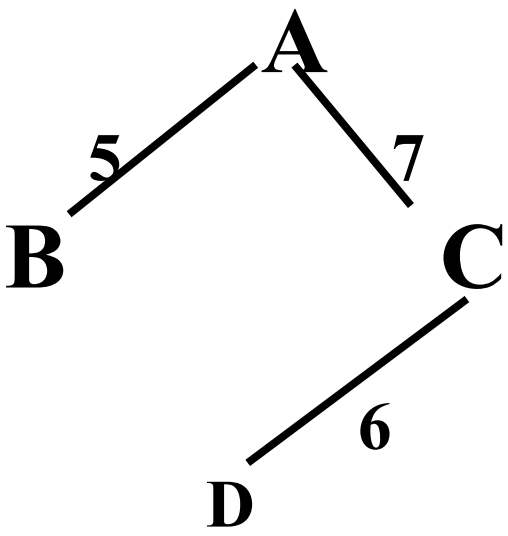
Priority Queue

<C, D, 6>

<B, D, 8>

<A, D, 10>

T



Priority Queue

<B, D, 8>

<A, D, 10>

La procedura termina perché T contiene tutti i vertici della rete.

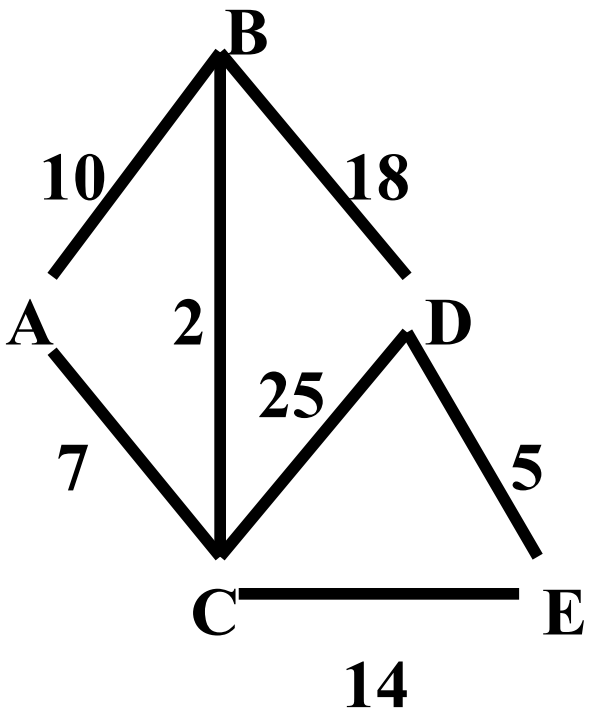
Ricerca del Cammino Minimo

- Data una rete (orientata o non orientata) ed una coppia di vertici distinti v1 e v2, cercare un cammino da v1 a v2 di peso totale minimo.
- Si usa una coda prioritaria.
- **L'algoritmo di Dijkstra** è essenzialmente una visita in ampiezza della rete che parte da v1 e termina quando viene estratta dalla coda prioritaria una coppia contenente v2.
- Un coppia è costituita da un vertice w e dalla somma dei pesi degli archi che compongono un cammino minimo da v1 a w.

- Per tenere traccia dei pesi totali usiamo una mappa **`weightSum`**, nella quale la chiave è il singolo vertice, w, ed il valore che le viene associato è la somma dei pesi degli archi sul cammino minimo da v1 a w
- Per ricostruire il cammino minimo, una volta giunti all'obiettivo si usa un'altra mappa, **`predecessor`**, nella quale a ciascun nodo è associato il nodo che lo precede lungo il cammino minimo scelto da v1 a w.
- Inizialmente la mappa contiene <v1, 0.0>

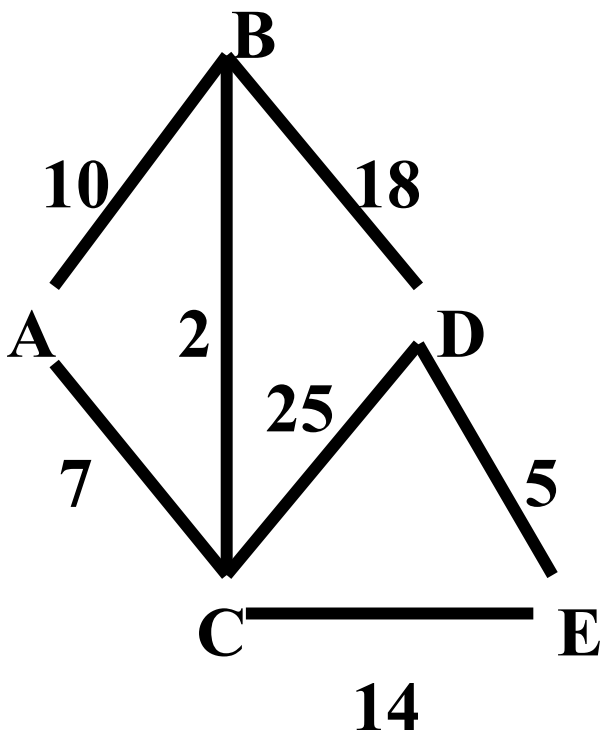
L'algoritmo di Dijkstra

- Sia `pq` una coda prioritaria di coppie <w, wweight>, where wweight è il peso totale di tutti gli archi sul cammino minimo corrente da v1 a w.
- Si inizia da v1 e si continua fino a quando una coppia con v2 non è rimossa da pq



Cerca il cammino minimo da A a D.

`Inizializza weightSum e predecessor,`
`e aggiungi <A, 0> a pq.`

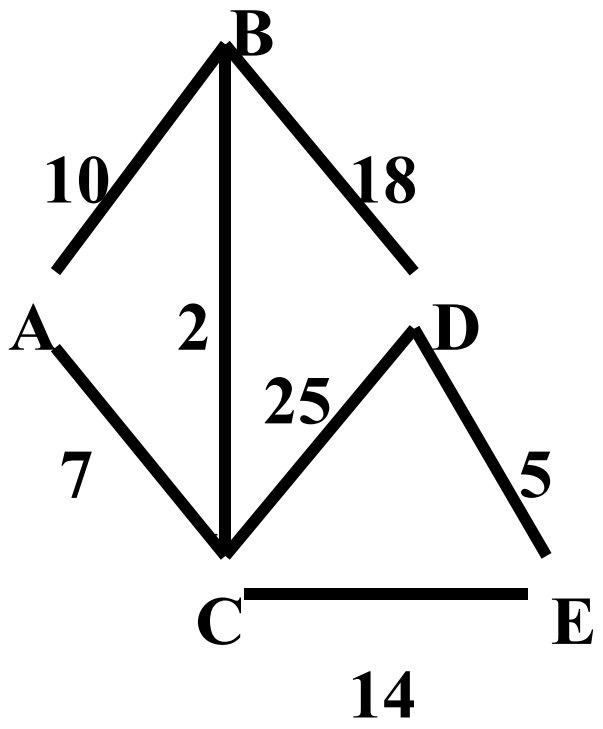


weightSum	predecessor	pq
A, 0	A	<A, 0>
B, 1000	null	
C, 1000	null	
D, 1000	null	
E, 1000	null	

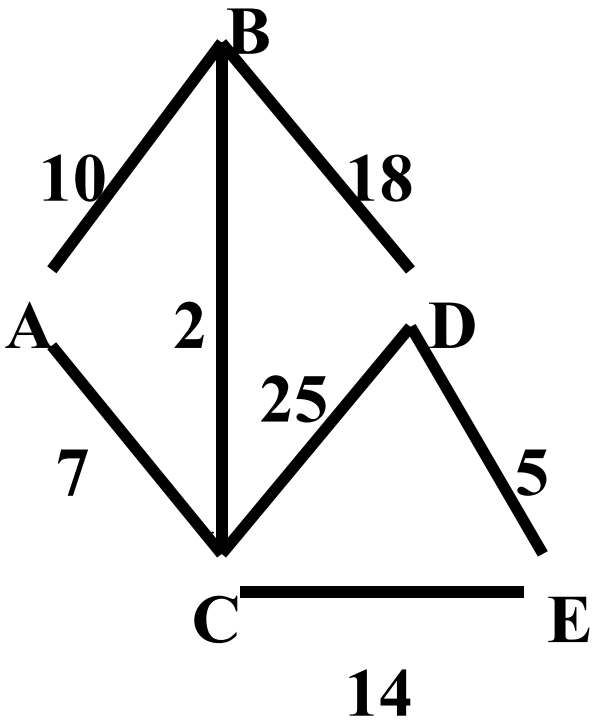
L'algoritmo di Dijkstra

- Durante ogni iterazione si rimuove da pq con un approccio greedy la coppia <w, weightSum> avente peso totale minimo.
- **Passo del rilassamento**: per ogni x adiacente a w, si verifica se il peso totale corrente associato ad x può diminuire usando un cammino che attraversa w:

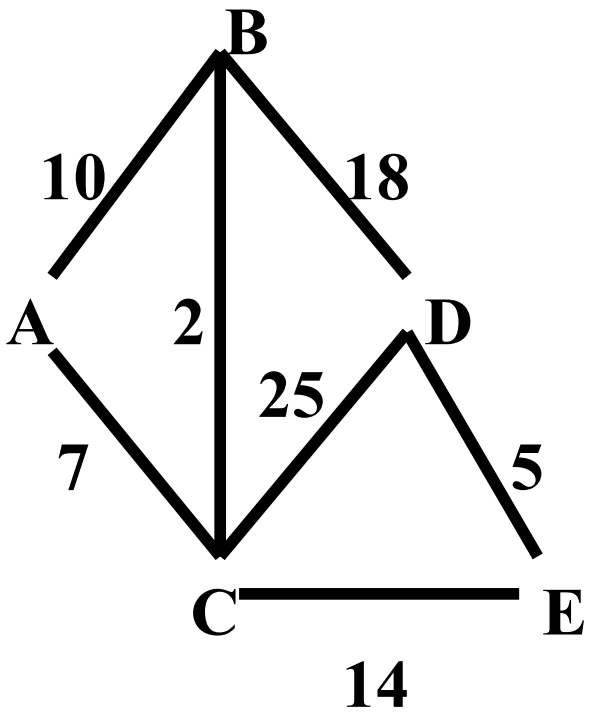
  w's weight sum + weight of (w, x) < x's weight sum ?"
- In questo caso si rimpiazza il corrente peso totale di x con il nuovo peso: *w's weight sum + weight of (w, x)* e si inserisce x ed il suo nuovo peso totale in pq.
- Inoltre, w diventa il predecessore di x.



weightSum	predecessor	pq
A, 0	A	<C, 7>
B, 10	A	<B, 10>
C, 7	A	
D, 1000	null	
E, 1000	null	

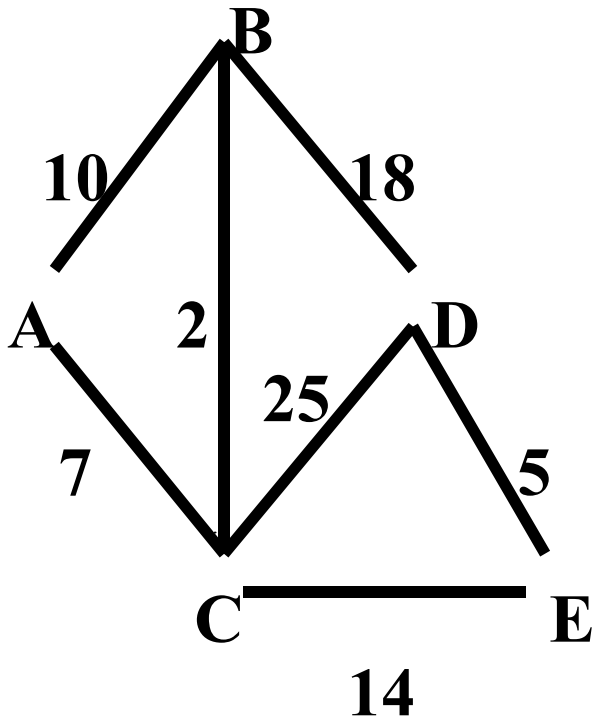


weightSum	predecessor	pq
A, 0	A	<B, 9>
B, 9	C	<B, 10>
C, 7	A	<E, 21>
D, 32	C	<D, 32>
E, 21	C	

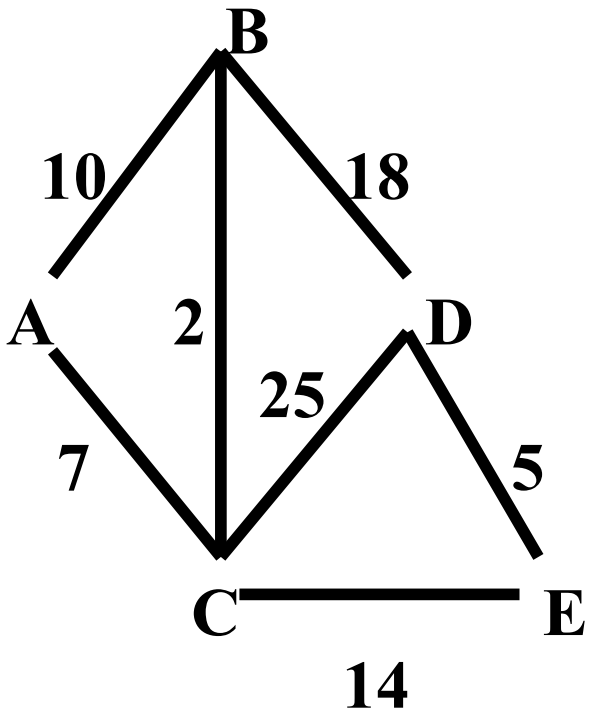


weightSum	predecessor	pq
A, 0	A	<B, 10>
B, 9	C	<E, 21>
C, 7	A	<D, 27>
D, 27	B	<D, 32>
E, 21	C	

Nota: Quando si rimuove <B, 10> da pq, non accade niente in quanto <B, 10> ha peso maggiore di <B, 9>.



weightSum	predecessor	pq
A, 0	A	<E, 21>
B, 9	C	<D, 27>
C, 7	A	<D, 32>
D, 27	B	
E, 21	C	



weightSum	predecessor	pq
A, 0	A	<D, 26>
B, 9	C	<D, 27>
C, 7	A	<D, 32>
D, 26	E	
E, 21	C	

Durante l'iterazione successiva, quando <D, 26> è rimossa da pq ci fermiamo.
Il cammino minimo da A a D, in base a predecessor, è A, C, E, D.