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 (richiami)

La classe Network

Grafo non orientato (indiretto)

- Un **grafo non orientato** è una raccolta di elementi distinti chiamati **vertici** o **nodi** e coppie di vertici distinte e non ordinate, dette **archi.**
- Graficamente rappresentiamo i vertici come punti o cerchi e gli archi con linee che collegano coppie di vertici

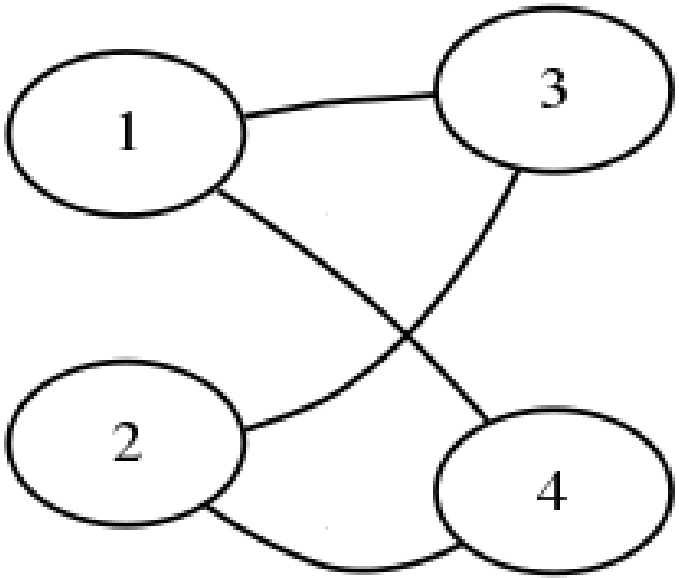
Applicazioni:

- **Facebook**: i vertici sono gli individui e gli archi sono le relazioni di amicizia.
- **Mappa stradale**: i vertici sono le città e gli archi sono le strade (a doppio senso di marcia) che le connettono.

Terminologia

- Due vertici sono **adiacenti** (o vicini) se formano un arco.
- Un **cammino** è una sequenza di nodi in cui ogni coppia di nodi consecutivi è un arco.
- La **lunghezza di un cammino** è uguale al numero di archi che lo compongono: un cammino di k vertici ha lunghezza k-1
- Un **ciclo** è un cammino semplice in cui il primo e l'ultimo nodo coincidono.

Esempio



V = {1, 2, 3, 4}
E = {(1,3), (1,4), (2,3), (2,4)}

- 1 e 3 sono adiacenti (vicini)
- 1 e 2 non sono adiacenti

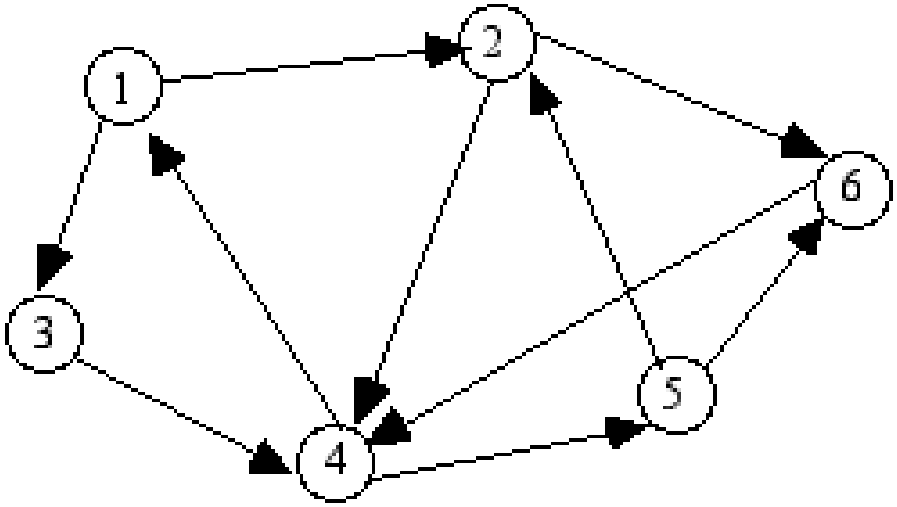
- Cammino: 1,3,2,4 (lungo 3)
- Ciclo: 3,2,4,1,3

Terminologia

- Un grafo non orientato è **completo** se ha il massimo numero di archi possibile: un grafo completo di n nodi ha $n(n-1)/2$ archi
- Un grafo non orientato è **connesso** se per ogni coppia di nodi distinti esiste un cammino che li connette.

Grafo orientato

- Un **grafo orientato** (o diretto, o digrafo) è una raccolta di vertici (o nodi) e archi, in cui gli archi sono coppie ordinate di nodi.
- Rappresentiamo graficamente gli archi con frecce che vanno dal primo al secondo nodo



Terminologia

- In un digrafo un **cammino** segue la direzione delle frecce.
  - Esempio: 1,3,4,5
- Un digrafo è **(fortemente) connesso** se, presi comunque due nodi u e v, esiste almeno un cammino da u a v e viceversa
  - nel caso di grafi non orientati il "viceversa" è scontato…
- Un digrafo è **debolmente connesso** se il grafo non orientato ottenuto ignorando l'orientamento degli archi è connesso.

Alberi

- Un **albero non orientato** è un grafo non orientato, connesso e aciclico in cui un nodo viene designato come radice
- Un **albero** (orientato o generico) è un grafo orientato che è vuoto o ha un nodo radice t.c.
  - Non ci sono archi entranti in radice
  - Ogni nodo non radice ha esattamente un arco entrante
  - Per ogni nodo non radice esiste un cammino che va dalla radice al nodo stesso
- Un albero binario non è semplicemente un albero orientato in cui ogni nodo ha al più due figli ! Perché?

Reti

- Un **grafo etichettato/pesato** sugli archi è un grafo in cui ad ogni arco è associata un'informazione aggiuntiva detta etichetta/peso
- Una **rete (o network)** è un grafo pesato sugli archi con numeri non negativi detti **pesi**
- Dato un cammino in una rete, il **peso totale del cammino** è la somma dei pesi degli archi nel cammino.

Applicazione:

- Mappa con strade (anche a senso unico) etichettate dalle distanze tra le città

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**
- Le altre classi possono essere dichiarate per ereditarietà (homework).

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>
```

*Nota: nell'ultima revisione è stata inserita l'interfaccia

Graph<Vertex, Weight>

e pertanto la def. della classe è cambiata in:

```
public class Network<Vertex extends Comparable<? super
        Vertex>> implements Graph<Vertex, Double>
```

- **Recall**: The **Iterable** interface has an iterator( ) method that returns a reference to an instance of a class that implements the Iterator interface. The hasNext( ) and next( ) methods allow us to use enhanced for statements.

La classe Network: Vertex-related method specifications

```
/**
 *  Determines is this Network object contains vertex.
 *  The worstTime (V, E) is O(log V).
 *  @return true - if this Network object contains vertex;
 *                          otherwise, return false.
 */
public boolean containsVertex (Vertex vertex)

/**
 *  Ensures that this Network object contains a specified
  vertex.
 *  The worstTime(V, E) is O(log V).
 *  @param vertex - the specified vertex.
 *  @return true - if vertex has been inserted in this Network
 *          object as a result of this call; return false if
  vertex
 *          was already in this Network object before this
  call.
 */
public boolean addVertex (Vertex vertex)
```

La classe Network: Vertex-related method specifications

```
/**
 *  Ensures that a specified vertex is not in this Network
  object.
 *  The worstTime(V, E) is O(V log V).
 *
 *  @param vertex - the specified vertex.
 *  @return true - if vertex has been removed from this Network
 *      object as a result of this call; return false if vertex
 *      was not in this Network object when this call was made.
 */
public boolean removeVertex (Vertex vertex)
```

La classe Network: Edge-related method specifications

```
/**
 *  @return the number of edges in this Network object.
 *   The worstTime(V, E) is O(V).
 */
public int edgeSize()

 /**
 *  @return the weight of <v1, v2>, if that is an edge in
 *                this Network object; otherwise, return -1.0.
 *  The worstTime(V, E) is O(log V).
 */
public double getEdgeWeight (Vertex v1, Vertex v2)

/**
* @return true - if this Network object contains the edge
*                 <v1, v2>; otherwise, return false.
* The worstTime(V, E) is O(log V).
*/
public boolean containsEdge (Vertex v1, Vertex v2)
```

La classe Network: Edge-related method specifications

```
/**
 *  Ensures that the specified edge, <v1, v2>, with the
 *  specified weight have been added to this Network object.
 *  The worstTime(V, E) is O(log V).
 *
 *  @return true - if the edge and weight have been added to
 *      this Network object as a result of this call; return
 *      false if this Network object already contained this
 *      edge and weight before this call.
 */
public boolean addEdge (Vertex v1, Vertex v2, double weight)
```

La classe Network: Edge-related method specifications

```
 /**
 *  Ensures that the edge <v1, v2> is not in this Network object.
 *  The worstTime(V, E) is O(log V).
 *
 *  @return true - if the edge <v1, v2> was removed from this
 *          Network object as a result of this call; otherwise,
 *          return false.
 */
public boolean removeEdge (Vertex v1, Vertex v2)
```

La classe Network: Network-as-a-whole method specifications

```
/**
 * Initializes this Network object to be empty, with
 * the vertices ordered naturally (Comparable interface).
 */
public Network()

/**
 * Initializes this Network object to contain a shallow copy of
 * network.
 */
public Network (Network<Vertex> network)
```

La classe Network: Network-as-a-whole method specifications

```
/**
 *  @return true - if this Network object has no vertices;
 *  otherwise, return false.
 */
public boolean isEmpty( )

/**
 *  @return the number of vertices in this Network object.
 */
public int size()
```

La classe Network: Network-as-a-whole method specifications

```
/**
 *  @return a String representation of this Network object.
 *   The worstTime(V, E) is O(V * V).
 */
public String toString()

/**
 *   Returns a LinkedList<Vertex> of the neighbors
 *   of a specified Vertex object.
 *   The worstTime(V, E) is O(V).
 *
 *  @param v - the Vertex object whose neighbors are returned.
 *
 *  @return a LinkedList<Vertex> of the neighbors of v.
 */
public LinkedList<Vertex> neighbors (Vertex v)
```

La classe Network: Network-as-a-whole method specifications

```
/**
 *  @return an Iterator over the vertices in this Network
 *   object.
 */
public Iterator<Vertex> iterator()
```

```
/**
 *  @return a breadth-first iterator over all vertices
 *                reachable from v.
 *   The worstTime(V, E) is O(V log V).
 */
public Iterator<Vertex> breadthFirstIterator (Vertex v)
```

La classe Network: Network-as-a-whole method specifications

```
/**
 *  @return a depth-first iterator over all vertices
 *                reachable from v.
 *   The worstTime(V, E) is O(V log V).
 */
public Iterator<Vertex> depthFirstIterator (Vertex v)


/**
 *  @return true – if this Network object is connected;
 *              otherwise, return false.
 *  The worstTime(V, E) is O(V * V * log V).
 */
public boolean isConnected( )
```

La classe Network: Network-as-a-whole method specifications

```
/**
 * @return a minimum spanning tree for this connected Network
 *         object in which for any vertices u and v, if v is a
 *         neighbor of u then u is a neighbor of v, and the
 *         weights of those two edges are the same.
 *  The worstTime(V, E) is O(E log V).
 */
public UndirectedWeightedTree<Vertex> getMinimumSpanningTree( )

/**
 *  @return a linked list with the shortest path from its
 *             v1 to v2 and total weight.
 *  The worstTime(V, E) is O(E log V).
 */
public LinkedList<Object> getShortestPath (Vertex v1, Vertex v2)
```

Campi nella classe Network

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

HashMap/TreeMap<Vertex, Double>

Campi nella classe Network

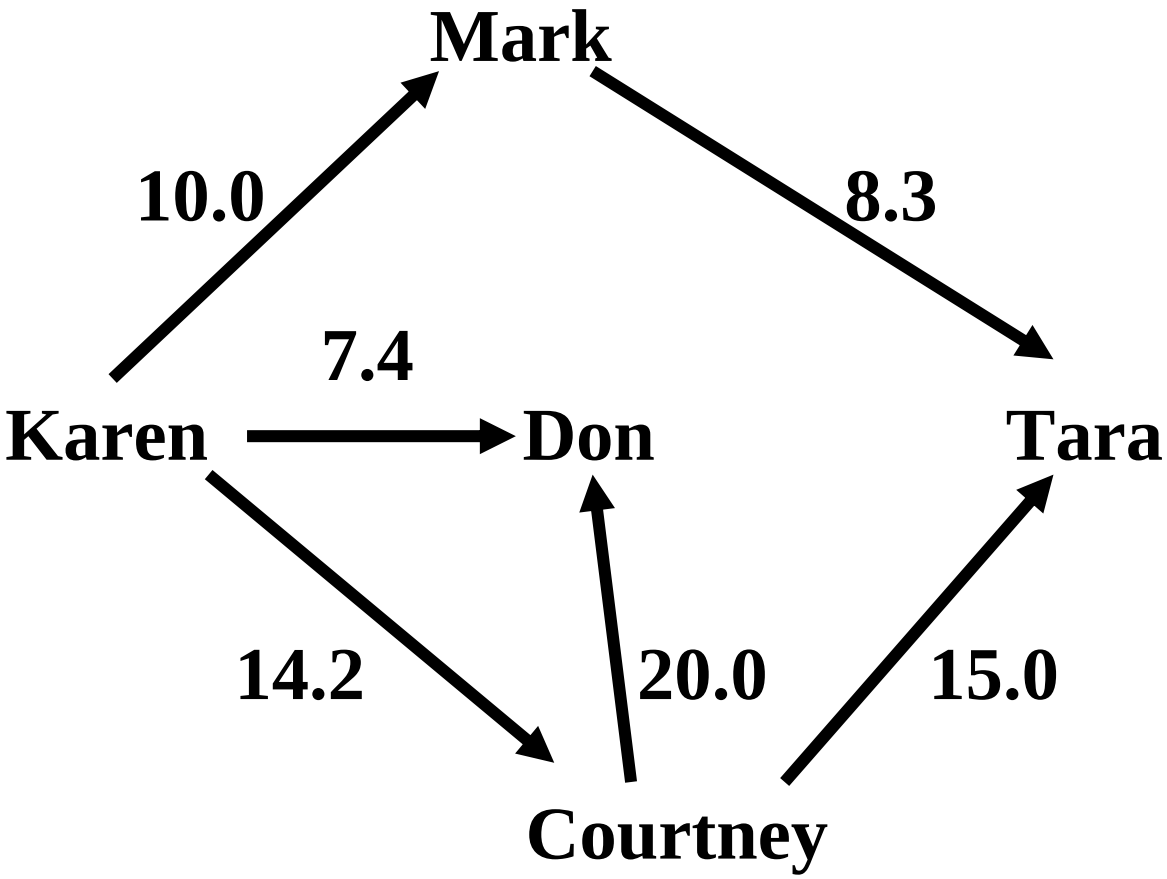
- Quale struttura possiamo usare per "associare" ogni vertice u la sua "neighbors map"?

  Un'altra HashMap/TreeMap !

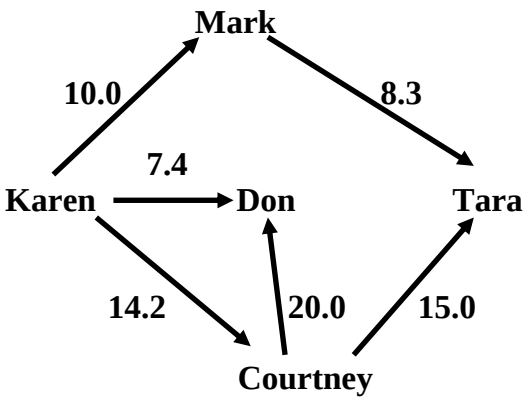
- La classe Network ha un solo campo, che mappa ogni vertice u alla Mappa di coppie vertice-peso dei vicini di u:

```
Map<Vertex, Map<Vertex, Double>> adjacencyMap;
```

«Neighbor Map»


Mark
10.0
8.3
7.4
Karen
Don
Tara
14.2
20.0
15.0
Courtney

TreeMap associata al nodo «Karen»



Karen (Don (7.4) Courtney (14.2) Mark (10.0))

Don ( ) **Mark (Tara (8.3))**

Courtney (Don(20.0) **Tara ( )**

Tara (15.0))

RICHIAMI: Algoritmi per grafi

- Un'operazione che costituisce un prerequisito per altri algoritmi è la possibilità di effettuare la scansione di un grafo:
- Visita "in ampiezza" (breadth-first iterator)
- Visita "in profondità" (depth-first iterator)

Algoritmo di visita generica (richiami)

- La visita parte da un vertice s qualsiasi, il quale viene aggiunto alla frangia F di visita, che contiene tutti i nodi raggiungibili da cui la visita può proseguire, ed esplora seguendo una qualche regola uno degli adiacenti di s
- Al generico passo di visita, viene scelto (secondo una qualche regola) un nodo u in F, e si visita (secondo una qualche regola) un arco (u,v) di G: se il vertice v viene scoperto per la prima volta mediante tale arco, esso viene marcato come visitato, e viene inserito in F; inoltre, il nodo u viene marcato come padre di v, e l'arco (u,v) viene etichettato come arco di visita
- Un vertice rimane in F fintantoché non sono stati scoperti tutti i suoi adiacenti
- Se il grafo è connesso (come abbiamo ipotizzato), la visita genera un albero di visita T del grafo, costituito da tutti i nodi del grafo e dagli archi di visita

Visite particolari (richiami)

- Se la frangia F è implementata come una coda (cioè, quando un vertice si trova in testa alla coda, lo si estrae, vengono visitati tutti i suoi adiacenti e si aggiungono alla coda i neo-visitati) si ha la visita in ampiezza (BFS)
- Se la frangia F è implementata come una pila (cioè, quando un vertice si trova in cima alla pila, viene visitato un vertice adiacente non ancora visitato, il quale viene aggiunto in cima alla pila e subito utilizzato per proseguire nella visita) si ha la visita in profondità (DFS)

Breadth-First Iteration
(also known as Breadth-First Search)

```
algoritmo visitaBFS(vertice s) → albero
1.    rendi tutti i vertici non marcati
2.    T ← albero formato da un solo nodo s
3.    Coda F
4.    marca il vertice s
5.    F.enqueue(s)
6.    while ( not F.isempty() ) do
7.        u ← F.dequeue()
8.        for each ( arco (u, v) in G ) do
9.            if ( v non è ancora marcato ) then
10.               F.enqueue(v)
11.               marca il vertice v
12.               rendi u padre di v in T
13.   return T
```

BreadthFirstIterator()

```
public BreadthFirstIterator (Vertex start)
{
  for every vertex in the network:
      mark that vertex as not reachable.
  mark start as reachable.
  queue.add (start);
} // algorithm for constructor
```

BreadthFirstIterator()

```
public boolean hasNext( )
{
  return !queue.isEmpty( );
} // algorithm for hasNext
```

BreadthFirstIterator()

```
public Vertex next( ) {
 current = queue.remove();
 for each vertex that is a neighbor of current:
     if that vertex has not yet been marked as
          reachable
     {
          mark that vertex as reachable;
          add that vertex to queue;
     } // if
  return current;
} // algorithm for method next
```

Depth-First Iteration

procedura `visitaDFSRicorsiva`$(vertice\ v, albero\ T)$

1. *marca e visita il vertice* v
2. **for each** (arco (v, w)) **do**
3. **if** (w non è marcato) **then**
4. aggiungi l'arco (v, w) all'albero T
5. `visitaDFSRicorsiva`(w, T)

algoritmo `visitaDFS`$(vertice\ s) \rightarrow albero$

6. $T \leftarrow$ albero vuoto
7. `visitaDFSRicorsiva`(s, T)
8. **return** T

DepthFirstIterator()
versione iterativa

```
public DepthFirstIterator (Vertex start)
{
  for every vertex in the network:
      mark that vertex as not reachable.
  mark start as reachable.
  stack.push (start);
} // algorithm for constructor
```

```
public Vertex next( )
{
  current = stack.pop();
  for each vertex that is a neighbor of
  current:
     if that vertex has not yet been marked
     as reachable
     {
          mark that vertex as reachable;
          stack.push (vertex);
      } // if
  return current;
} // algorithm for method next
```

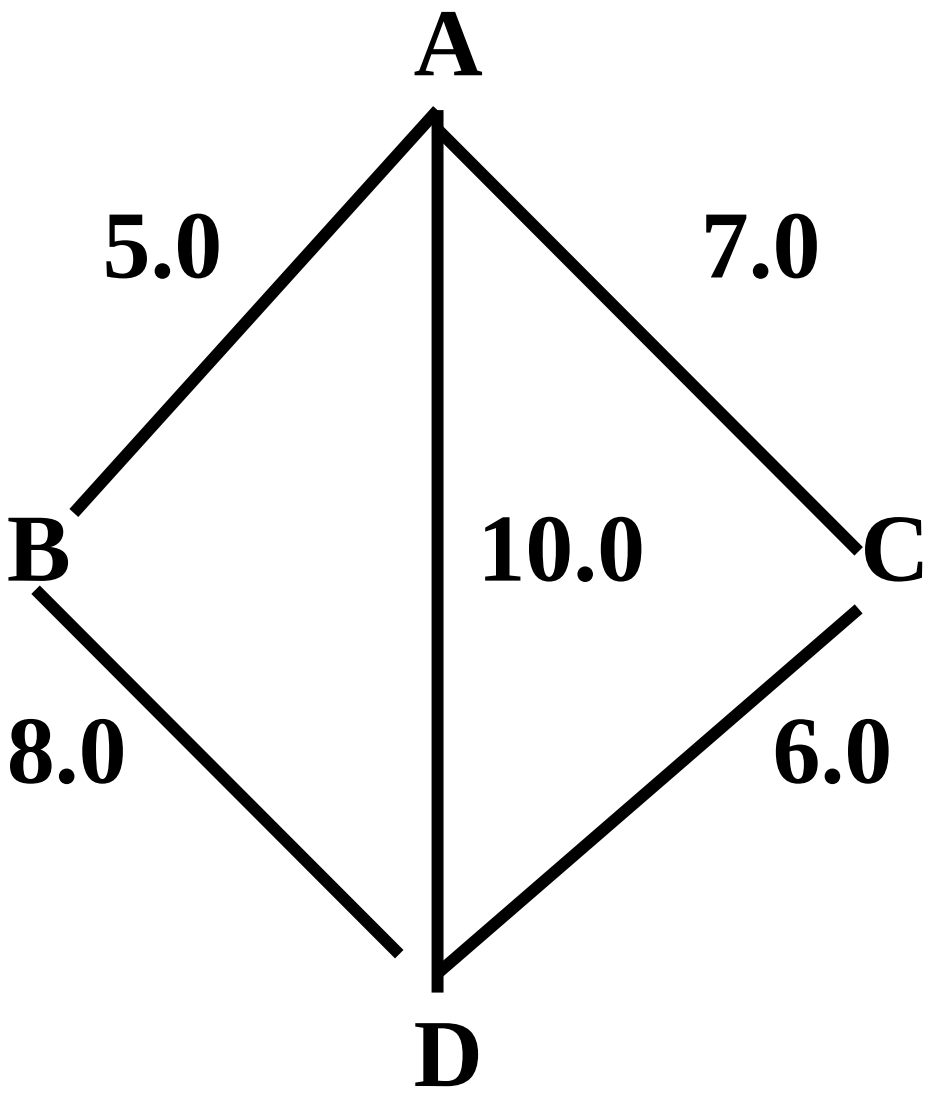
Connessione (forte)

- Una scansione in profondità o in ampiezza può visitare tutti i vertici del 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!*

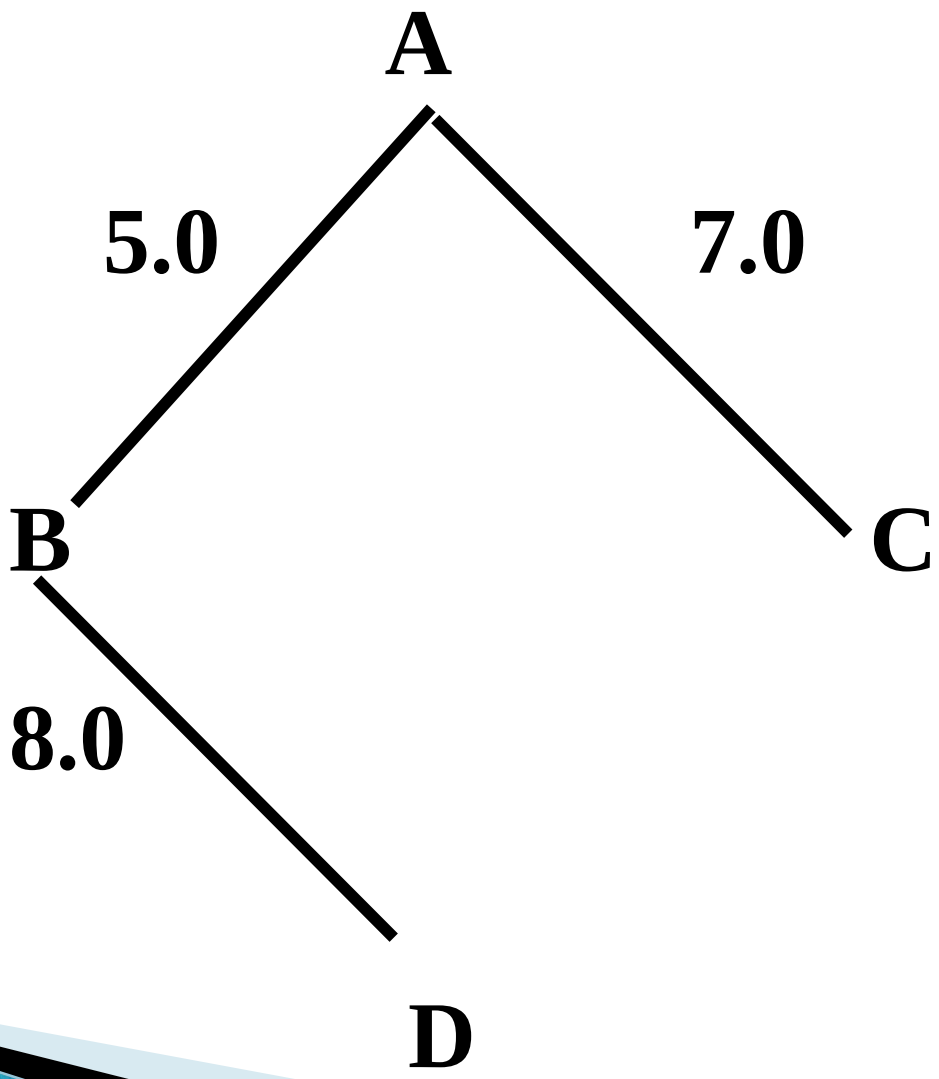
```
public boolean isConnected( )
{
 for ogni vertice v in questo digrafo
 {
  //conta il numero di vertici raggiungibili da v
  //Costruisci un BreadthFirstIterator, bfItr, che
  //parta da v
  int count = 0;
  while (bfItr.hasNext()){bfItr.next(); count++;}
  if (count < numero dei vertici in questo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

Ricerca del Minimo spanning Tree

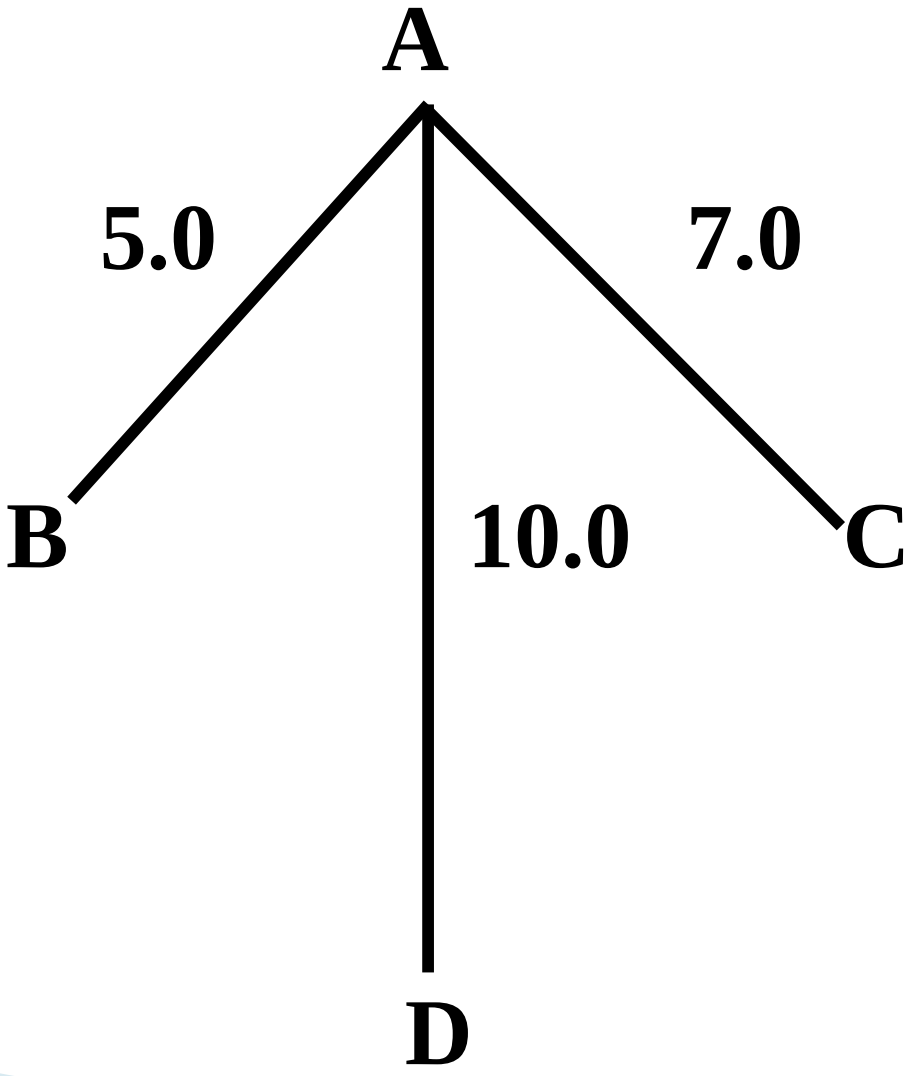
- In una rete non orientata e connessa uno spanning tree è un albero pesato che contiene tutti i vertici della rete e alcuni tra i suoi archi (ed i corrispondenti pesi)
- Un **minimo spanning tree** (MST) è uno spanning tree in cui la somma di tutti i pesi non è maggiore della somma di tutti i pesi in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


A
5.0
7.0
B
10.0
C
8.0
6.0
D

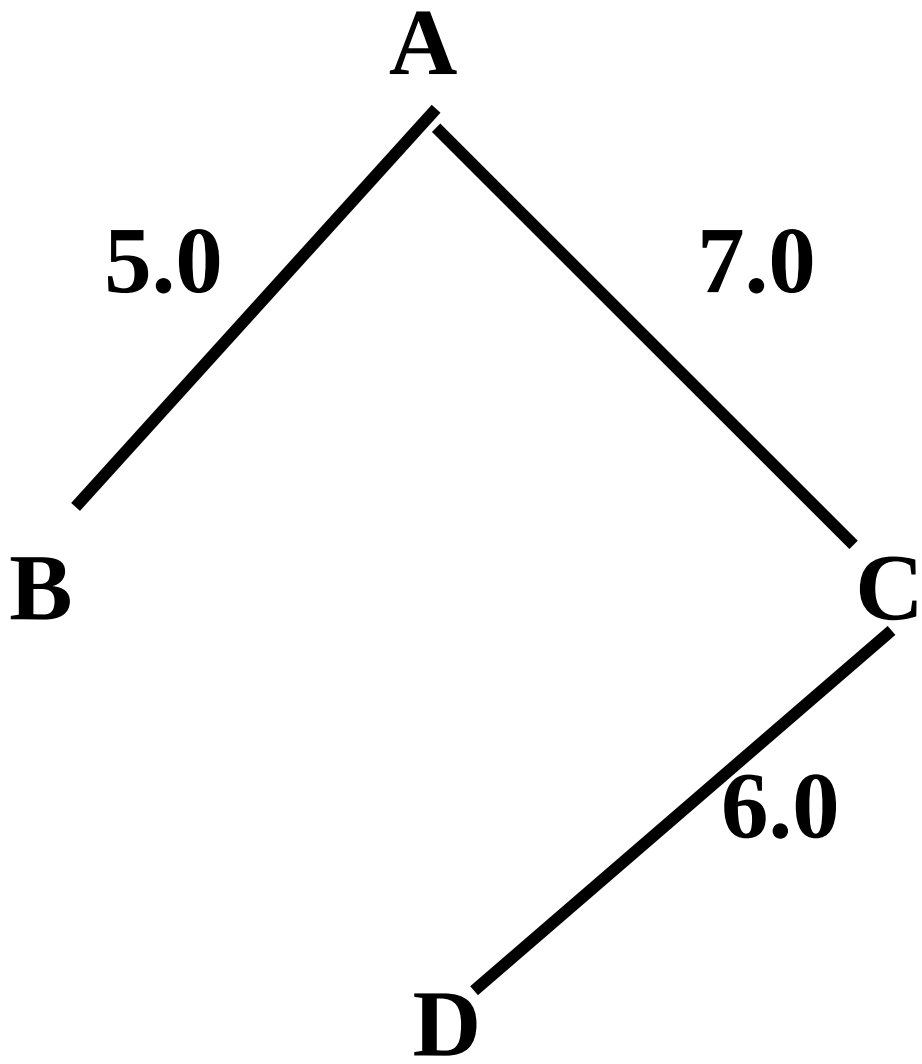
Uno spanning tree:



Un altro spanning tree:



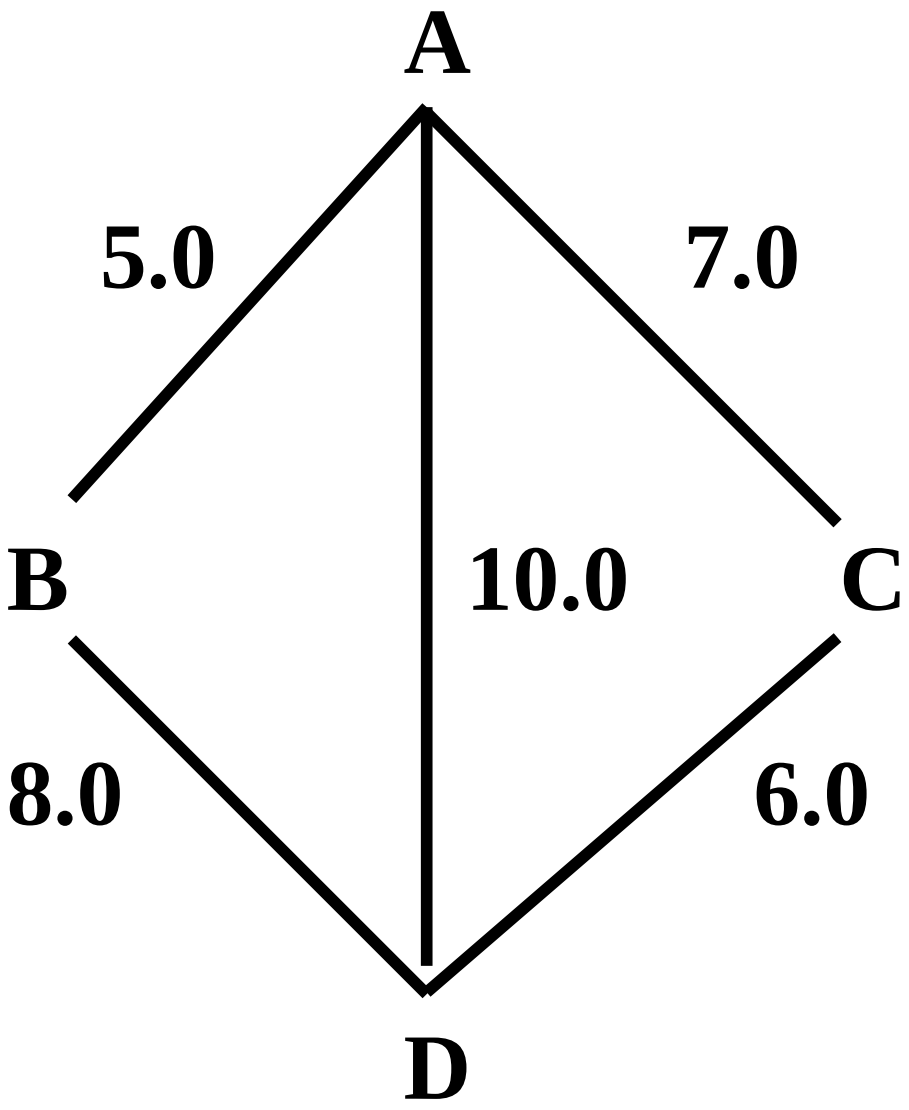
Uno spanning tree minimo:



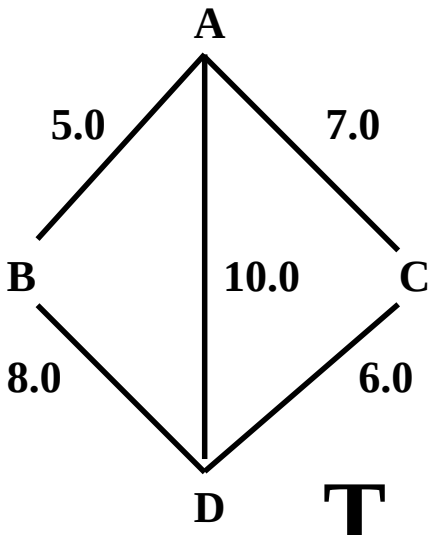
Nota: Un qualunque vertice può essere identificato come radice.

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 di priorità**



Iniziamo con il nodo B.

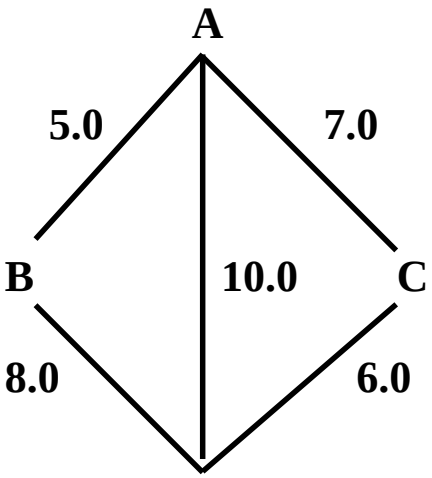


T
B

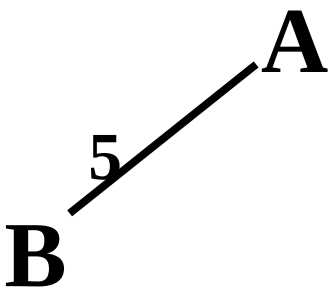
Priority Queue

<B, A, 5>

<B, D, 8>



T

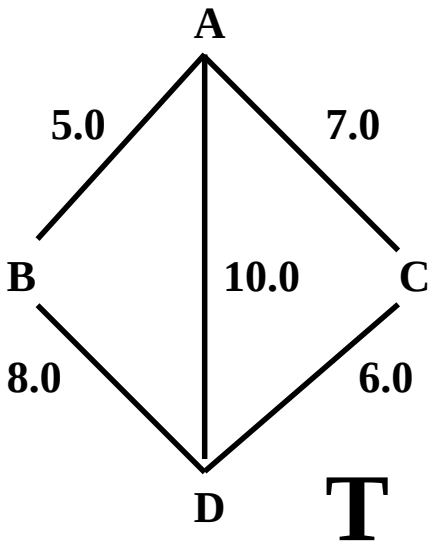


Priority Queue

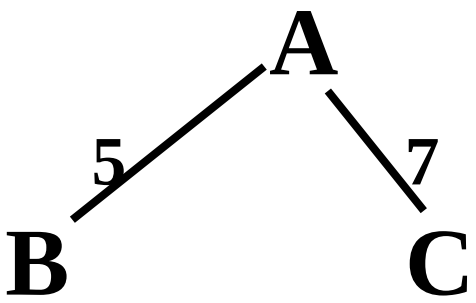
<A, C, 7>

<B, D, 8>

<A, D, 10>



T



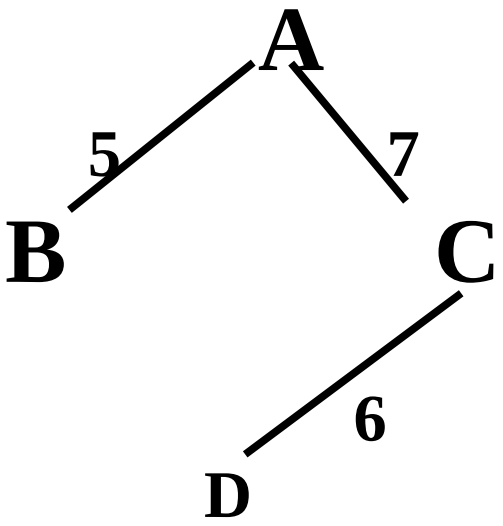
Priority Queue

<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.

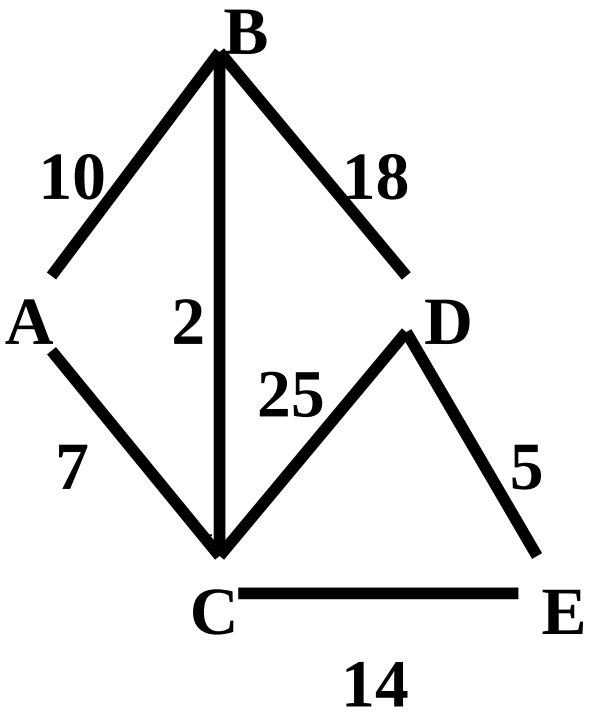
Ricerca del Cammino Minimo

- 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 di priorità.
- **L'algoritmo di Dijkstra** è essenzialmente una visita in ampiezza della rete che parte da v1 e termina quando viene estratta dalla coda di priorità 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>

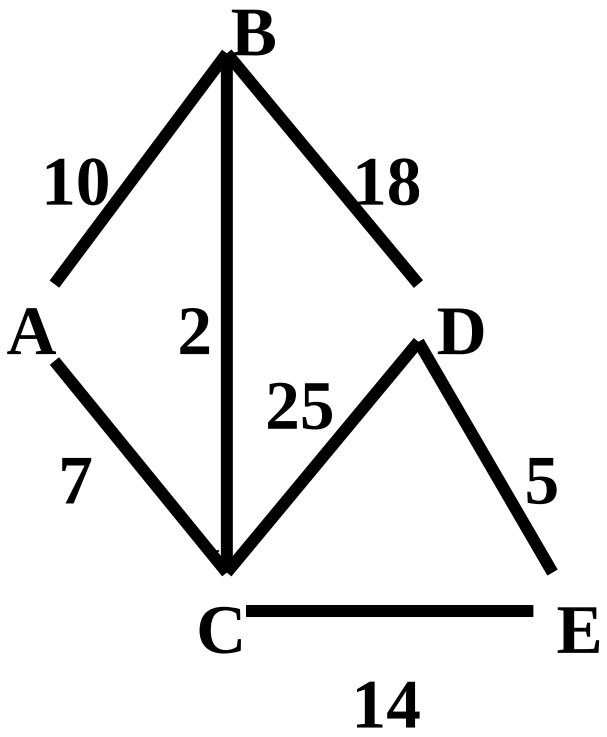
L'algoritmo di Dijkstra

- 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.

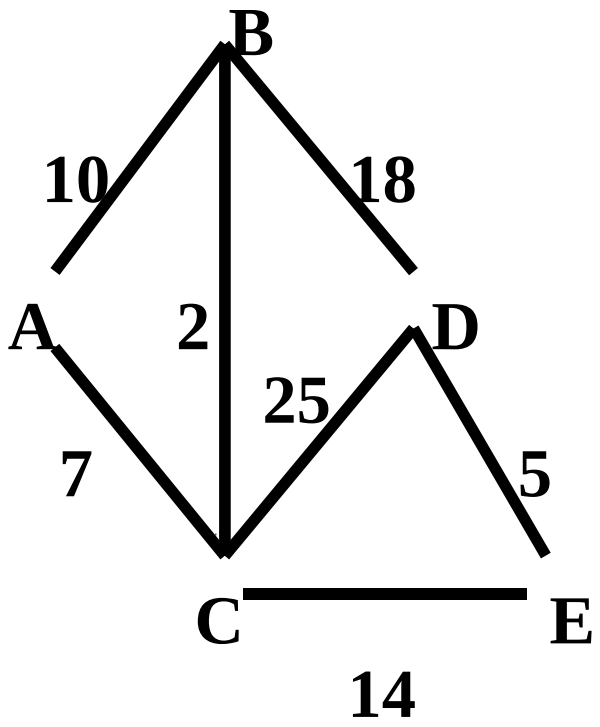


weightSum	predecessor	pq
A, 0	A	<A, 0>
B, 1000	null	
C, 1000	null	
D, 1000	null	
E, 1000	null	

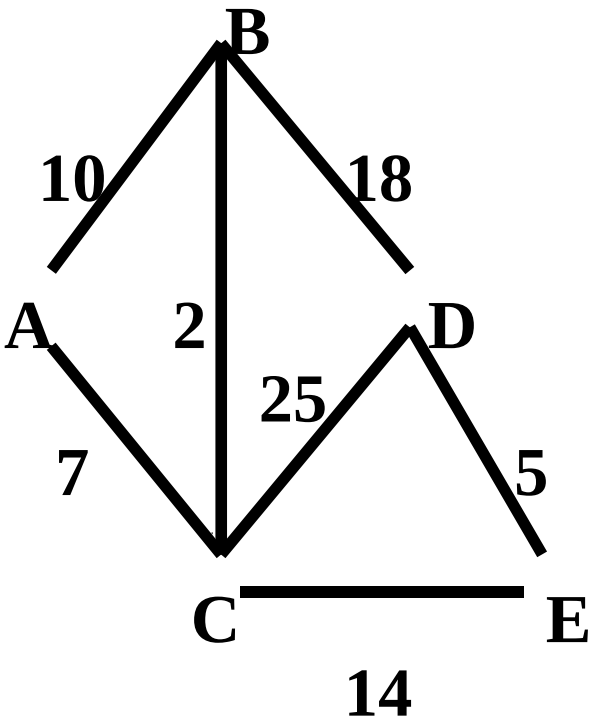
L'algoritmo di Dijkstra

- 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:

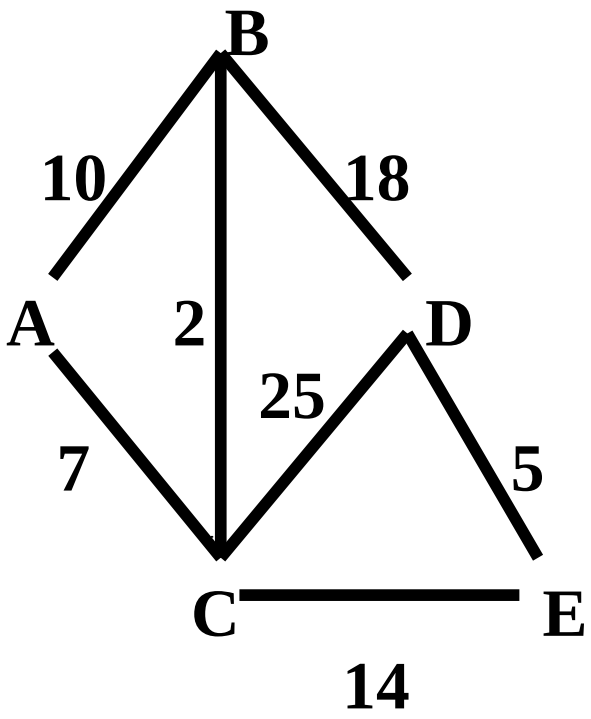
  w's weight sum + weight of (w, x) < x's weight sum ?"
- In questo caso si rimpiazza il corrente peso totale di x con il nuovo peso: *w's weight sum + weight of (w, x)* e si inserisce x ed il suo nuovo peso totale in pq.
- Inoltre, w diventa il predecessore di x.



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

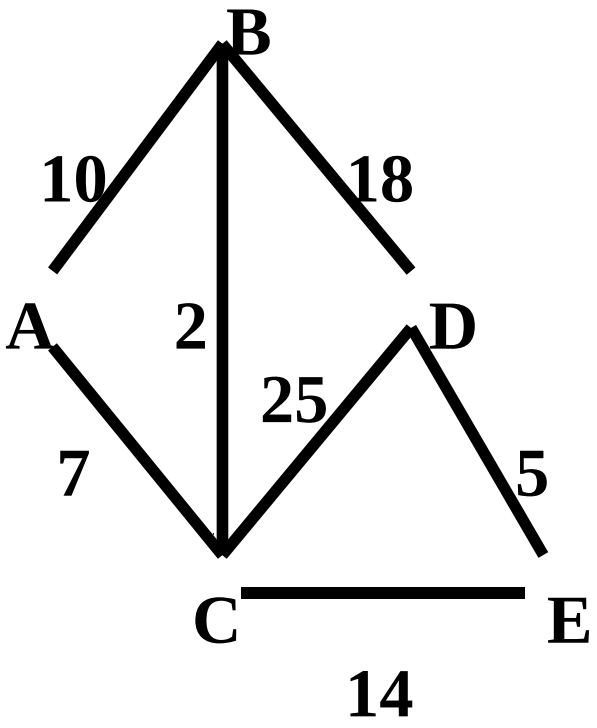


weightSum	predecessor	pq
A, 0	A	<B, 9>
B, 9	C	<B, 10>
C, 7	A	<E, 21>
D, 32	C	<D, 32>
E, 21	C	

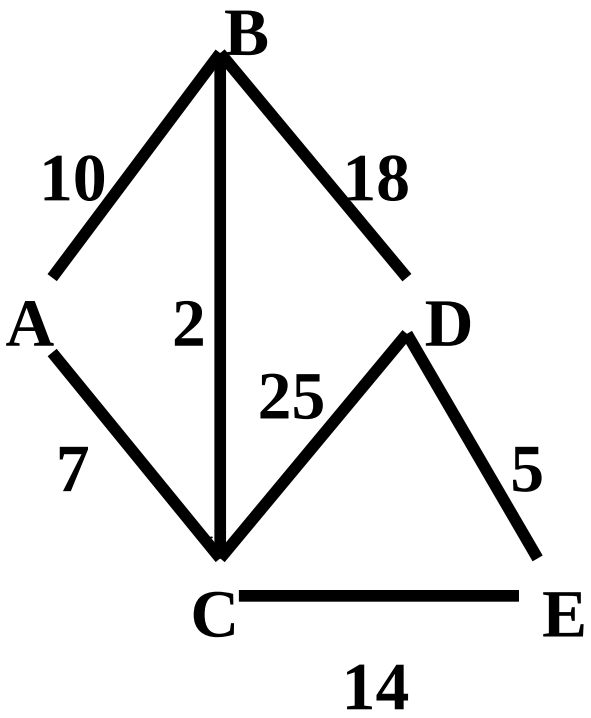


weightSum	predecessor	pq
A, 0	A	<B, 10>
B, 9	C	<E, 21>
C, 7	A	<D, 27>
D, 27	B	<D, 32>
E, 21	C	

Nota: Quando si rimuove <B, 10> dap q, 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.