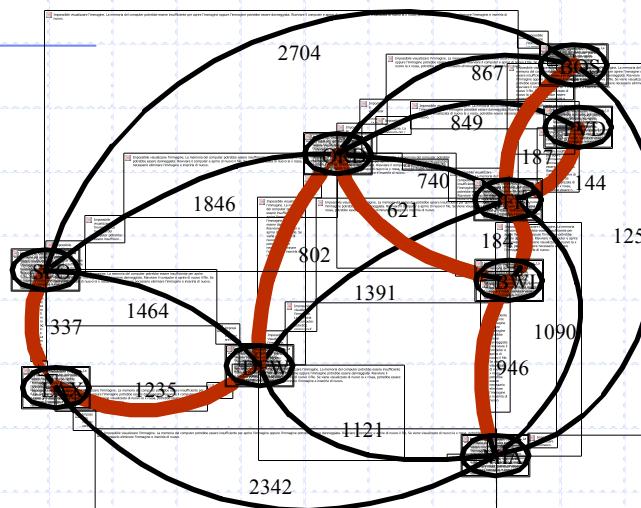


# Minimum Spanning Tree



# Minimum Spanning Tree

## Sottografo ricoprente

- Sottografo di un grafo  $G$  contenente tutti i vertici di  $G$

## Spanning tree

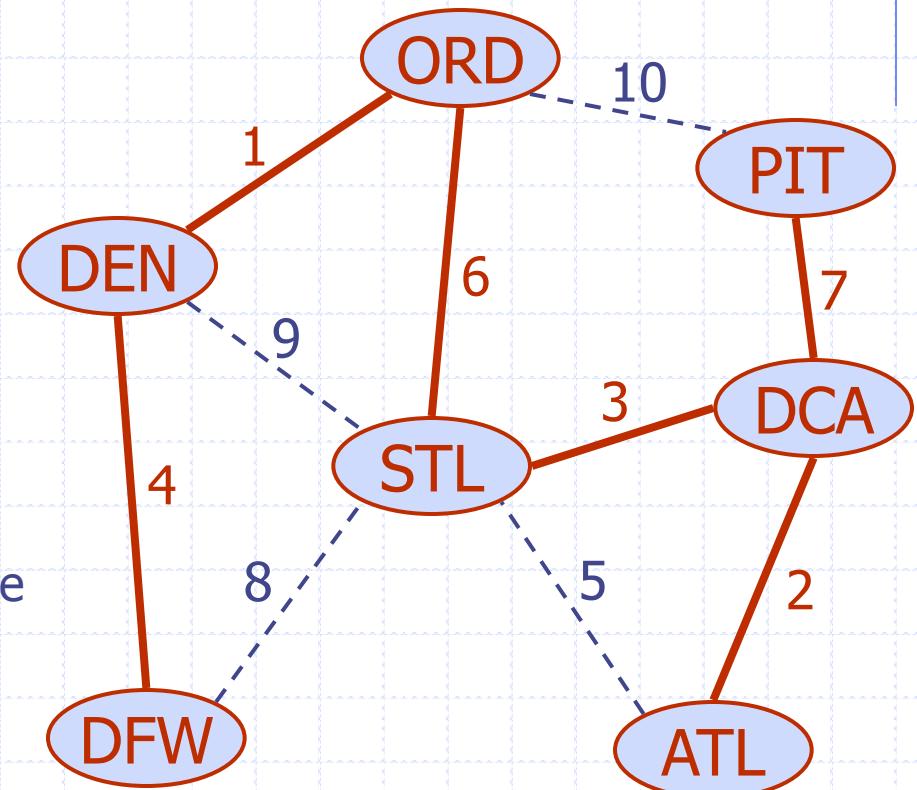
- Sottografo ricoprente che e' esso stesso un albero non radicato

## Minimum spanning tree (MST)

- Albero ricoprente di un grafo pesato con minimo costo totale degli archi

## ◆ Applicazioni

- Reti di comunicazione
- Reti di trasporto



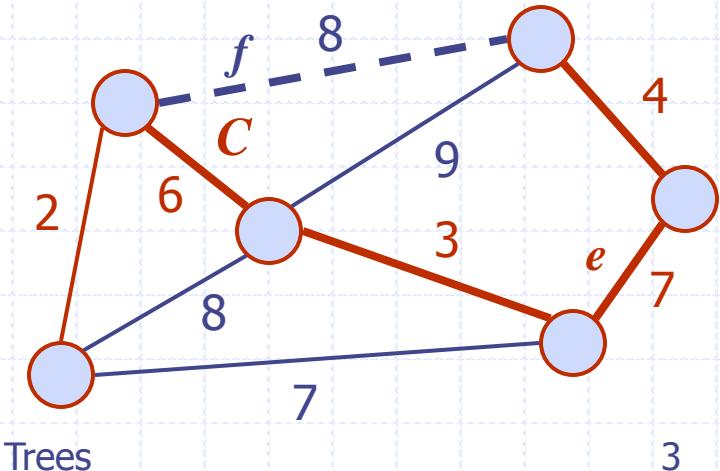
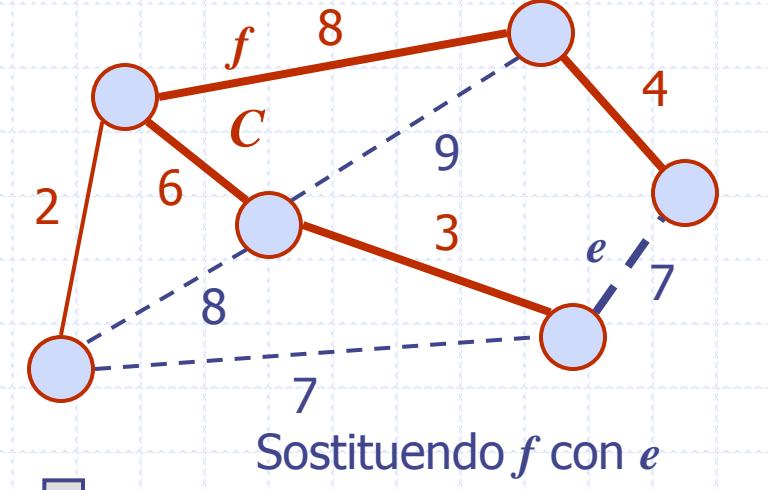
# Proprieta' di Ciclo

## Proprieta' di Ciclo:

- Sia  $T$  il minimum spanning tree di un grafo pesato  $G$
- Sia  $e$  un arco di  $G$  che non e' in  $T$  e  $C$  il ciclo formato da  $e$  in  $T$
- Per ogni arco  $f$  di  $C$ ,  $\text{weight}(f) \leq \text{weight}(e)$

## Prova:

- Per contraddizione
- Se  $\text{weight}(f) > \text{weight}(e)$  possiamo ottenere un albero di peso minore sostituendo  $e$  con  $f$



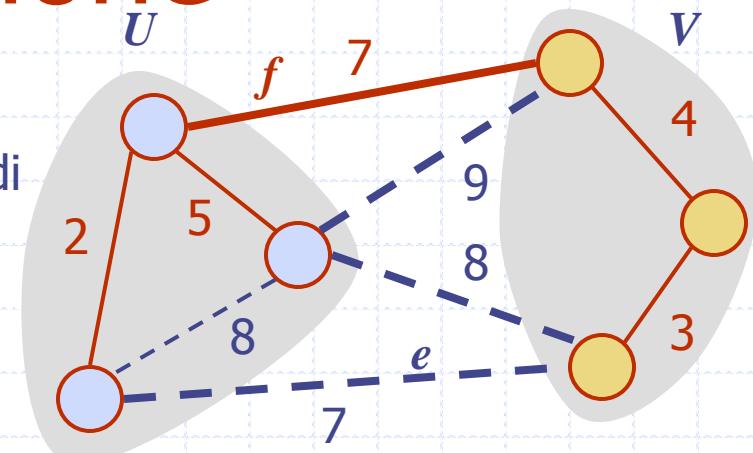
# Proprieta' di Partizione

## Proprieta' di Partizione:

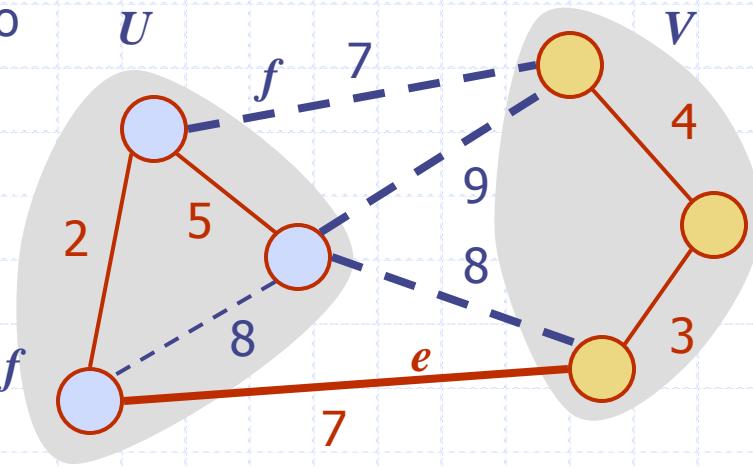
- Considera una partizione dei vertici di  $G$  in due sottoinsiemi  $U$  e  $V$
- Sia  $e$  un arco di peso minimo che attraversa la partizione
- Esiste uno spanning tree di  $G$  contenente  $e$

## Prova:

- Sia  $T$  un MST di  $G$
- Se  $T$  non contiene  $e$ , considera il ciclo  $C$  formato da  $e$  con  $T$  e sia  $f$  un arco di  $C$  che attraversa la partizione
- Dalla proprietà di ciclo,  
$$\text{weight}(f) \leq \text{weight}(e)$$
- Quindi,  $\text{weight}(f) = \text{weight}(e)$
- Otteniamo un altro MST sostituendo  $f$  con  $e$



Sostituendo  $f$  con  $e$  otteniamo  
un altro MST



# Algoritmo di Kruskal

- ◆ una coda di priorita' memorizza gli archi fuori dalla nuvola
  - Chiave: peso
  - Elemento: arco
- ◆ Alla fine dell'algoritmo
  - Otteniamo una nuvola che ricopre lo MST
  - Un albero  $T$  che e' il nostro MST

**Algorithm *KruskalMST(G)***

for each vertex  $V$  in  $G$  do

    define a  $Cloud(v)$  of  $\leftarrow \{v\}$

let  $Q$  be a priority queue.

Insert all edges into  $Q$  using their weights as the key

$T \leftarrow \emptyset$

while  $T$  has fewer than  $n-1$  edges do

    edge  $e = T.removeMin()$

    Let  $u, v$  be the endpoints of  $e$

    if  $Cloud(v) \neq Cloud(u)$  then

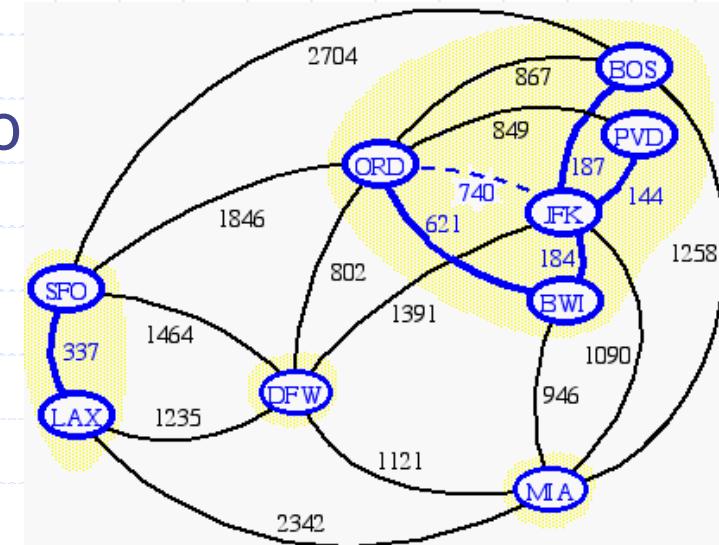
        Add edge  $e$  to  $T$

        Merge  $Cloud(v)$  and  $Cloud(u)$

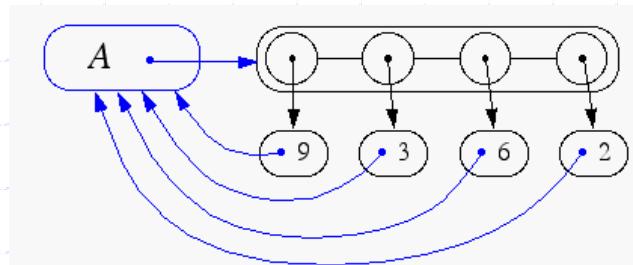
return  $T$

# Struttura Dati per l'Algoritmo di Kruskal

- ◆ L'algoritmo mantiene una foresta di alberi
- ◆ Un arco e' accettato se unisce due alberi distinti
- ◆ Necessita una struttura dati che mantiene una **partizione**, cioe' una collezione di insiemi disgiunti, con le operazioni:
  - find**(u): restituisce l'insieme che memorizza u
  - union**(u,v): sostituisce gli insiemi che memorizzano u e v con la loro unione



# Rappresentazione di una Partizione



- ◆ Ogni insieme e' memorizzato in una sequenza
- ◆ Ogni elemento ha un riferimento all'insieme
  - operazione **find(u)** costa  $O(1)$ , e restituisce l'insieme acui  $u$  appartiene.
  - nell'operazione **union(u,v)**, muoviamo gli elementi dall'insieme piu' piccolo alla sequenza dell'insieme piu' grande e aggiorniamo i loro riferimenti
  - il tempo dell'operazione **union(u,v)** e'  $\min(n_u, n_v)$ , dove  $n_u$  e  $n_v$  sono le dimensioni degli insiemi che memorizzano  $u$  e  $v$
- ◆ Quando un elemento e' processato, viene mosso in un insieme di dimensione almeno doppia, quindi ogni elemento e' processato al piu' log  $n$  volte

# Implementazione basata sulla Partizione

- ◆ Una versione basata sulla partizione dell'algoritmo di Kruskal realizza fusioni con union e test con find.

## Algorithm Kruskal( $G$ )

**Input:** A weighted graph  $G$ .

**Output:** An MST  $T$  for  $G$ .

Let  $P$  be a partition of the vertices of  $G$ , where each vertex forms a separate set.

Let  $Q$  be a priority queue storing the edges of  $G$ , sorted by their weights

Let  $T$  be an initially-empty tree

**while**  $Q$  is not empty **do**

$(u,v) \leftarrow Q.\text{removeMinElement}()$

**if**  $P.\text{find}(u) \neq P.\text{find}(v)$  **then**

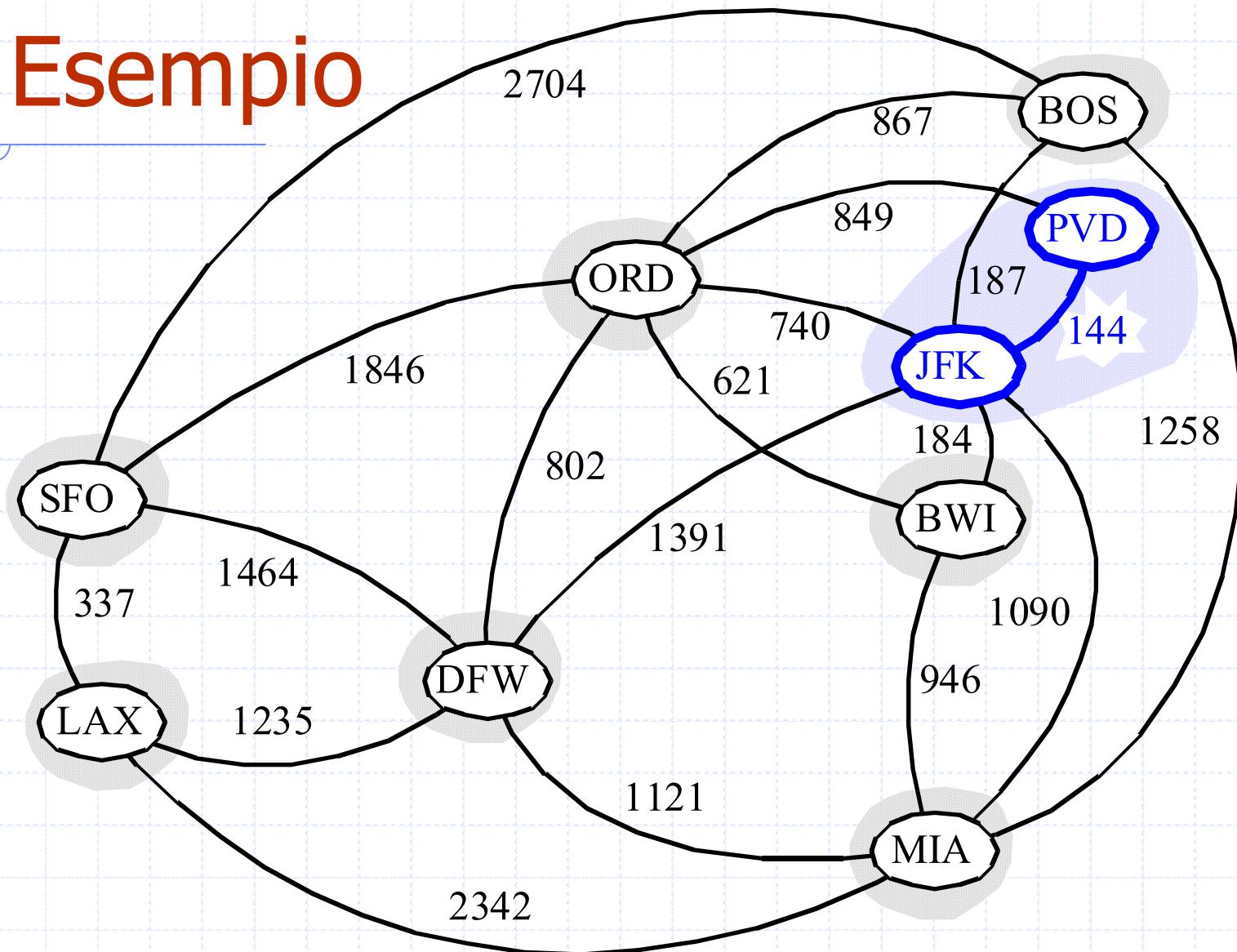
        Add  $(u,v)$  to  $T$

$P.\text{union}(u,v)$

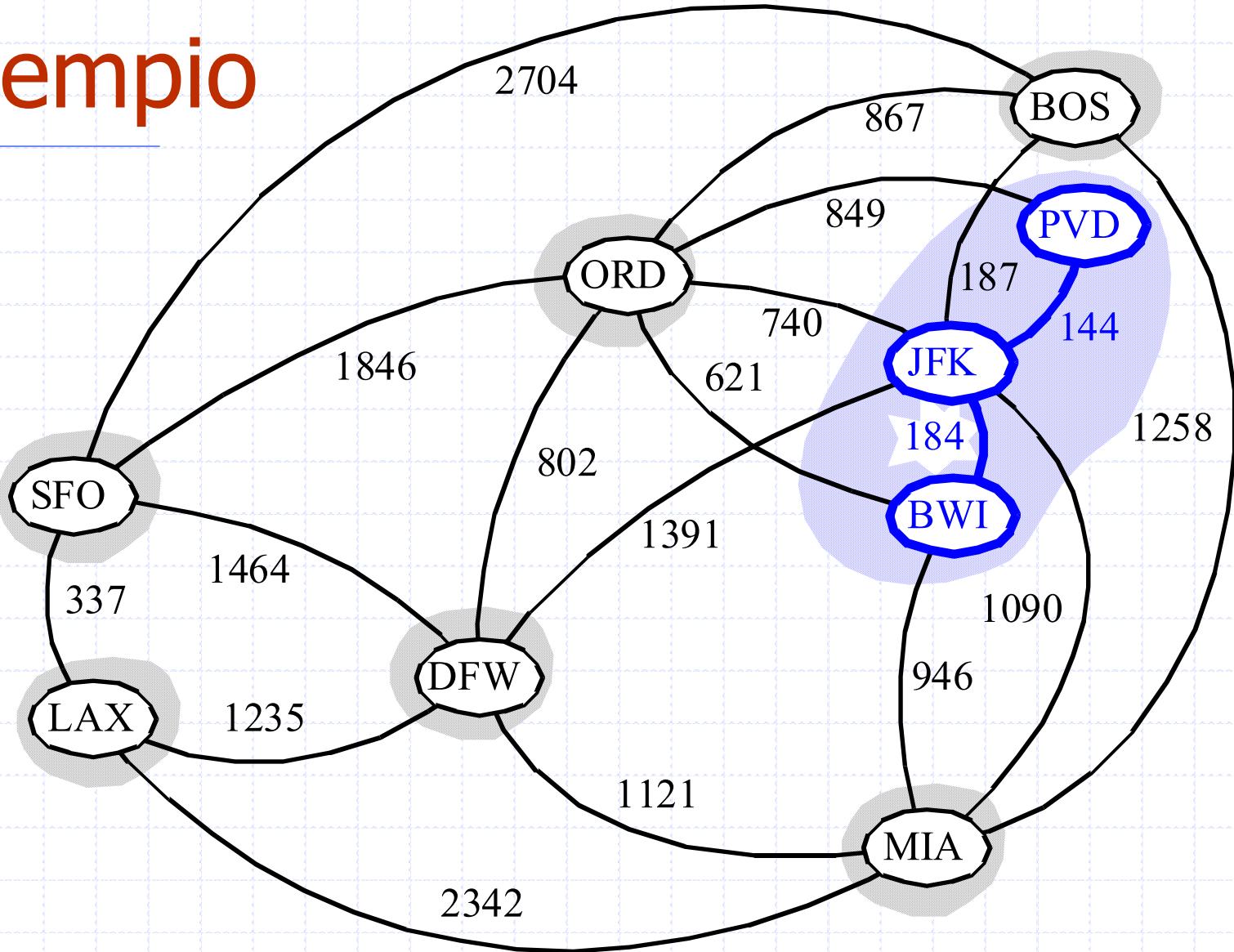
**return**  $T$

Running time:  
 $O((n+m)\log n)$

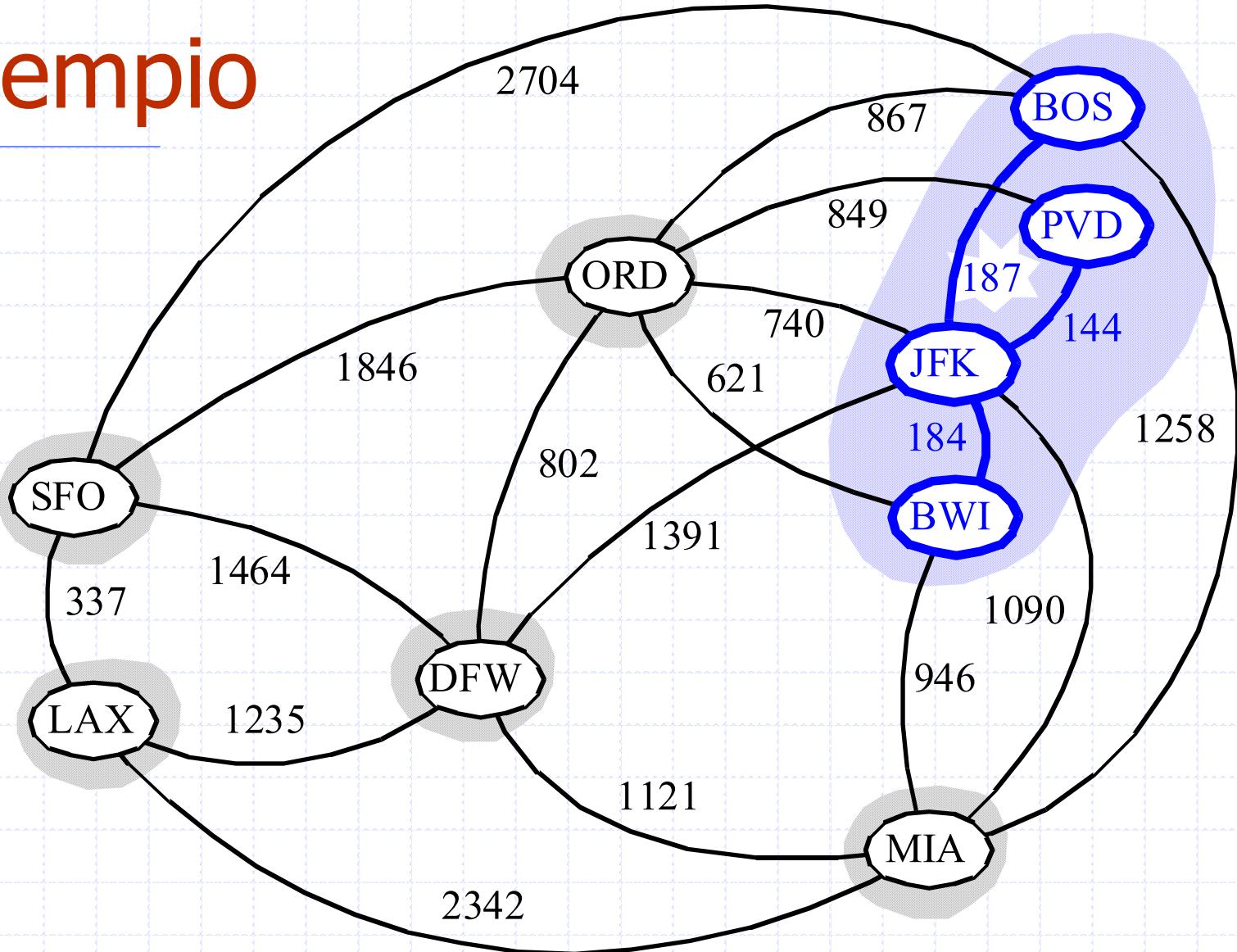
# Kruskal Esempio



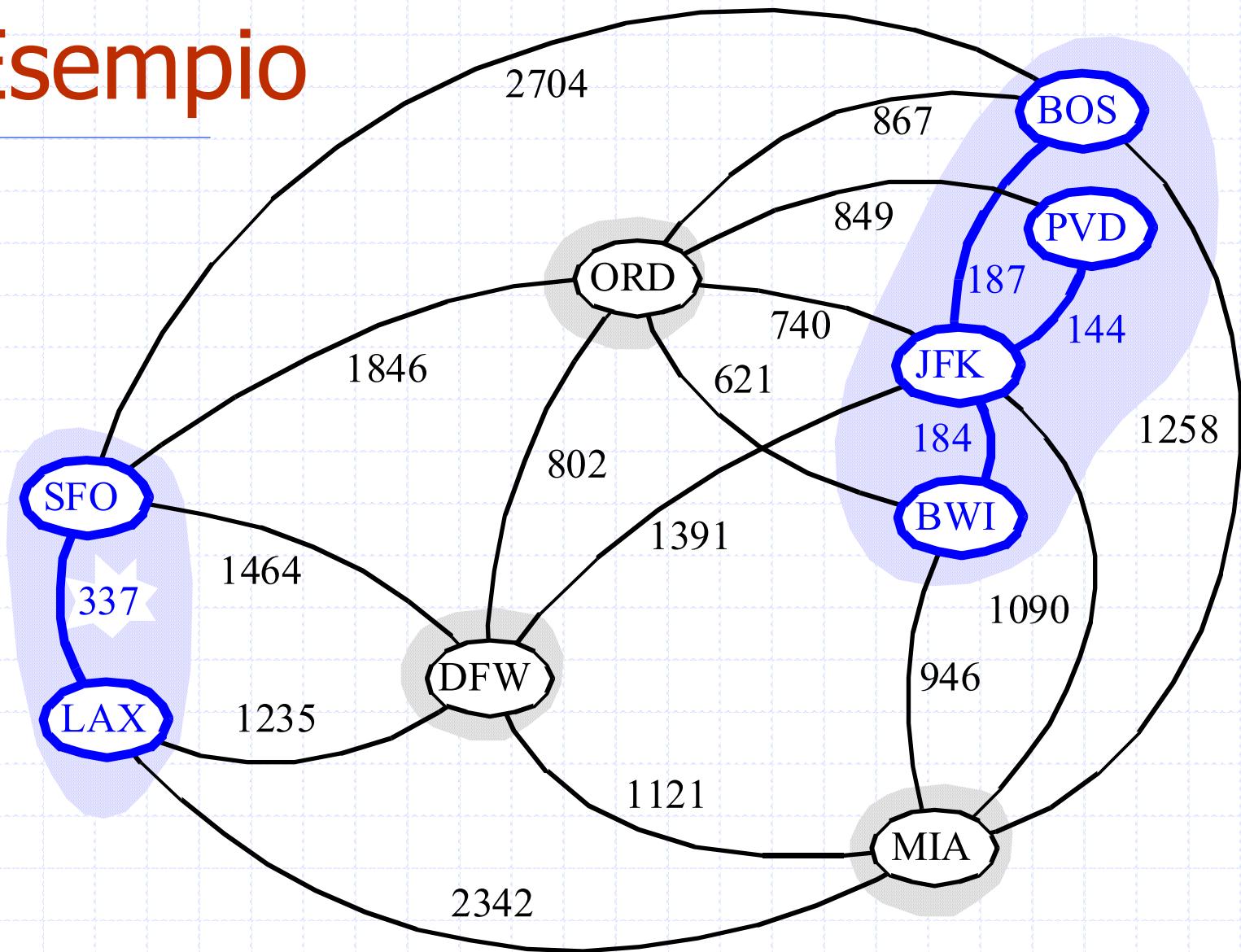
# Esempio



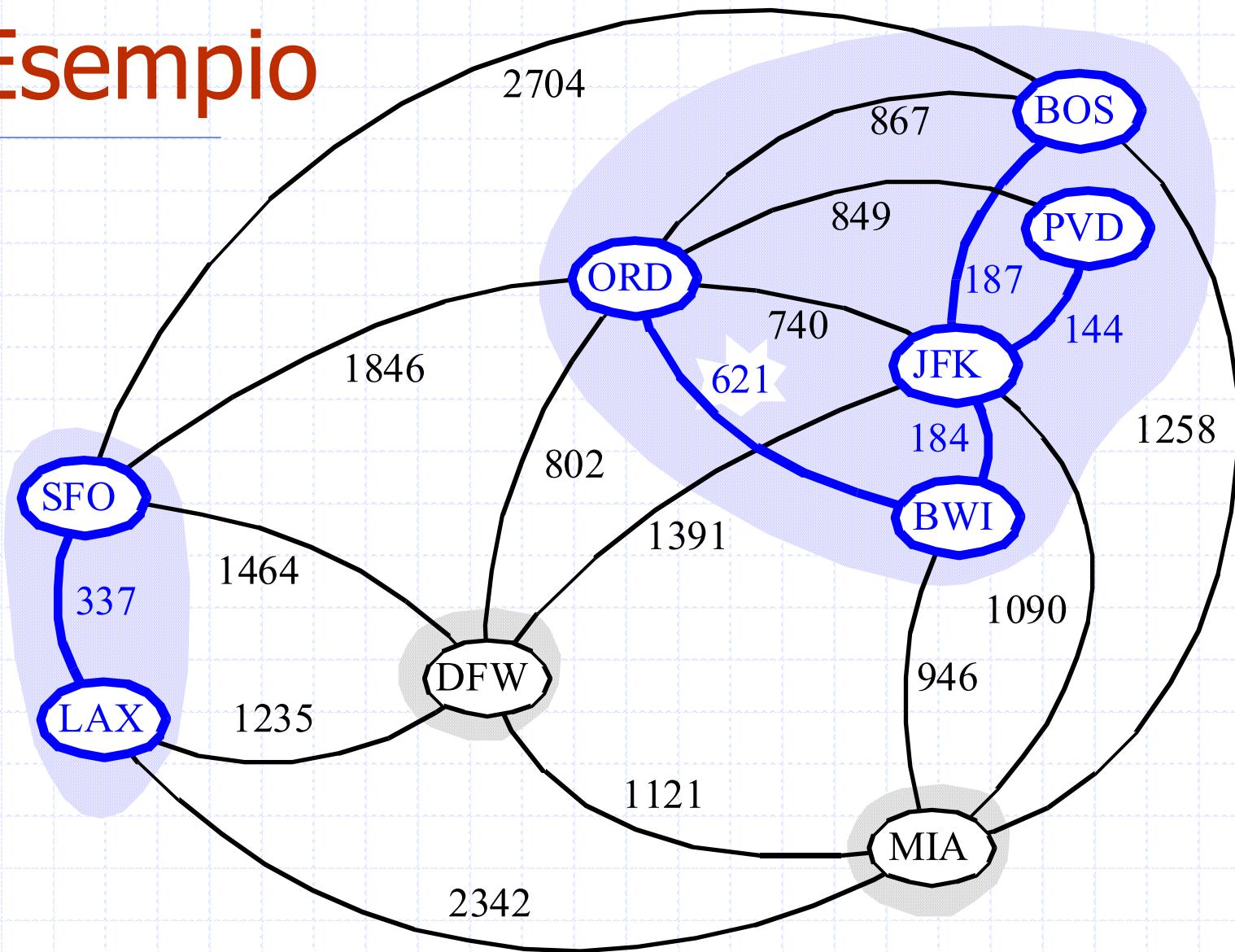
# Esempio



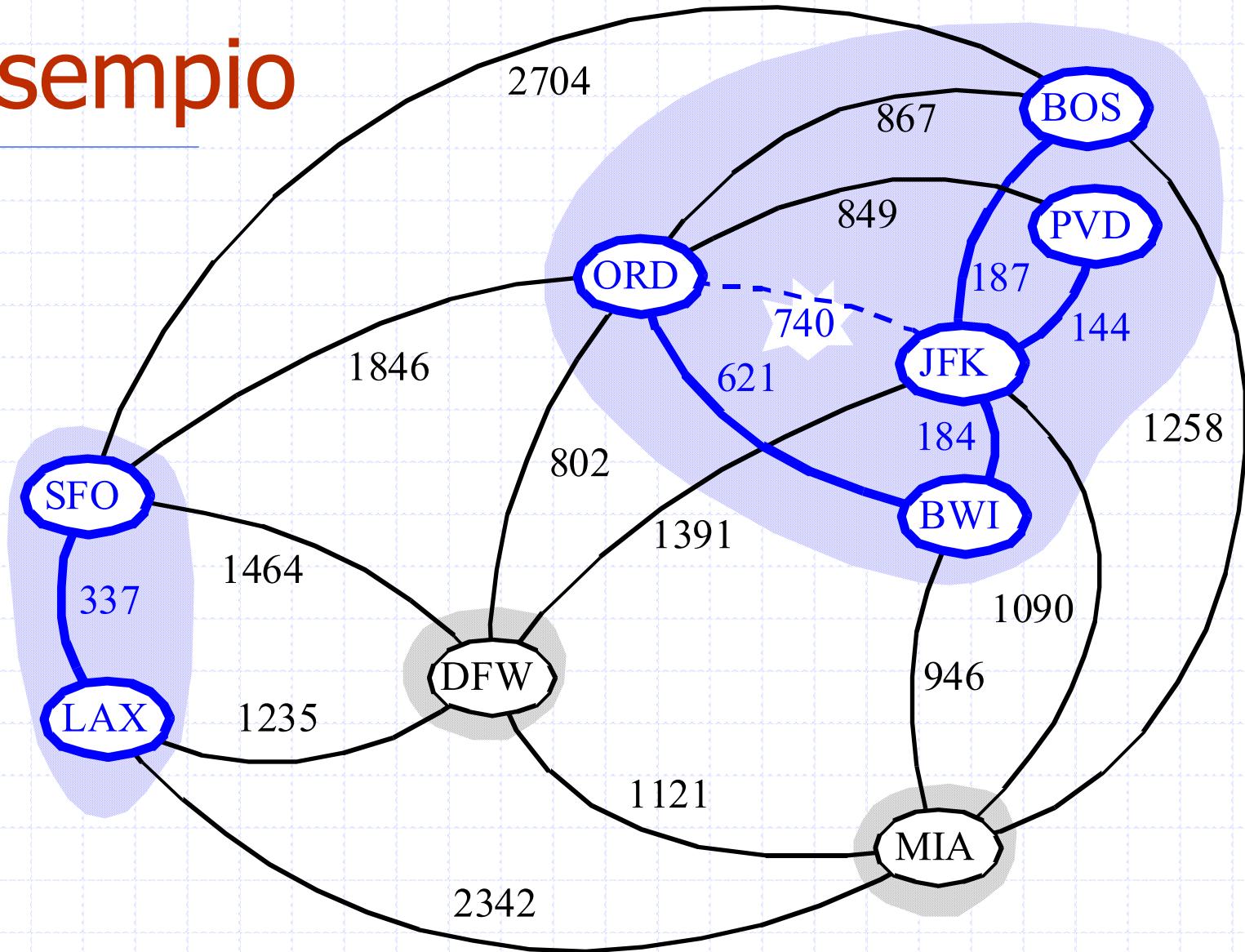
# Esempio



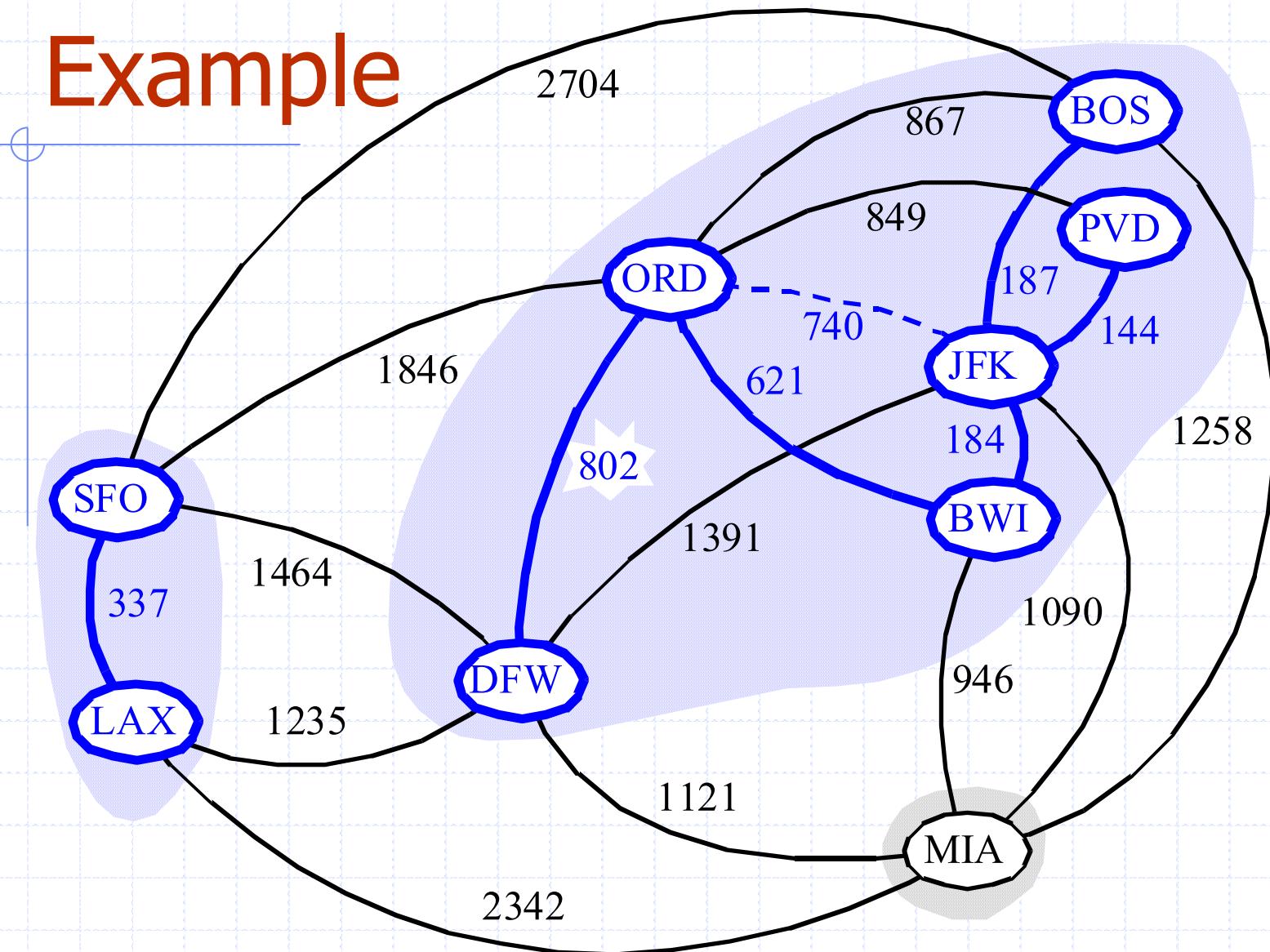
# Esempio



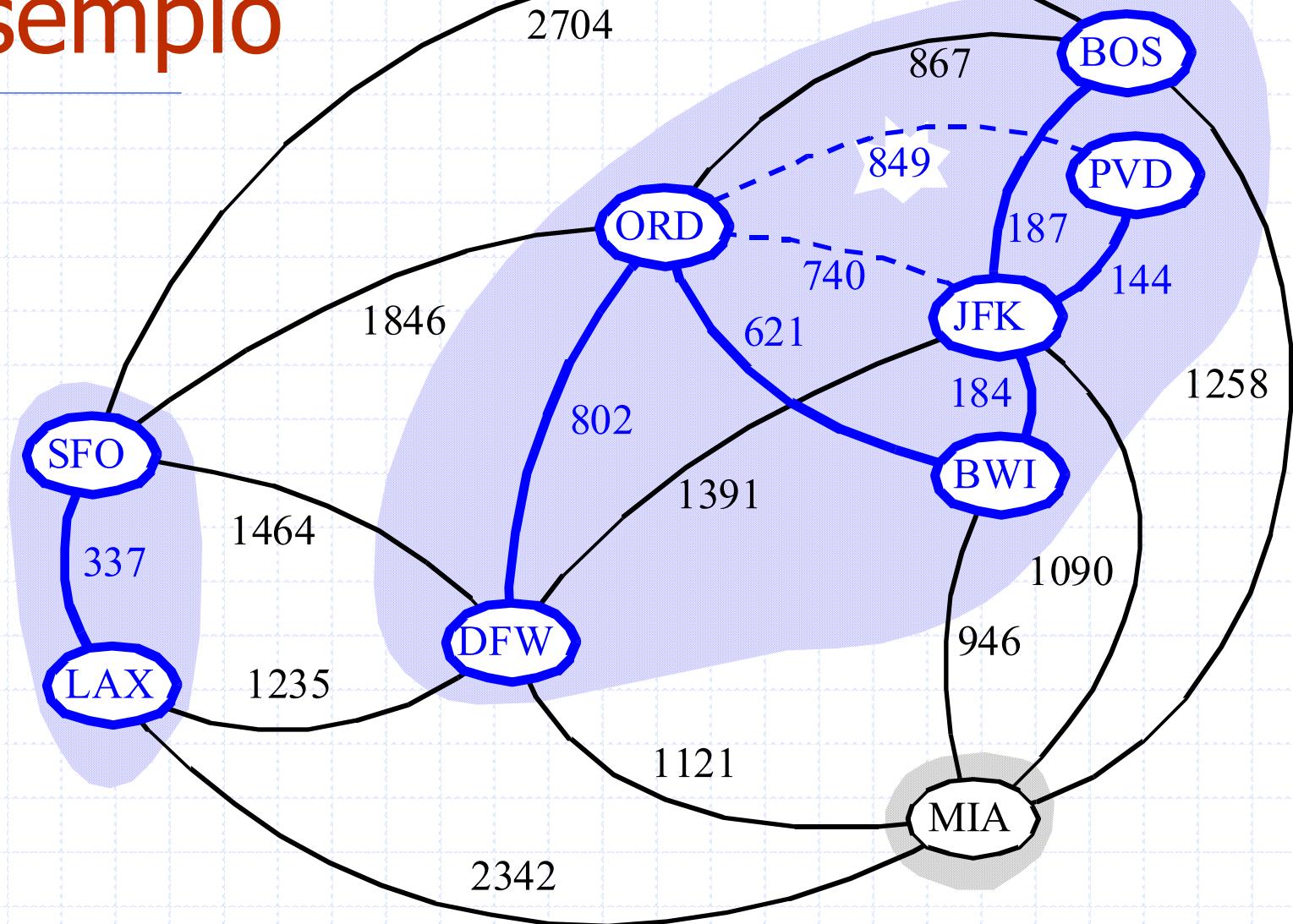
# Esempio



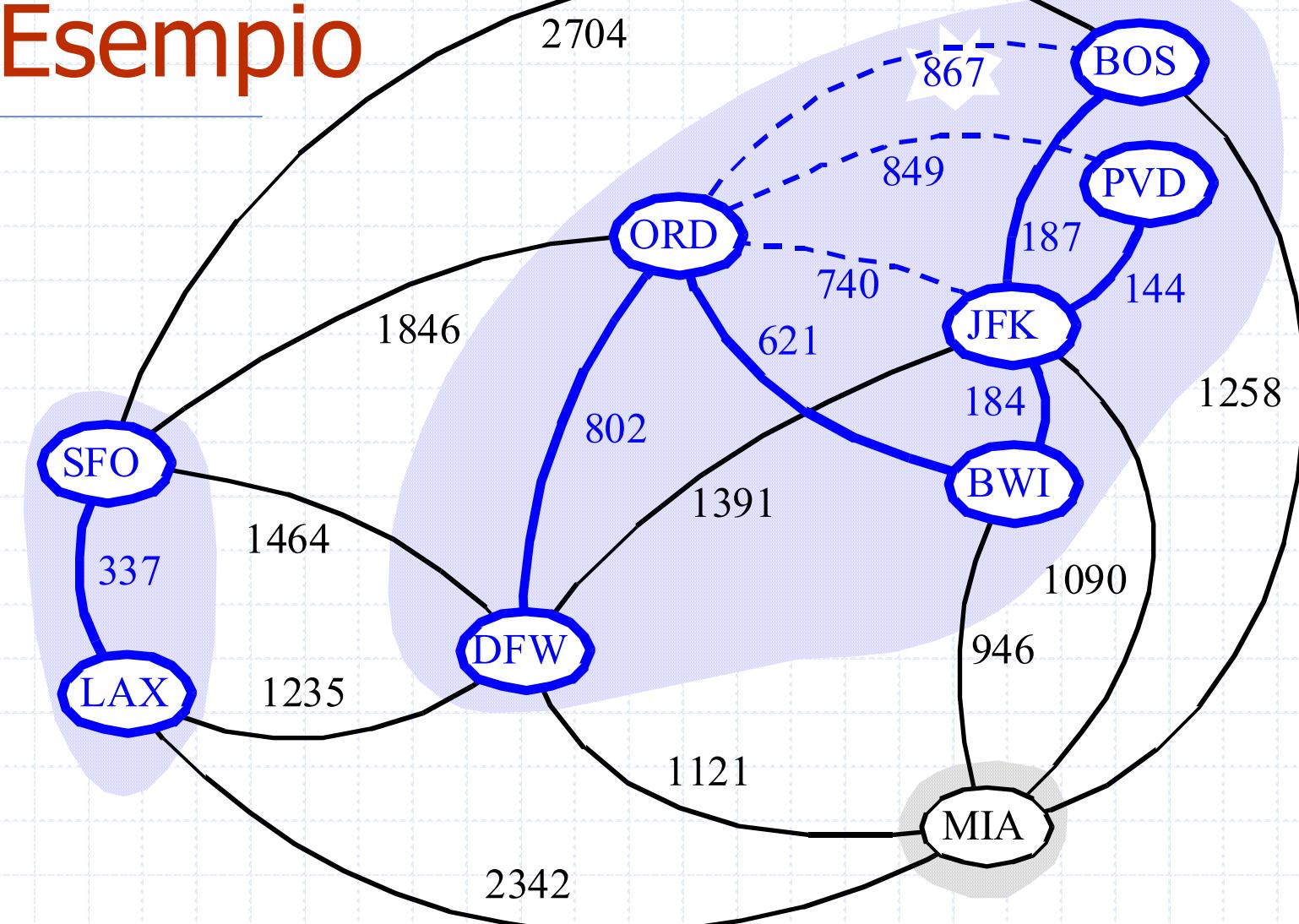
# Example



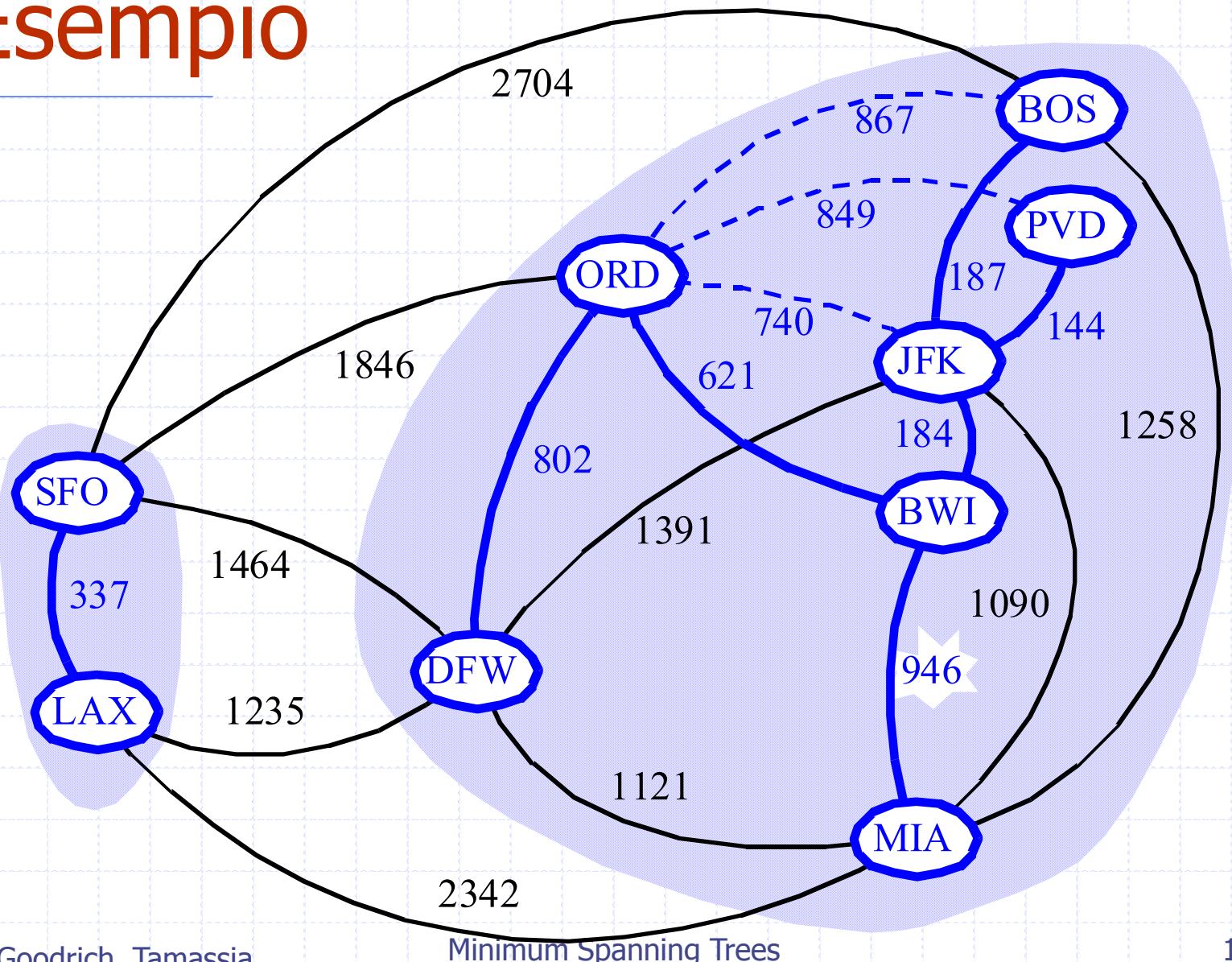
# Esempio



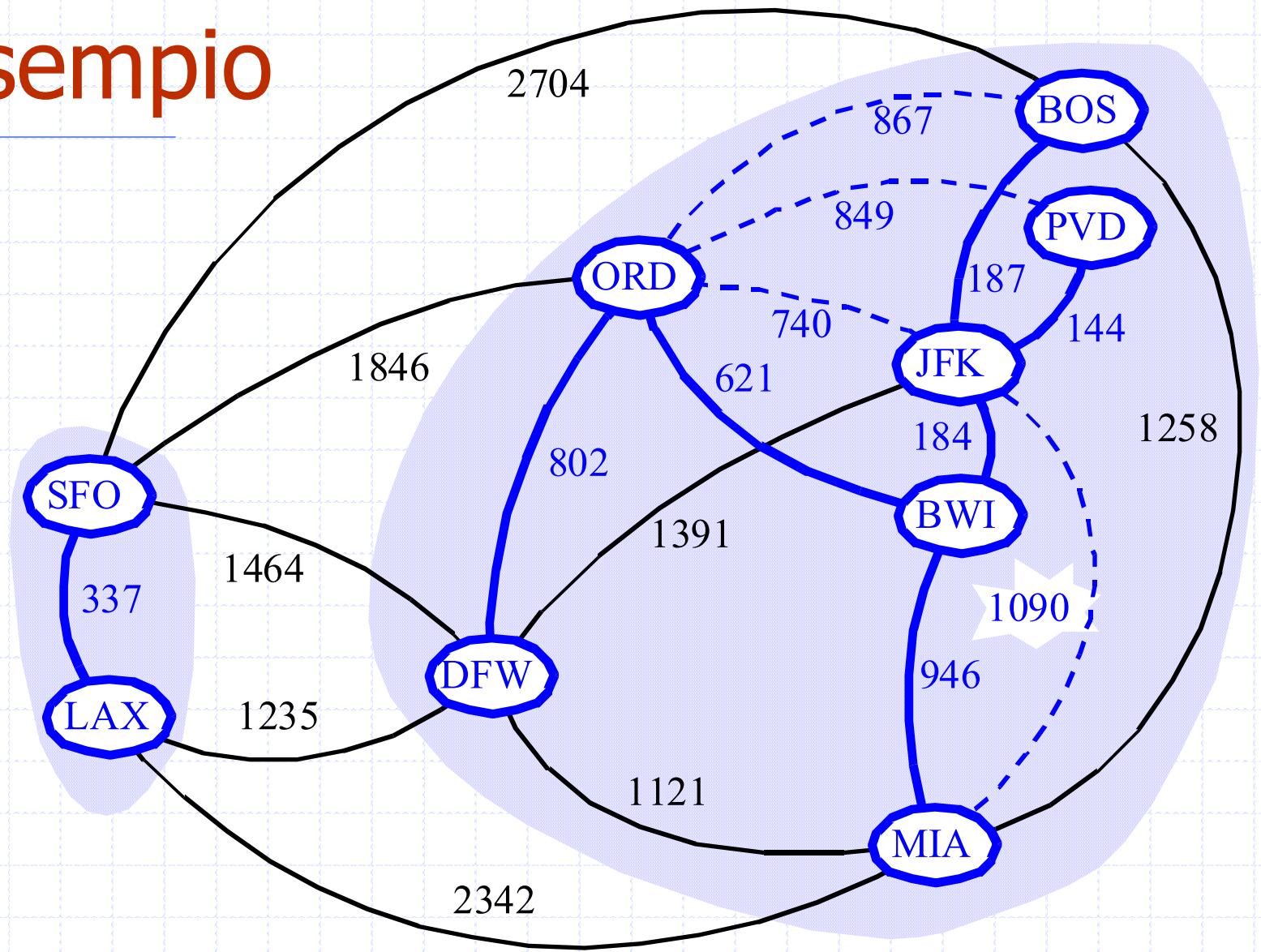
# Esempio



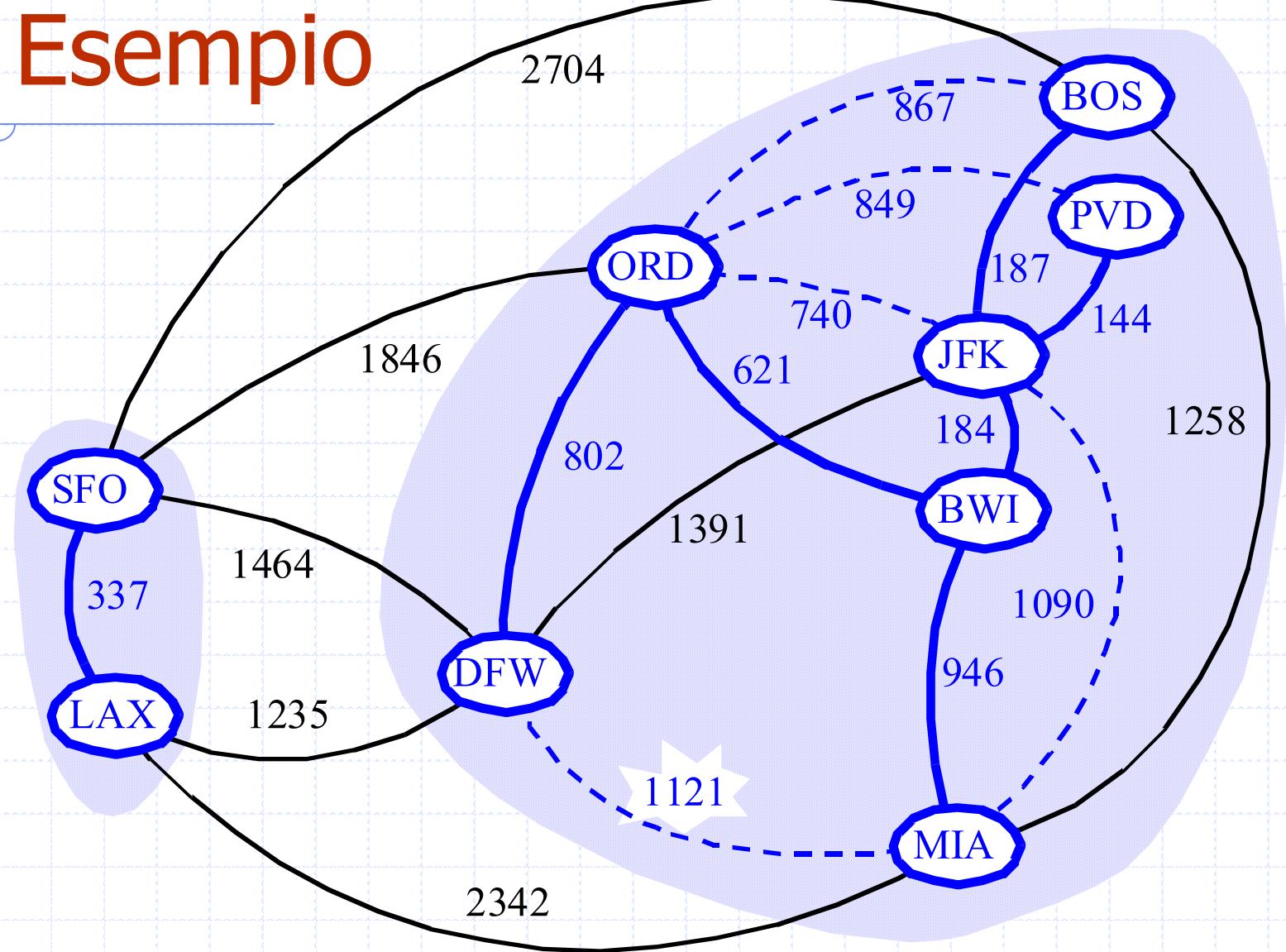
# Esempio



# Esempio

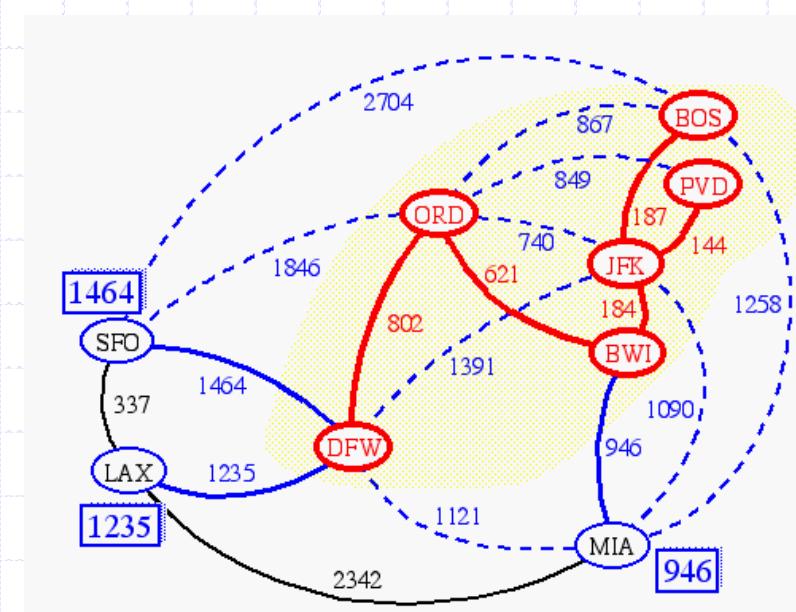


# Esempio



# Algoritmo di Prim-Jarnik's

- ◆ Simile all'algoritmo di Dijkstra (per un grafo connesso)
- ◆ Prendiamo un vertice arbitrario  $s$  e cresciamo il MST come una nuvola di vertici, iniziando da  $s$
- ◆ Memorizziamo per ogni vertice  $v$  un'etichetta  $d(v) = \text{il piu' piccolo peso di un arco che connette } v \text{ ad un vertice nella nuvola}$
- ◆ Ad ogni passo:
  - Aggiungiamo alla nuvola il vertice  $u$  fuori la nuvola con la piu' piccola etichetta di distanza
  - Aggiorniamo le etichette dei vertici adiacenti a  $u$

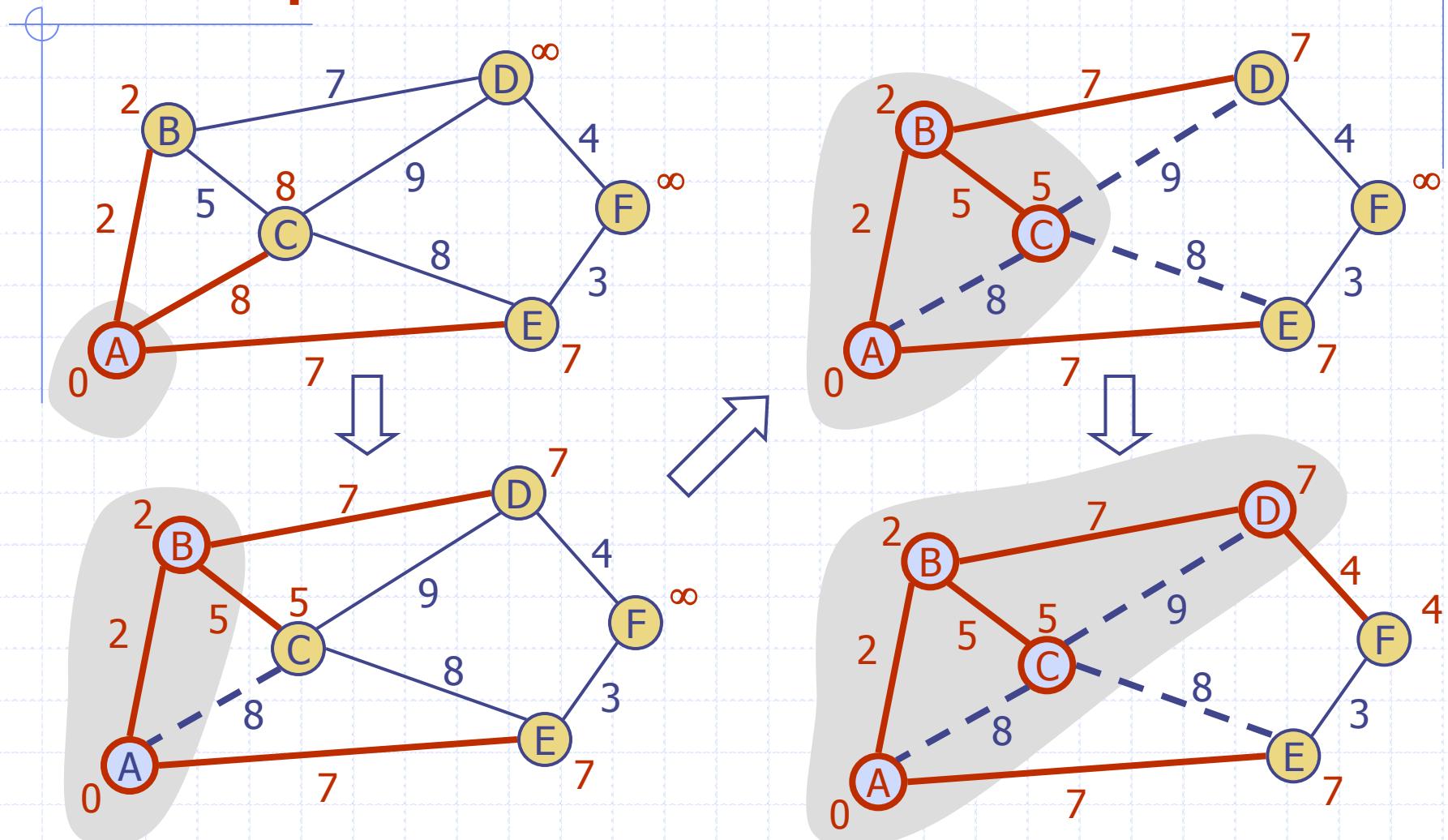


# Algoritmo di Prim-Jarnik

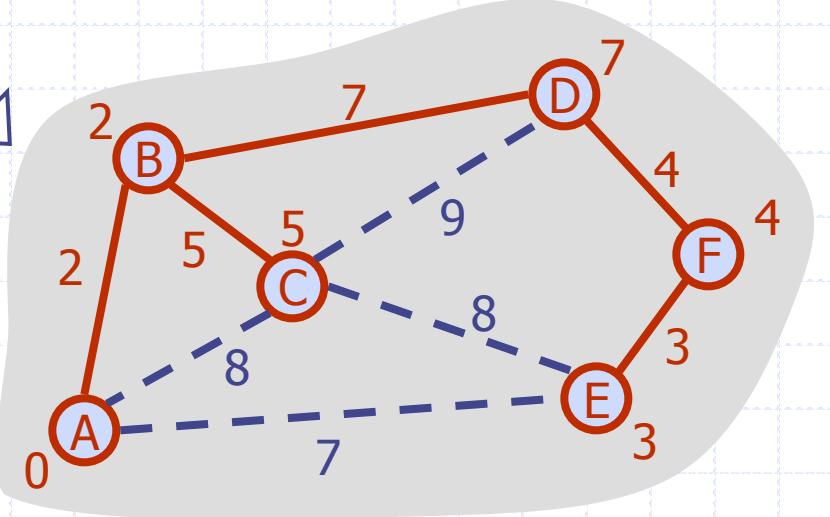
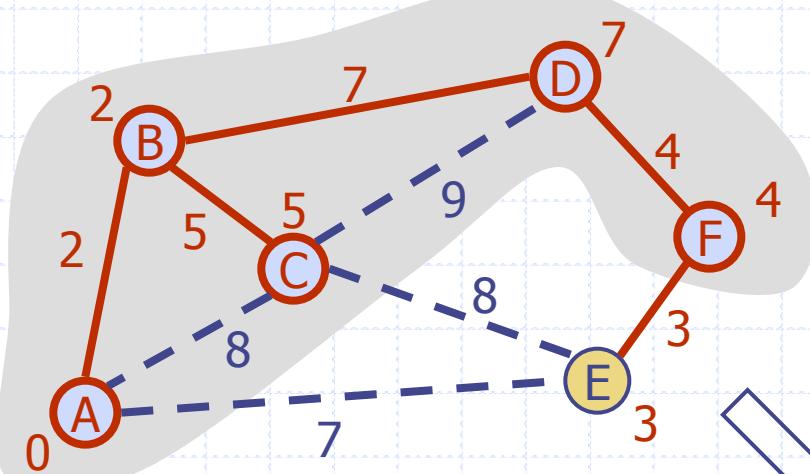
- ◆ Una coda di priorita' memorizza i vertici fuori della nuvola
  - Chiave: distanza
  - Elemento: vertice
- ◆ Metodo basato su Locator
  - *insert(k,e)* restituisce un locator
  - *replaceKey(l,k)* modifica la chiave di un elemento
- ◆ Memorizziamo tre etichette per ogni vertice:
  - Distanza
  - Arco Parent nell'MST
  - Locator nella coda di priorita'

```
Algorithm PrimJarnikMST(G)
   $Q \leftarrow$  new heap-based priority queue
   $s \leftarrow$  a vertex of  $G$ 
  for all  $v \in G.vertices()$ 
    if  $v = s$ 
      setDistance( $v, 0$ )
    else
      setDistance( $v, \infty$ )
      setParent( $v, \emptyset$ )
       $l \leftarrow Q.insert(getDistance(v), v)$ 
      setLocator( $v, l$ )
  while  $\neg Q.isEmpty()$ 
     $u \leftarrow Q.removeMin()$ 
    for all  $e \in G.incidentEdges(u)$ 
       $z \leftarrow G.opposite(u, e)$ 
       $r \leftarrow weight(e)$ 
      if  $r < getDistance(z)$ 
        setDistance( $z, r$ )
        setParent( $z, e$ )
         $Q.replaceKey(getLocator(z), r)$ 
```

# Esempio



# Esempio(cont.)



# Analisi

## ◆ Operazioni su grafi

- Metodo `incidentEdges` e' chiamato una volta per ogni vertice

## ◆ Operazioni sulle etichette

- Assegna/accede  $O(\deg(z))$  volte le etichette distanza e locator di un vertice  $z$
- Assegna/accede un'etichetta in tempo  $O(1)$

## ◆ Operazioni sulla coda di priorita'

- Ogni vertice e' inserito e rimosso una volta dalla coda di priorita', dove ogni inserimento o rimozione ha costo  $O(\log n)$
- La chiave di un vertice nella coda di priorita' e' modificata al massimo  $\deg(w)$  volte, dove ogni modifica di chiave costa  $O(\log n)$

## ◆ Prim-Jarnik's ha costo $O((n + m) \log n)$ se il grafo e' rappresentato con una lista di adiacenza

- Ricorda che  $\sum_v \deg(v) = 2m$

## ◆ Il tempo di esecuzione e' anche $O(m \log n)$ poiche' il grafo e' connesso

# Baruvka's Algorithm (Ex. C-12.28)

- ◆ Like Kruskal's Algorithm, Baruvka's algorithm grows many "clouds" at once.

**Algorithm *BaruvkaMST*( $G$ )**

$T \leftarrow V$  {just the vertices of  $G$ }

**while**  $T$  has fewer than  $n-1$  edges **do**

**for each** connected component  $C$  in  $T$  **do**

        Let edge  $e$  be the smallest-weight edge from  $C$  to another component in  $T$ .

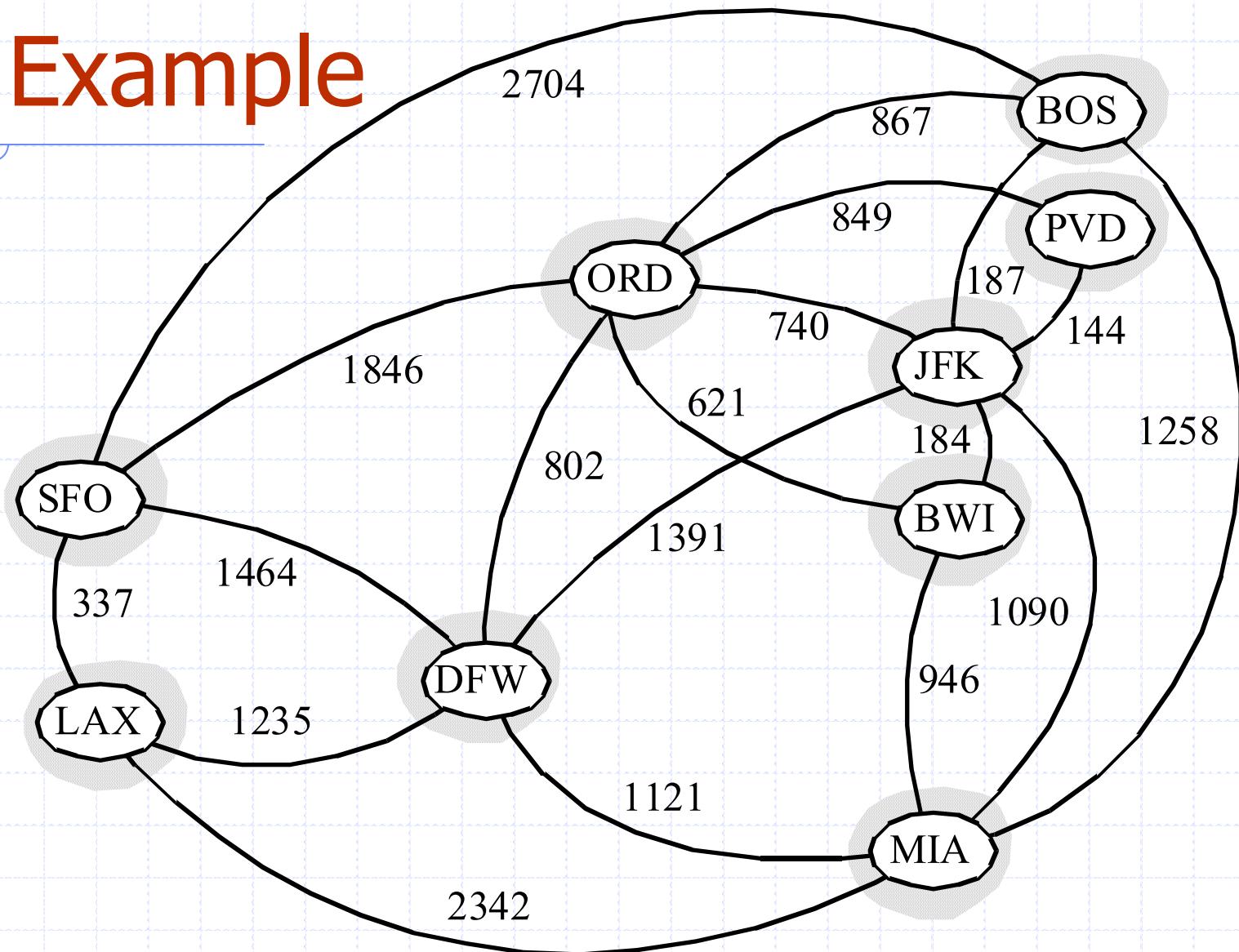
**if**  $e$  is not already in  $T$  **then**

            Add edge  $e$  to  $T$

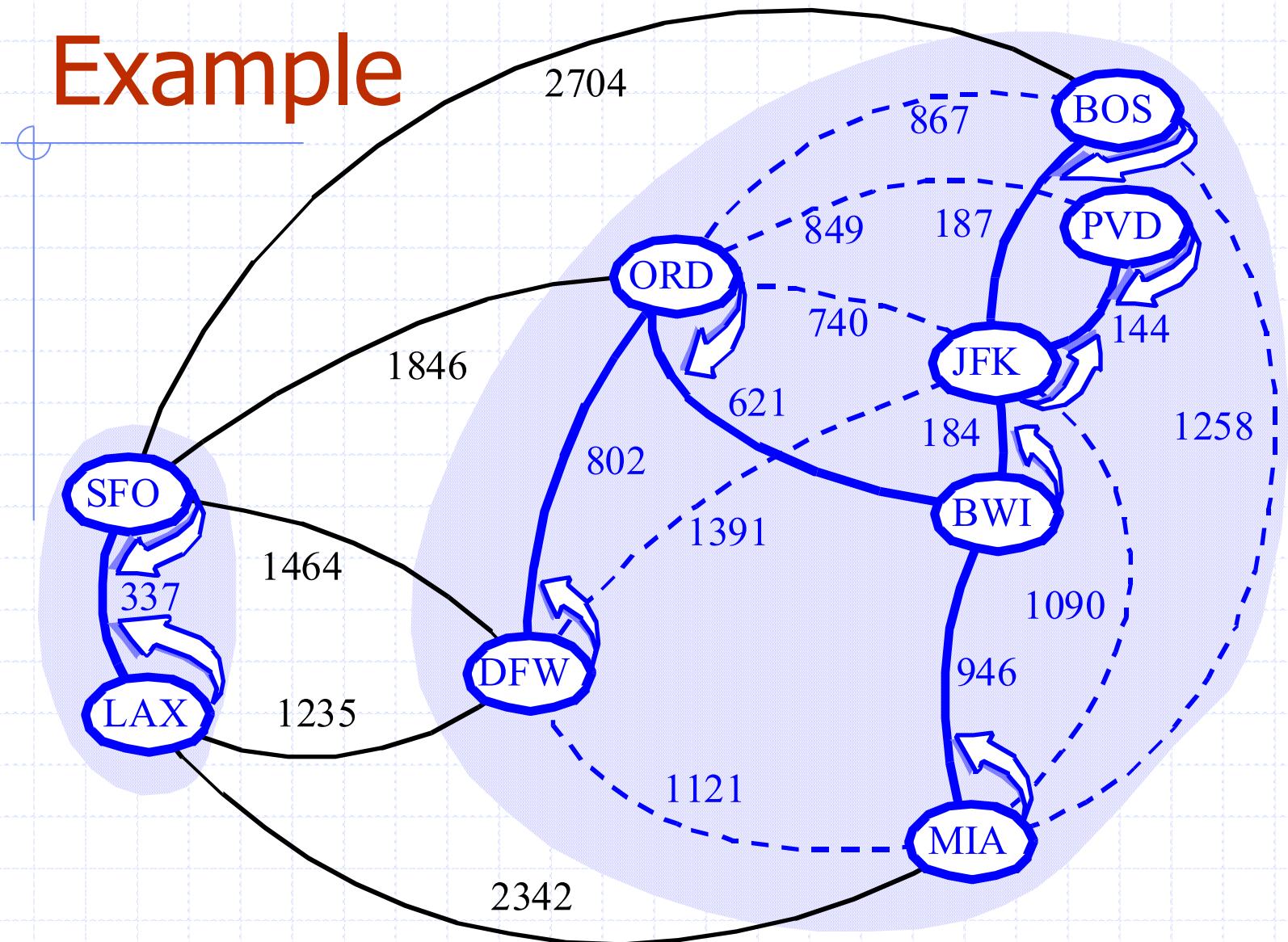
**return**  $T$

- ◆ Each iteration of the while-loop halves the number of connected components in  $T$ .
  - The running time is  $O(m \log n)$ .

# Baruvka Example



# Example



# Example

