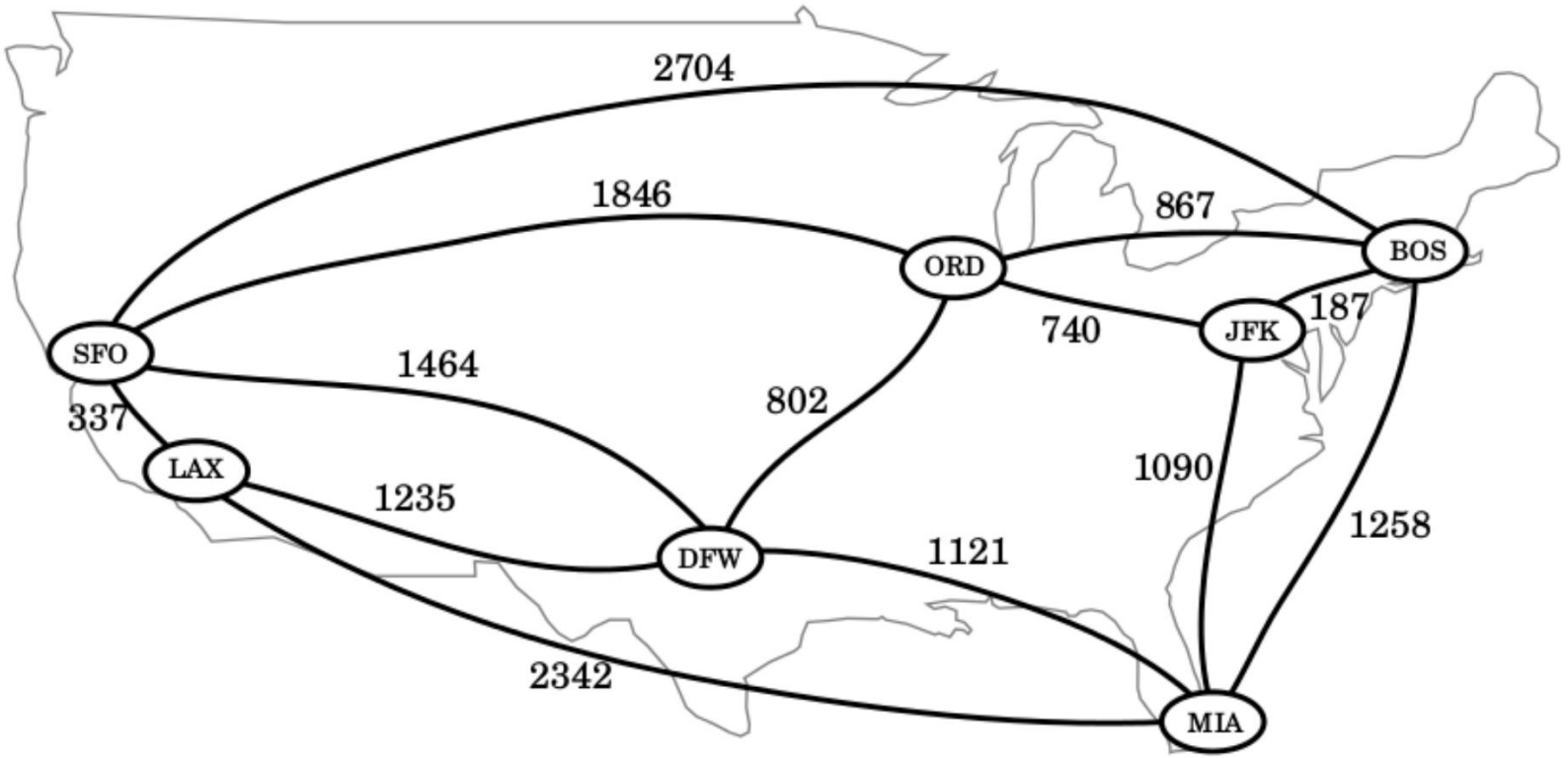
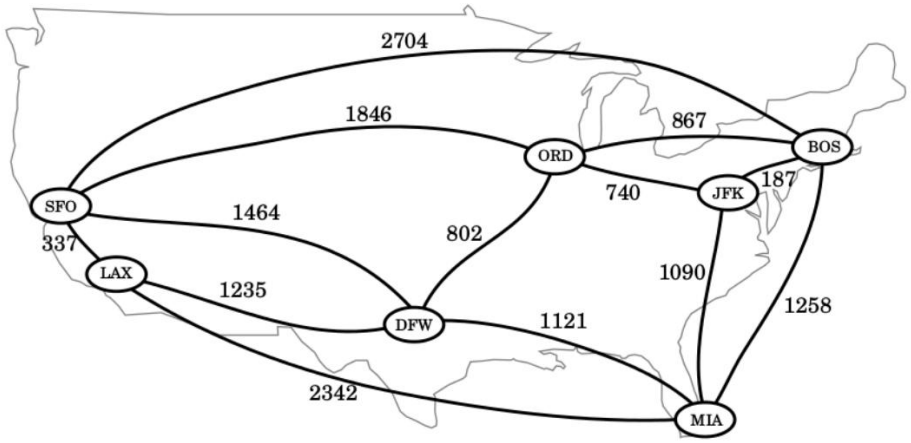
Percorsi più brevi

- La strategia di ricerca BFS può essere utilizzata per trovare il percorso più breve da un vertice iniziale v a ogni altro vertice in un grafo connesso
- Questo approccio è utile quando ogni arco è buono come ogni altro, in particolare fallisce quando si deve prendere in considerazione un grafico pesato
- Un grafico pesato è un grafico che ha un'etichetta numerica w(e associata a ciascun bordo e (il peso di quel bordo)
- Per e = (u, v) abbiamo w(u, v) = w(e)

Percorsi più brevi



Percorsi più brevi



Possibili percorsi da JFK a LAX:

Sentiero	Pesi 740	Totale
JFK-ORD-SFO-LAX	+ 1846 + 337	2923
JFK-ORD-DFW-LAX	740+802+1235	2777
JFK-MIA-LASS	1090 + 2342	3432
JFK-MIA-DFW-LAX	1090+1121+1235	3446

JFK - BOS - ORD - OFS - LAX 187 + 867 + 1846 + 337 3237

JFK-BOS-ORD-DFW-LAX 187 + 867 + 802 + 1235 3091

Percorsi più brevi

Il peso (o lunghezza) di un percorso P in un grafo pesato G è la somma dei pesi degli archi di P.

Se P = ((v0, v1),(v1, v2), . . . ,(vkÿ1, vk)) allora

$$w(P) = \sum_{io=0}^{kÿ1} w(vi , vi+1)$$

La distanza da un vertice u a un vertice v in G, indicata con d(u, v) è la lunghezza di un percorso di lunghezza minima (percorso più breve) da u a v (se tale percorso esiste)

Percorsi più brevi

Algoritmo di Dijkstra

- Una classe di algoritmi risolve il problema di trovare un percorso minimo da alcuni vertici a ciascun altro vertice in un grafo pesato G
- Uno di questi, l'algoritmo di Dijkstra, applica il metodo greedy: un dato problema viene risolto selezionando ripetutamente la scelta migliore tra quelle disponibili ad ogni iterazione
- L'idea principale è quella di eseguire una ricerca BFS ponderata partendo dal vertice sorgente s e creando una "nuvola" di vertici, ciascuno dei quali entra nella nuvola in ordine di distanza da s
- Quindi, in ogni iterazione, il vertice successivo scelto è il vertice esterno alla nuvola più vicino a s
- L'algoritmo termina quando non ci sono più vertici all'esterno del cloud (o quando quelli all'esterno del cloud non sono collegati a quelli all'interno del cloud)

Percorsi più brevi

Rilassamento dei bordi

- Definiamo un'etichetta D[v] per ogni vertice v in V
- Queste etichette memorizzeranno sempre la lunghezza del percorso migliore ottenuto finora da s a v
- Inizialmente D[s] = 0 e D[v] = ÿ per ogni v ÿ= s
- Definiamo l'insieme C (la “nuvola” di vertici) come l'insieme vuoto
- Ad ogni iterazione viene selezionato un vertice u non in C con il minimo Etichetta D[u] e u viene inserito in C

Percorsi più brevi

Rilassamento dei bordi

- Una volta che un nuovo vertice u viene inserito in C, l'etichetta D[v] di ciascun vertice adiacente a u e all'esterno di C viene aggiornata, per riflettere il fatto che potrebbe essere un modo nuovo e migliore per raggiungere v tramite u
- Questo aggiornamento è noto come rilassamento, perché prende una vecchia stima e verifica se può essere migliorata per avvicinarsi al suo valore

Più precisamente:

---

se D[u] + w(u,v) < D[v] allora
D[v] = D[u] + w(u,v)

Percorsi più brevi

Rilassamento dei bordi

Algoritmo Percorso più breve(G,s):
D[s] = 0 D[v]
= infinito per ogni vertice v != s

Sia una coda con priorità Q che contenga tutti i vertici di G utilizzando le etichette D come chiavi

mentre Q non è vuoto do u = valore restituito da Q.remove_min() per ogni vertice v adiacente a u tale che v sia in Q
Fare
se D[u] + w(u,v) < D[v] allora
D[v] = D[u] + w(u,v)
Cambiare in D[v] la chiave del vertice v in Q
restituisce l'etichetta D[v] di ogni vertice v

Percorsi più brevi

Costo computazionale

. . .

Sia una coda con priorità Q che contenga tutti i vertici di G utilizzando le etichette D come chiavi

mentre Q non è vuoto do u = valore
restituito da Q.remove_min() per ogni vertice v
adiacente a u tale che v sia in Q
Fare
se D[u] + w(u,v) < D[v] allora
D[v] = D[u] + w(u,v)
Cambiare in D[v] la chiave del vertice v in Q
restituisce l'etichetta D[v] di ogni vertice v

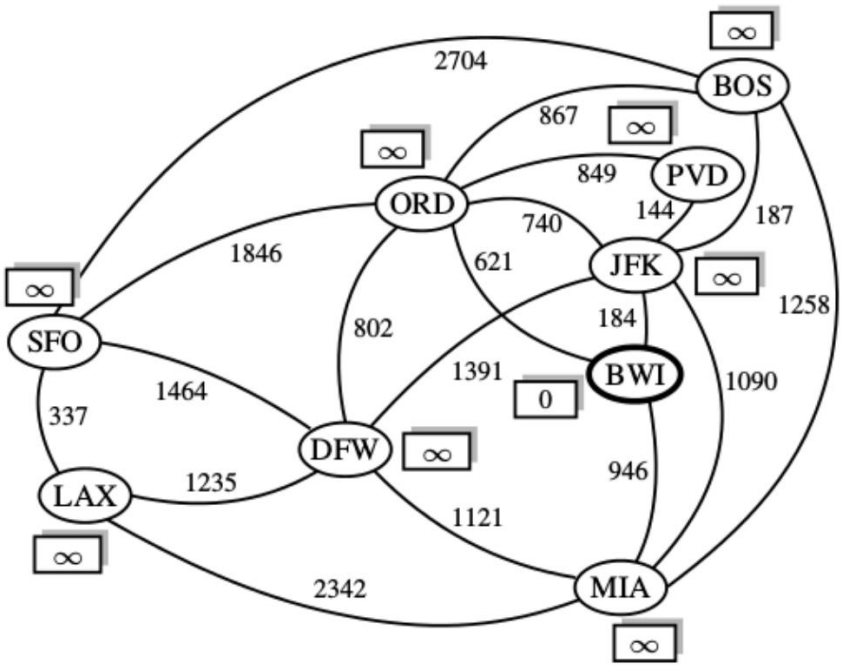
Percorsi più brevi

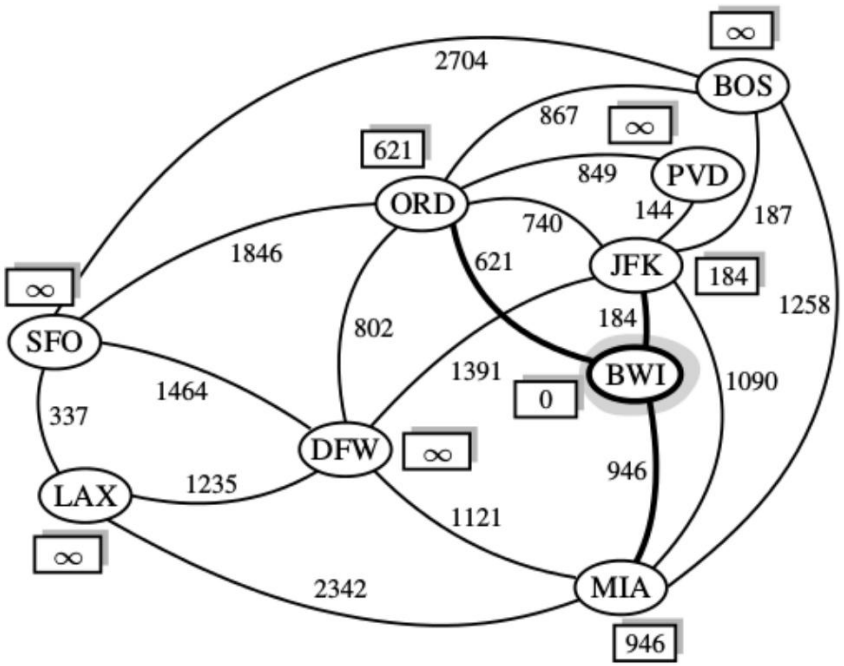
Costo computazionale

- Il ciclo for nidificato viene eseguito nel tempo O(m).
- Il ciclo while esterno viene eseguito in tempo O(n).
- L'aggiornamento di Q richiede tempo O(m).
- Se Q è una coda con priorità, viene eseguita ciascuna delle operazioni precedenti O(logn), quindi il tempo di esecuzione complessivo è O((n + m)logn), ovvero (circa) O(n ^{2} log)

Percorsi più brevi

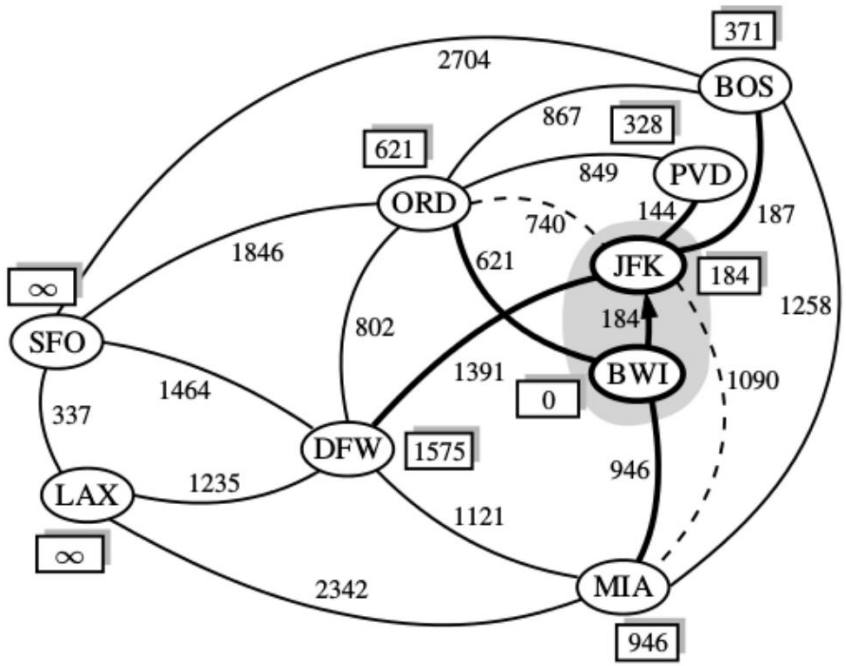
Algoritmo di Dijkstra

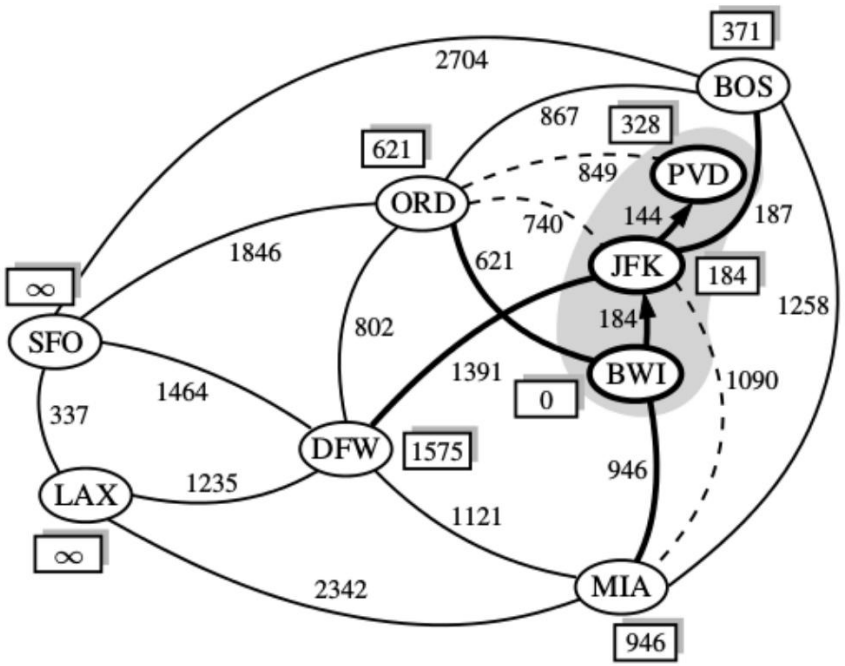




Percorsi più brevi

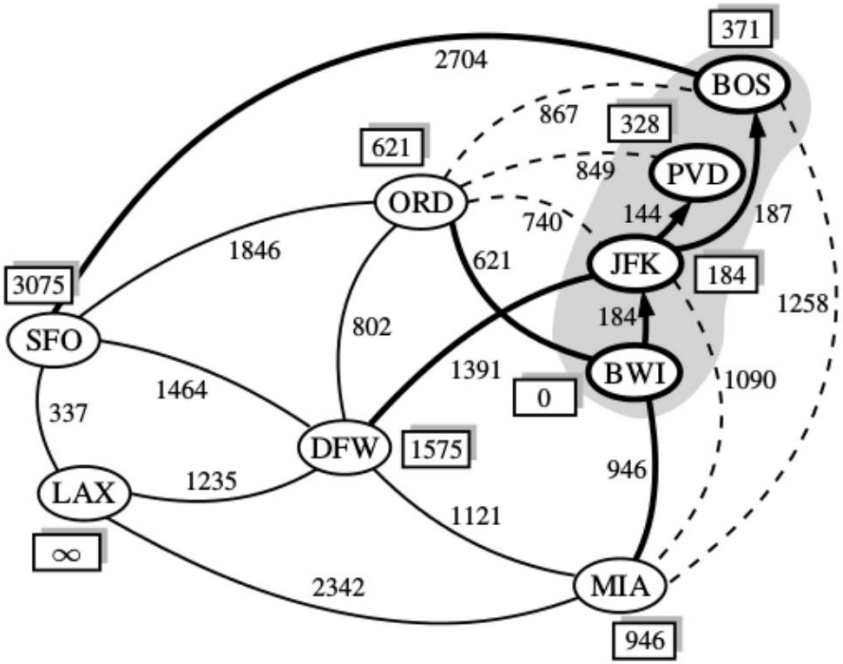
Algoritmo di Dijkstra

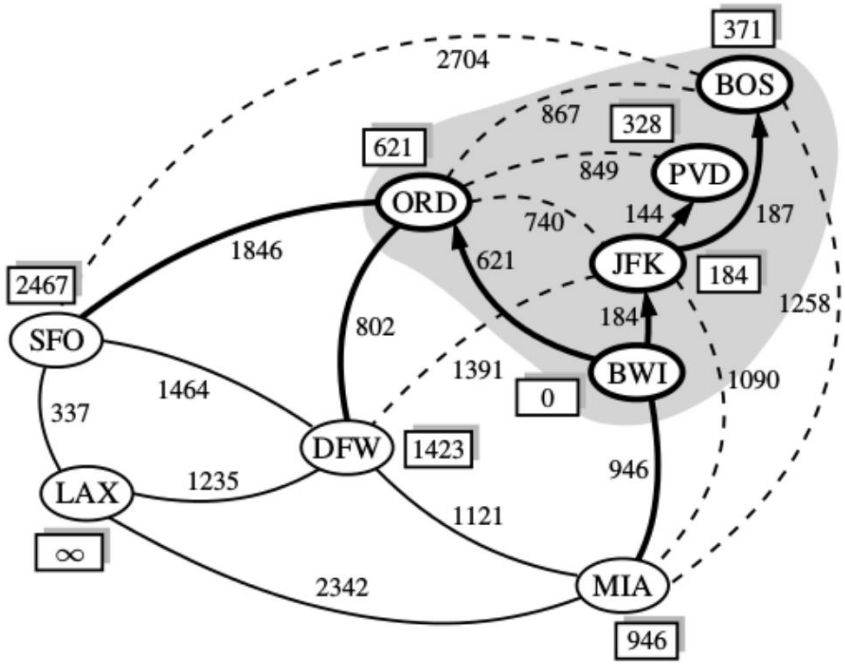




Percorsi più brevi

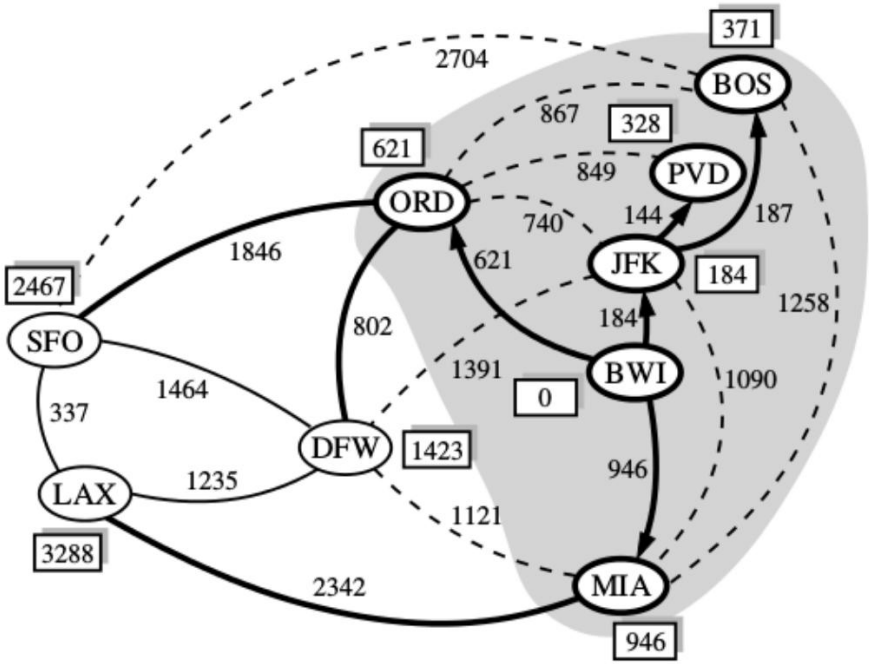
Algoritmo di Dijkstra

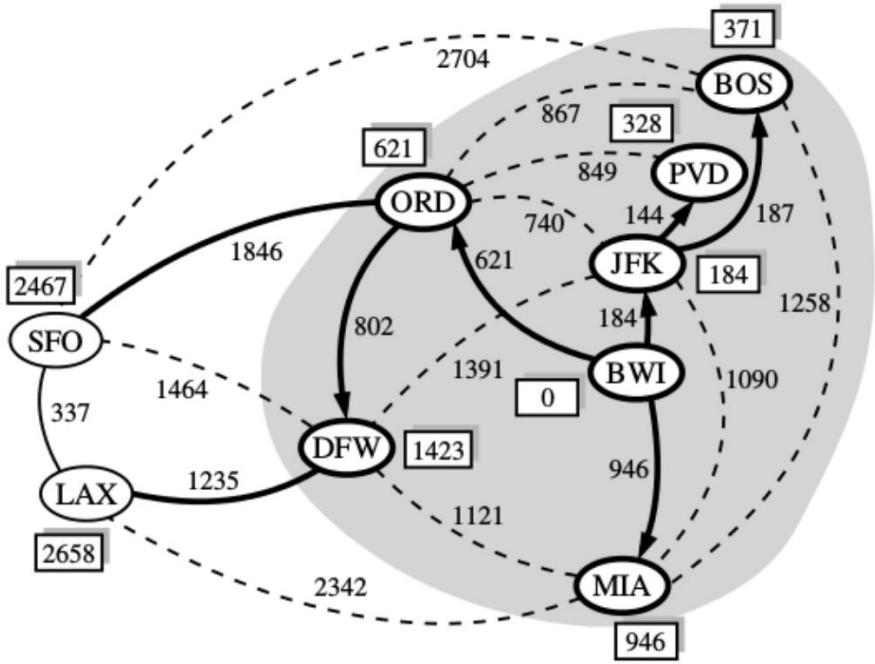




Percorsi più brevi

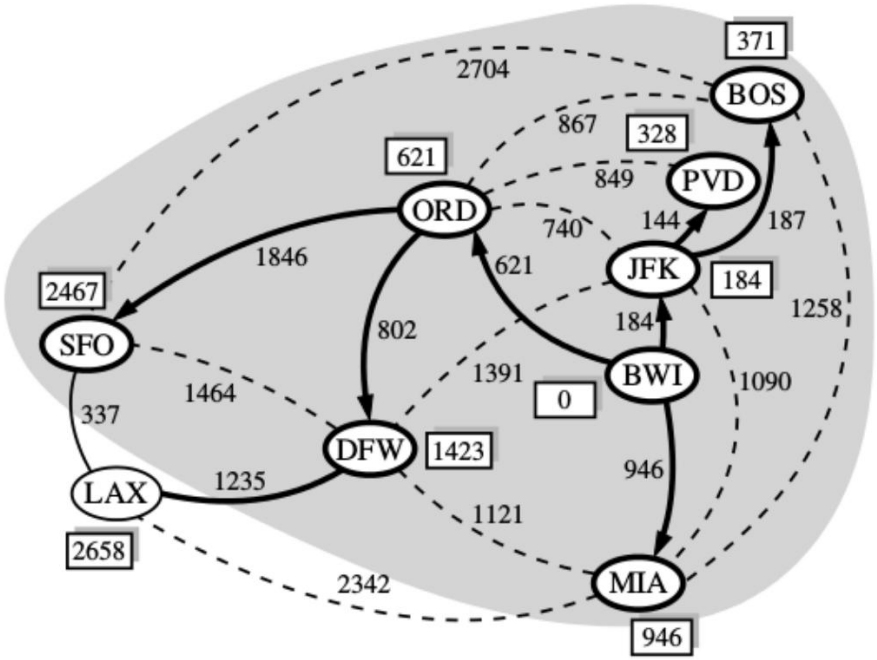
Algoritmo di Dijkstra

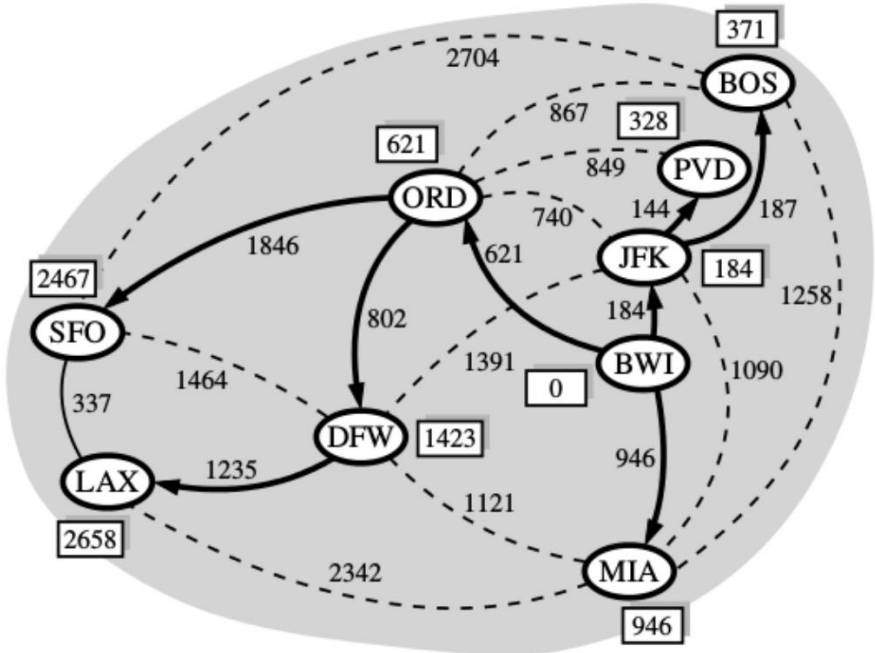




Percorsi più brevi

Algoritmo di Dijkstra





Algoritmo di Dijkstra

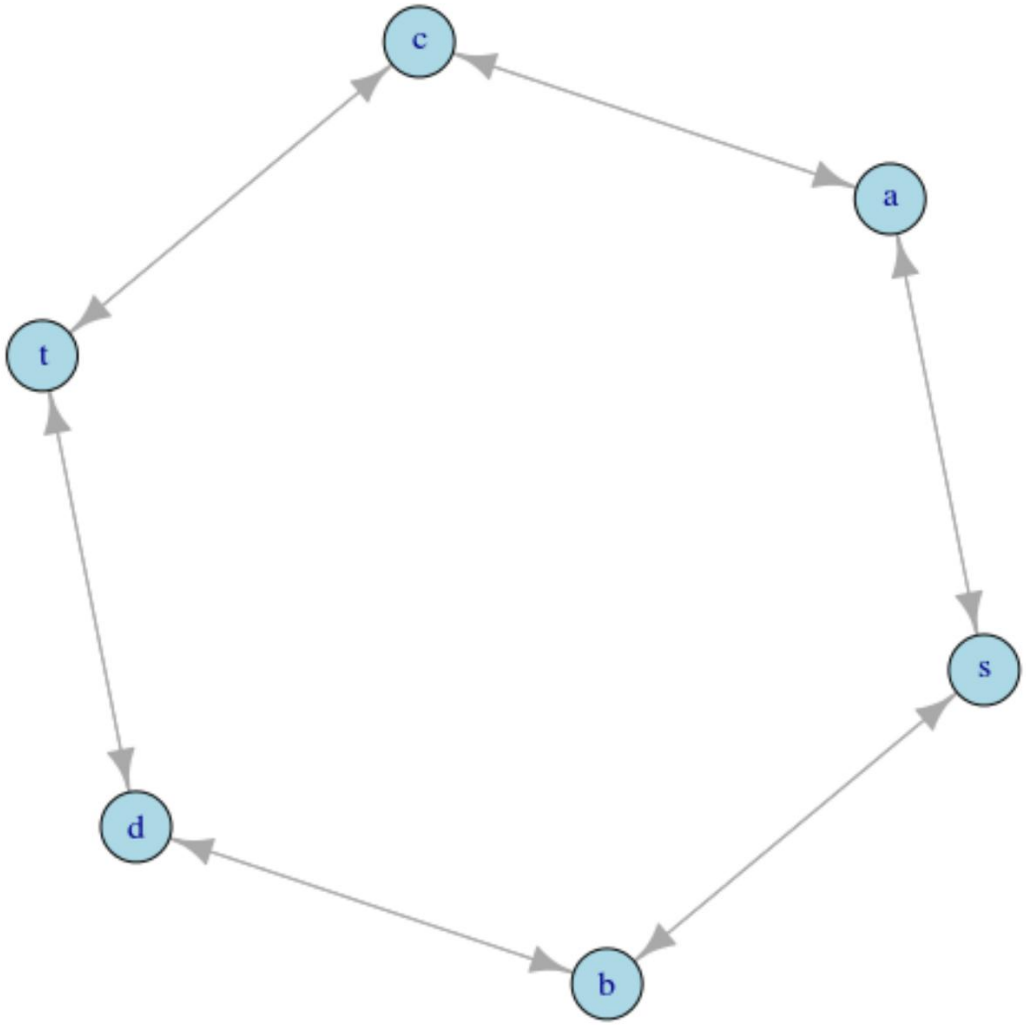
Albero del percorso più breve

```
def albero_percorso_più_corto(g, s, d):

    albero = {}
    per v in d:
        se v non è s:
            for e in g.incident_edges(v, False):
                u = e.opposto(v) wgt =
                e.elemento() if d[v] ==
                d[u] + wgt:
                    albero[v] = e
    restituisce albero
```

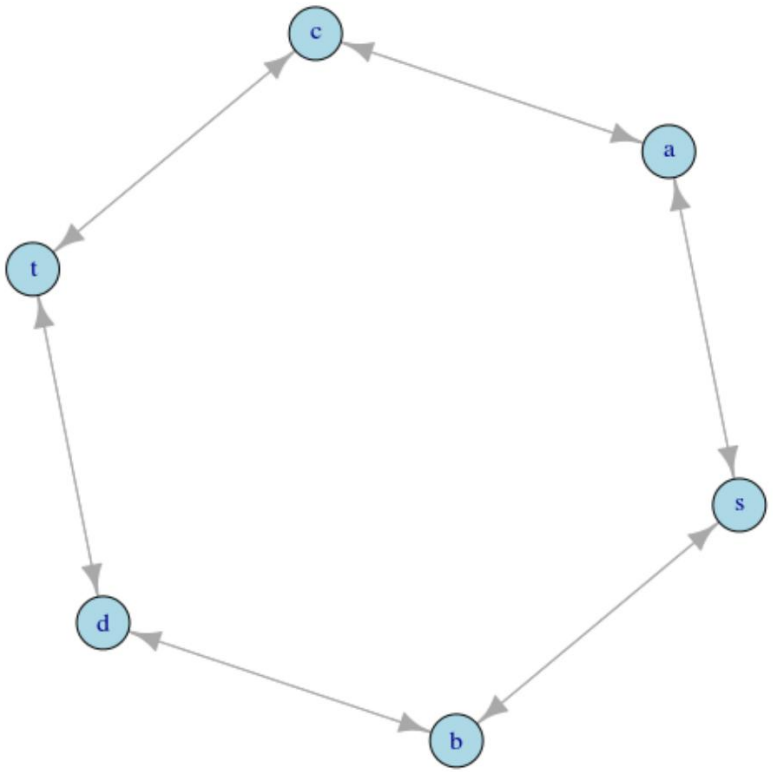
Percorsi più brevi

Algoritmo di Dijkstra



Percorsi più brevi

Algoritmo di Dijkstra




```
grafico = {'s': {'a': 2, 'b': 1}, 'a': {'s': 3, 'b': 4,
                 'c' :8}, 'b': { 's': 4, 'a': 2, 'd': 2}, 'c':
                      {'a':
                 2, 'd': 7, 't': 4}, 'd': {'b ': 1, 'c': 11, 't':
                      5},
                 't': {'c': 3, 'd': 5}}
```

Percorsi più brevi

Algoritmo di Dijkstra

```
def dijkstra(grafico,origine,destinazione,visitato=[],distanze={},
        predecessori={}): se src
         non è nel grafico:
                raise TypeError('la radice del percorso più breve
                       impossibile trovare l'albero nel grafico')
         se dest non è nel grafico:
                raise TypeError('impossibile trovare la destinazione del percorso più
                         breve nel grafico')
         se src == dest:
                path=[]
                pred=dest
                while pred != None:
                       path.append(pred)
                       pred=predecessors.get(pred,None)
                print('percorso più breve: '+str(percorso)+" cost="+str( distanze
                       [destinazione]))
         altro :
                se non visitato:
                       distanze[src]=0
```

```
. . .

per il vicino nel grafico[src]:
    se il vicino non è visitato:
    nuova_distanza = distanze[src] + grafico[src][vicino] se nuova_distanza <
    distanze.get(vicino,float('inf'))
            :
        distanze[vicino] = nuova_distanza
        predecessori[vicino] = src

visitato.append(src) non
visitato={}

per k nel grafico: se
    k non è visitato:
        unvisited[k] = distances.get(k,float('inf')) x=min(unvisited,
key=unvisited.get)
dijkstra(graph,x,dest,visited,distances,predecessors)
```

Alberi di copertura minimi

- Tutti i computer di un ufficio devono essere collegati utilizzando il minor numero di cavi (non è ammesso il wireless)
- Questo problema può essere modellato utilizzando un grafo pesato non orientato G i cui vertici rappresentano i computer e i cui bordi rappresentano tutte le possibili coppie (u, v) di computer
- Il peso w(u, v) del bordo (u, v) è uguale alla quantità di cavo necessaria per collegare il computer u al computer v
- Il problema è trovare un albero che contenga tutti i vertici di G e ha il peso totale minimo su tutti questi alberi

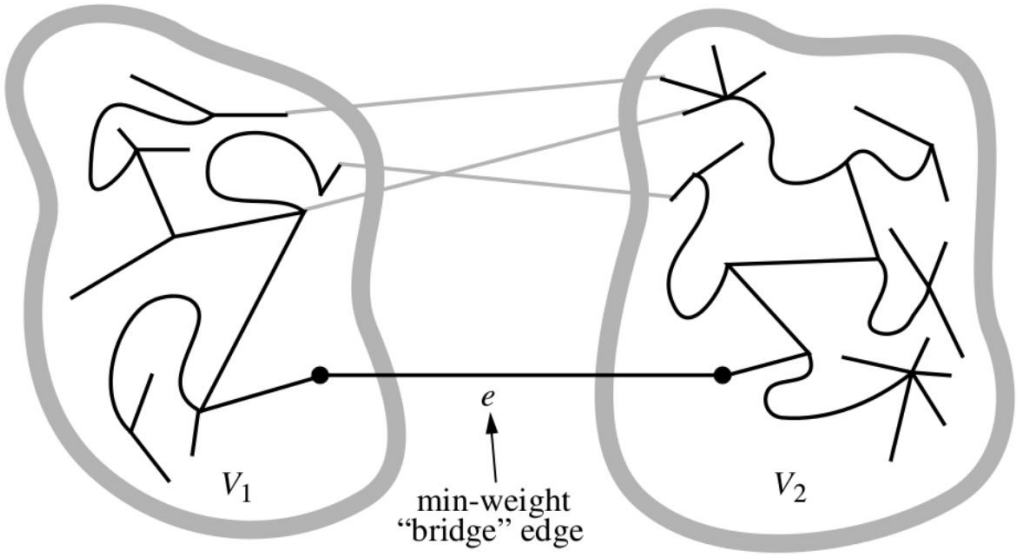
Alberi di copertura minimi

Dato un grafo pesato non orientato G, siamo interessati a trovare un albero T che contenga tutti i vertici in G e minimizzi la somma

$$w(T) = \sum_{(u,v) \text{ in } T} w(u,v)$$

Un albero di questo tipo che contiene ogni vertice di un grafo connesso G è detto albero di copertura e il problema di calcolare un albero di copertura T con il peso totale più piccolo è noto come problema dell'albero di copertura minimo (MST).

Alberi di copertura minimi



Proposizione

Sia G un grafo connesso pesato, e siano V1 e V2 una partizione dei vertici di G in due insiemi disgiunti non vuoti. Sia e un arco in G di peso minimo tra quelli con un estremo in V1 e l'altro in V2. Esiste un albero di copertura minimo T che ha e come uno dei suoi bordi.

Alberi di copertura minimi

Algoritmo di Prim-Jarnik

- In questo algoritmo, uno spanning tree minimo viene cresciuto a partire da un singolo cluster a partire da alcuni vertici “radice”.
- L'idea principale è simile a quella dell'algoritmo di Djkstra: viene definita una nuvola di vertici che cresce ad ogni iterazione
- Ad ogni iterazione viene scelto un arco di peso minimo e = (u, v) che collega un vertice u nella nuvola C ad un vertice v esterno alla nuvola C
- Il vertice v viene portato nella nuvola C e il processo viene ripetuto finché non si forma uno spanning tree
- Come nell'algoritmo di Dijkstra'a, viene mantenuta un'etichetta D[v] per ogni vertice esterno alla nuvola C, in modo che D[v] memorizzi il peso dello spigolo minimo osservato per unire v alla nuvola C

Alberi di copertura minimi

Algoritmo di Prim-Jarnik

Algoritmo Prim_Jarnik(G):
Input: un grafo non orientato, pesato e connesso G
con n vertici e m spigoli
Risultato: un albero di copertura minimo T per G

Scegli qualsiasi vertice s di GD[s]
= 0 per ogni
vertice v != s da fare
D[v] = infinito
Inizializza T = 0
Inizializza una coda con priorità Q con una voce (D[v], (v,
Nessuno)) per ogni vertice v, dove D[v] è la chiave
nella coda di priorità e (v, None) è il valore associato

mentre Q non è vuoto do (u, e) =
valore restituito da Q.remove_min()
Connetti il vertice u a T usando lo spigolo e per
ogni spigolo e' = (u,v) tale che v sia in Q do se w(u,v) < D[v] do

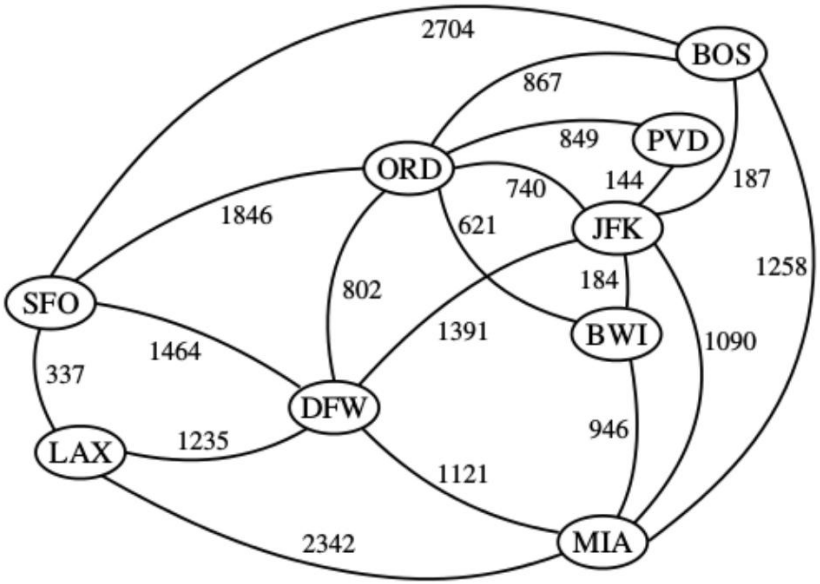
Alberi di copertura minimi

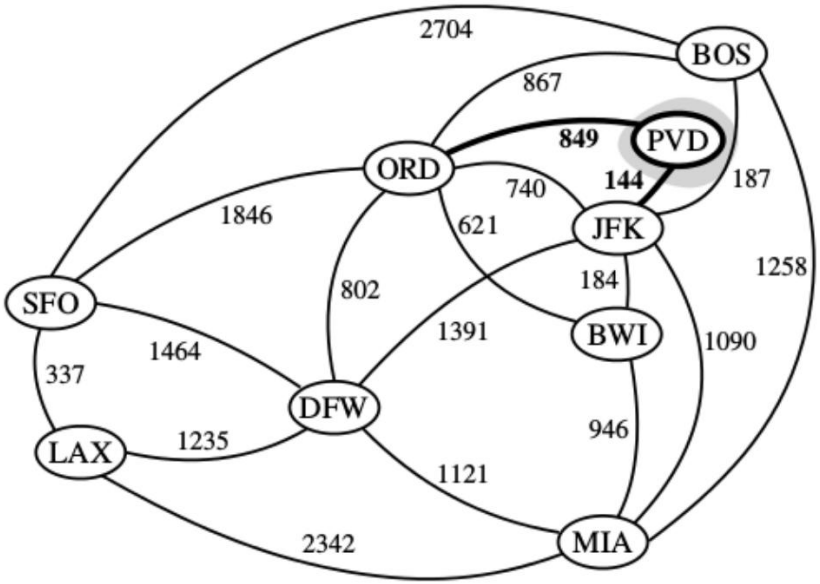
Analisi dell'algoritmo di Prim-Jarnik

- Vengono eseguiti n inserimenti in Q
- Successivamente vengono eseguiti n estratti-min
- Un totale di m priorità sono aggiornamenti
- Con una coda con priorità, ogni operazione viene eseguita in O(logn) e il tempo complessivo per gli algoritmi è O((n + m)logn), ovvero O(mlogn) per un grafo connesso
- Utilizzando un elenco non ordinato, il tempo di esecuzione sarà $O(n^2)$

Alberi di copertura minimi

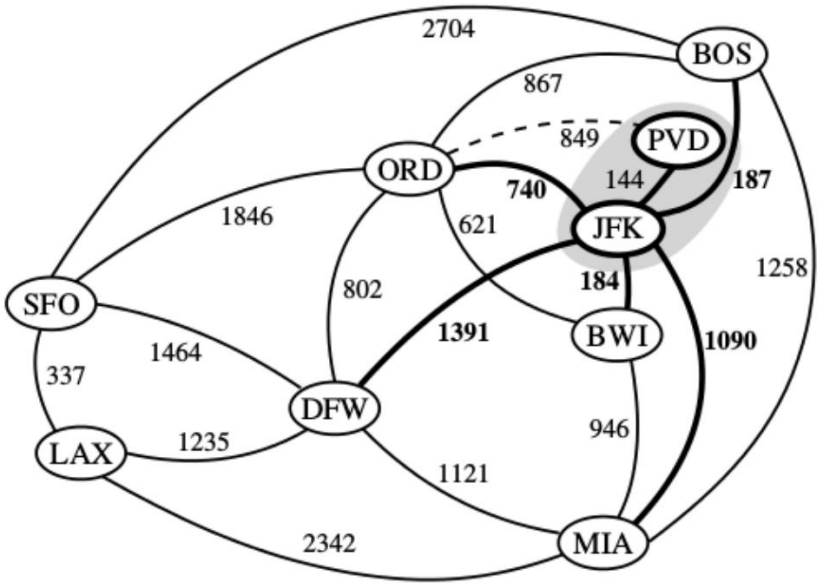
Algoritmo di Prim-Jarnik

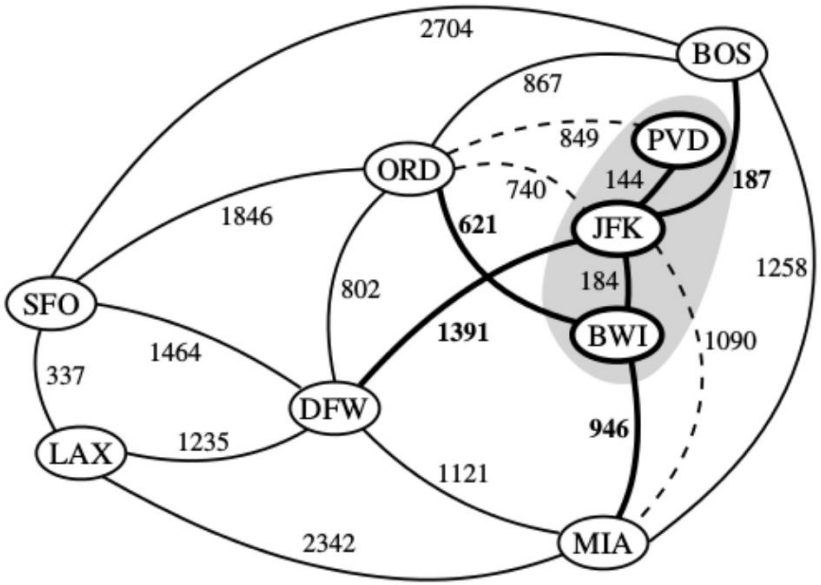




Alberi di copertura minimi

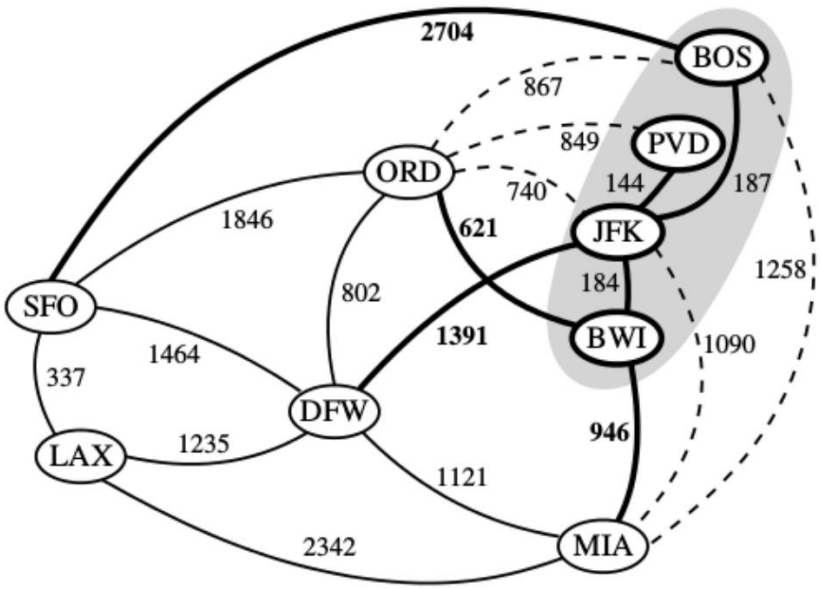
Algoritmo di Prim-Jarnik

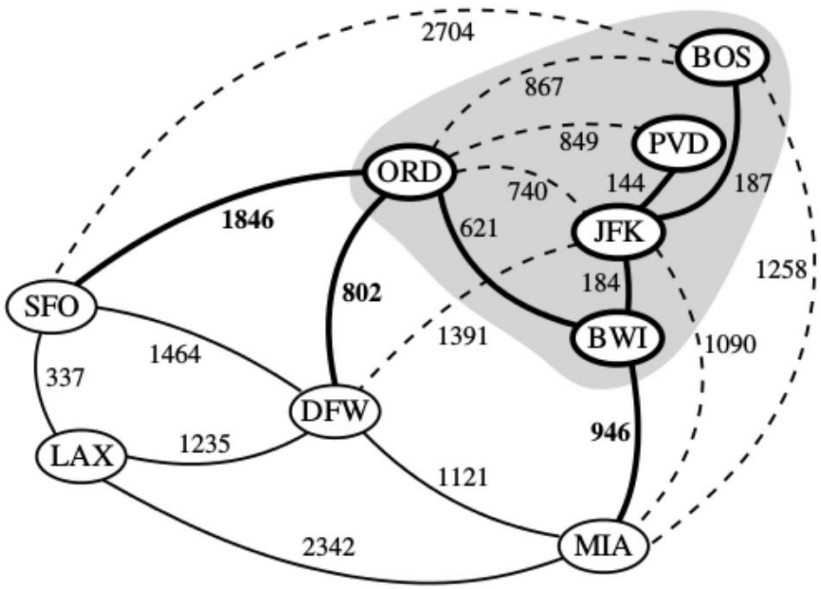




Alberi di copertura minimi

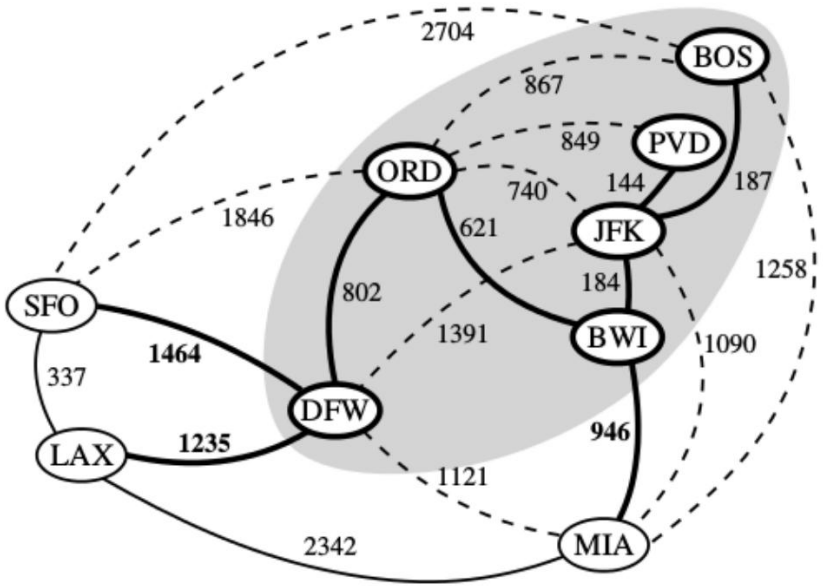
Algoritmo di Prim-Jarnik

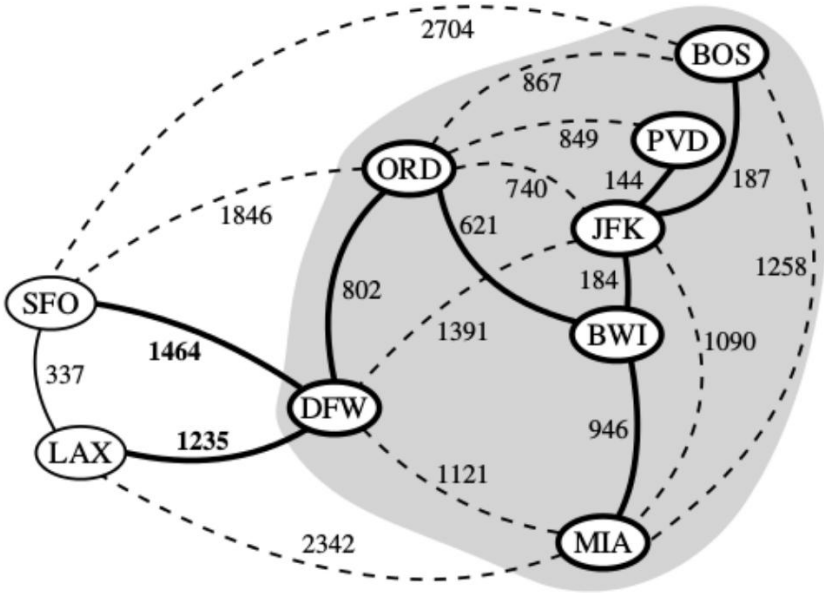




Alberi di copertura minimi

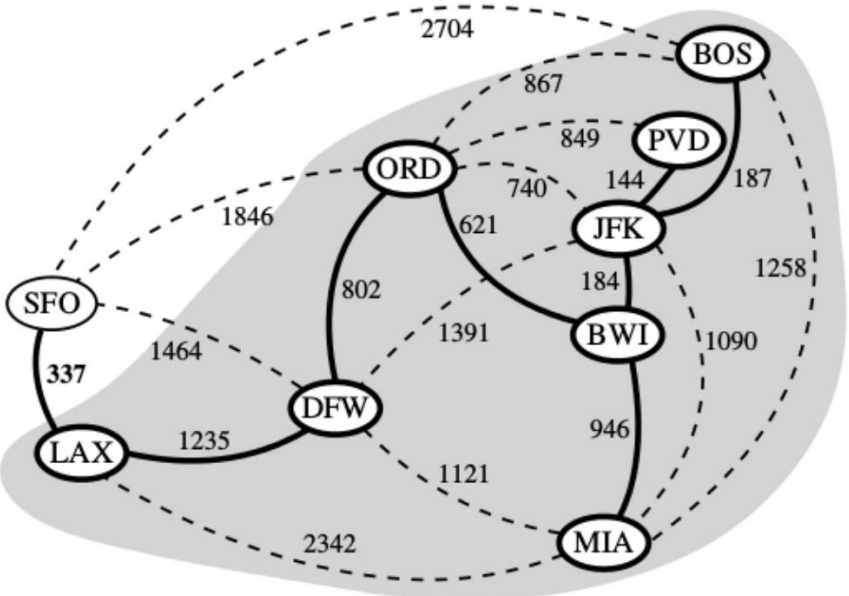
Algoritmo di Prim-Jarnik

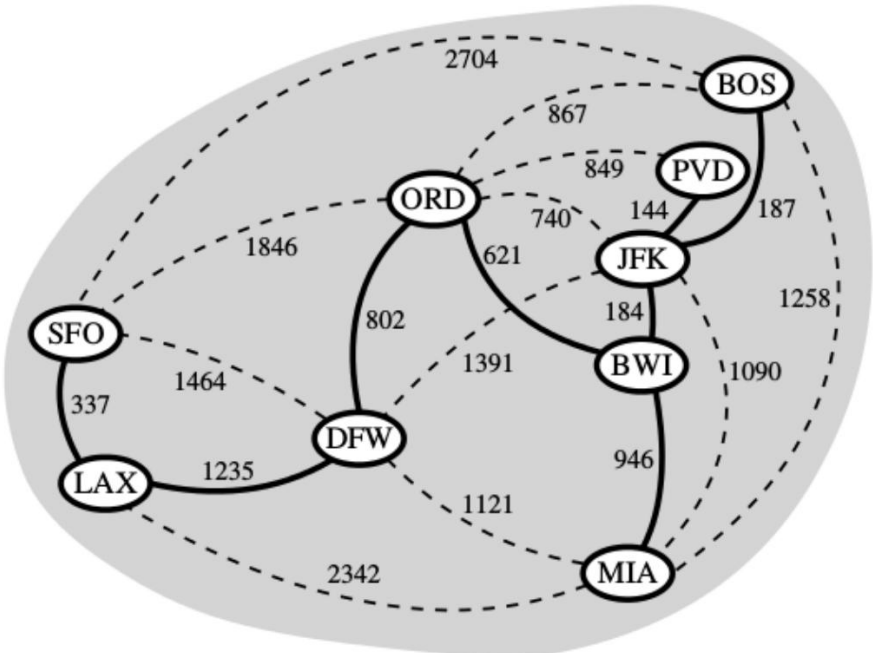




Alberi di copertura minimi

Algoritmo di Prim-Jarnik





Alberi di copertura minimi

Algoritmo di Prim-Jarnik

```
def MST_PrimJarnik(g): d = {}
    albero =
    [] pq =
    AdaptableHeapPriorityQueue() pqlocator = {}


    for v in g.vertici(): if len(d) == 0:
        d[v] = 0

        altro:
            d[v] = float('inf')
        pqlocator[v] = pq.add(d[v], (v,None))

     .   .   .

(continua)
```

Alberi di copertura minimi

Algoritmo di Prim-Jarnik

```
. . .

    mentre non pq.is_empty():
        key,value = pq.remove_min() u,edge
        = value del
        pqlocator[u] se edge
        non è None: tree.append(edge)

        per il collegamento in g.incident_edges(u):
            v = link.opposite(u) se v in
            pqlocator:
                wgt = link.element() if wgt <
                d[v]: d[v] = wgt

                    pq.update(pqlocator[v], d[v], (v, link))

    albero di ritorno
```

Alberi di copertura minimi

Algoritmo di Kruskal

- L'algoritmo di Kruskal mantiene una foresta di cluster, unendo ripetutamente coppie di cluster finché un singolo cluster non occupa il grafico
- Inizialmente ogni vertice è di per sé un cluster singleton
- L'algoritmo considera ciascun bordo a turno, ordinato per peso crescente
- Se un arco e collega due cluster diversi, allora e viene aggiunto all'insieme degli archi del MST e i cluster collegati da e vengono fusi in un unico cluster
- Al contrario, se e collega due vertici che sono già nello stesso cluster, allora e viene scartato
- L'algoritmo termina quando sono stati aggiunti abbastanza archi per formare uno spanning tree

Alberi di copertura minimi

Algoritmo di Kruskal

```
Algoritmo Kruskal(G):
    Input: un semplice grafo pesato connesso G con n vertici e
                m spigoli
    Risultato: un albero di copertura minimo T per G

    per ogni vertice v in G do
        Definisci un cluster elementare C(v) = {v}
        Inizializza una coda con priorità Q per contenere tutti gli archi
            in Sol, usando i pesi come chiavi
        T = 0
        mentre T ha meno di n-1 archi
            (u,v) = valore restituito da Q.remove_min()
            Sia C(u) il cluster contenente u, e sia C(v) il cluster contenente v se C(u) !
                = C(v) allora

                Aggiungi il bordo (u,v) a T
                Unisci C(u) e C(v) in un unico cluster
        albero di ritorno T
```

Alberi di copertura minimi

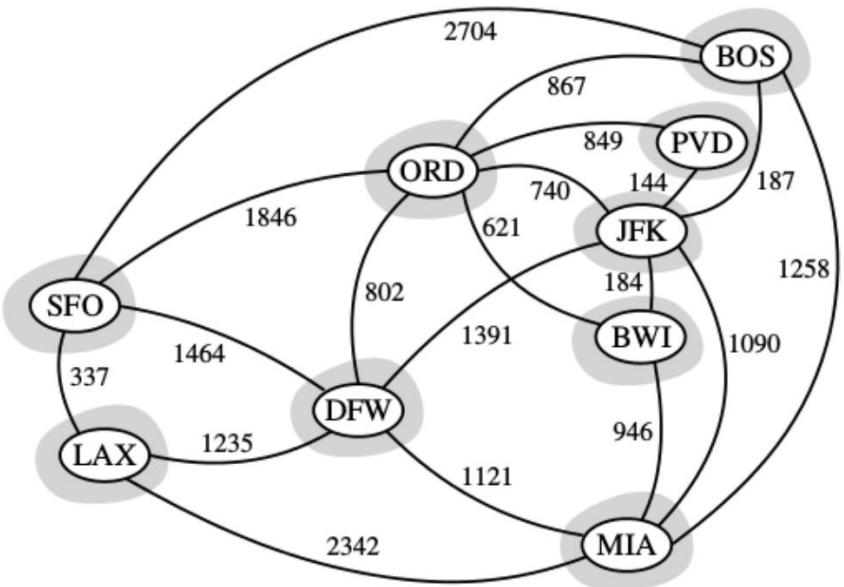
Algoritmo di Kruskal

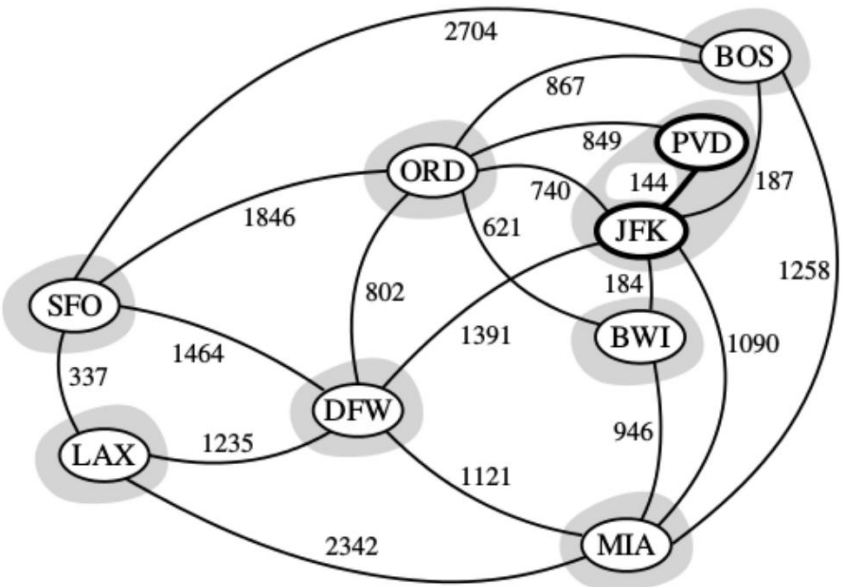
- La correttezza dell'algoritmo di Kruskal si basa sul fatto cruciale relativo agli alberi di copertura minimi della Proposizione
- Ogni volta che l'algoritmo di Kruskal aggiunge un arco (u, v) al MST T, possiamo definire un partizionamento dell'insieme dei vertici V lasciando che V1 sia il cluster contenente v e lasciando che V2 contenga il resto dei vertici in V
- Ciò definisce un partizionamento disgiunto dei vertici di V Inoltre,
- poiché stiamo estraendo gli archi da Q in ordine in base ai loro pesi, e deve essere un arco di peso minimo con un vertice in V1 e l'altro in V2.

- Pertanto, l'algoritmo di Kruskal aggiunge sempre un vantaggio MST valido

Alberi di copertura minimi

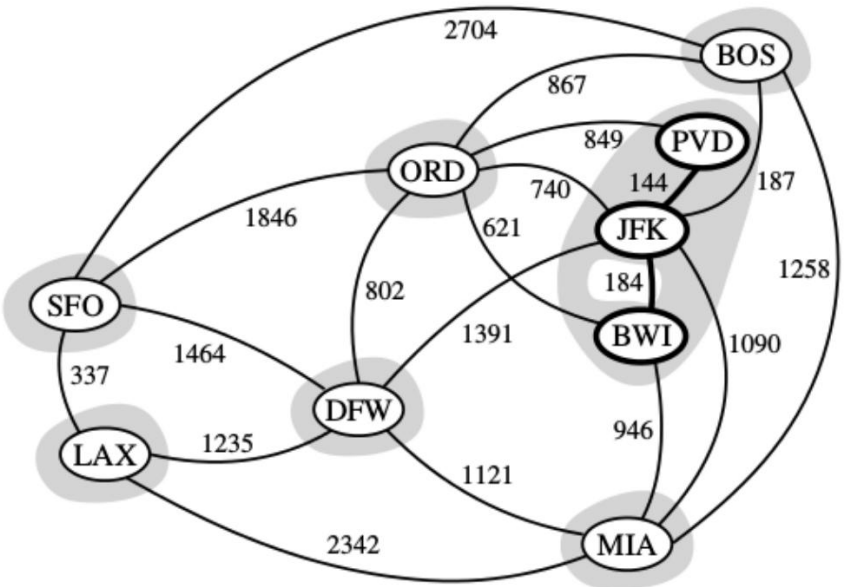
Algoritmo di Kruskal

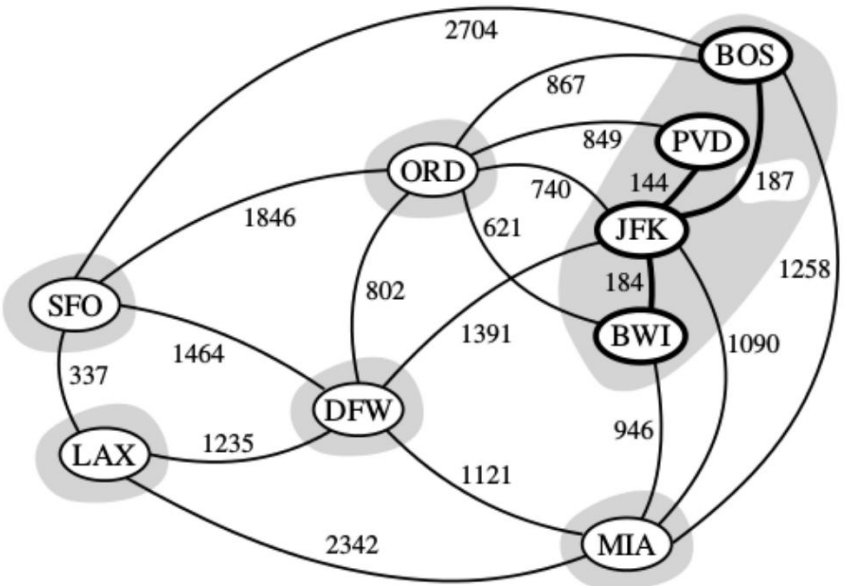




Alberi di copertura minimi

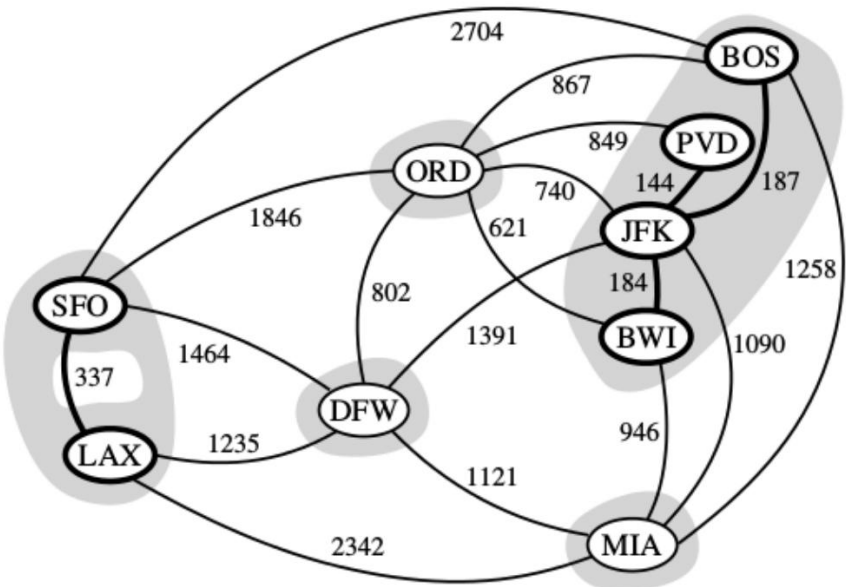
Algoritmo di Kruskal

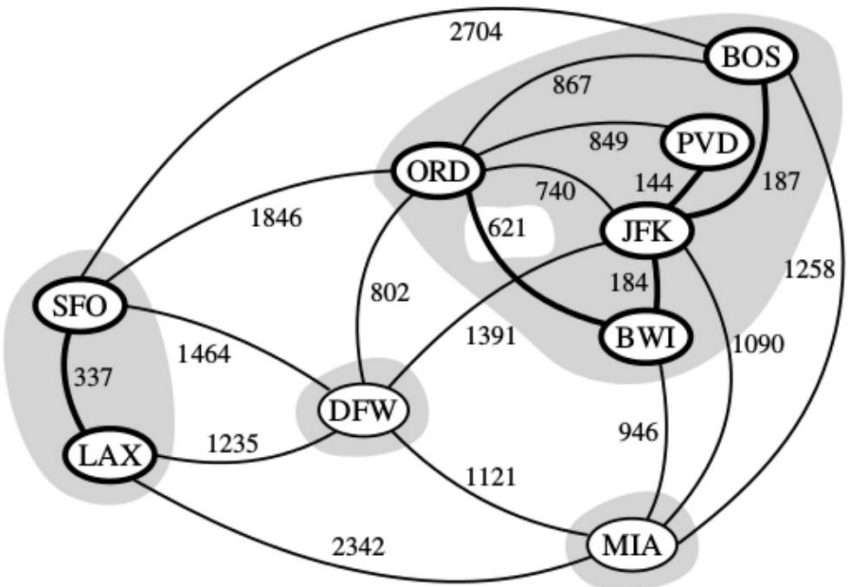




Alberi di copertura minimi

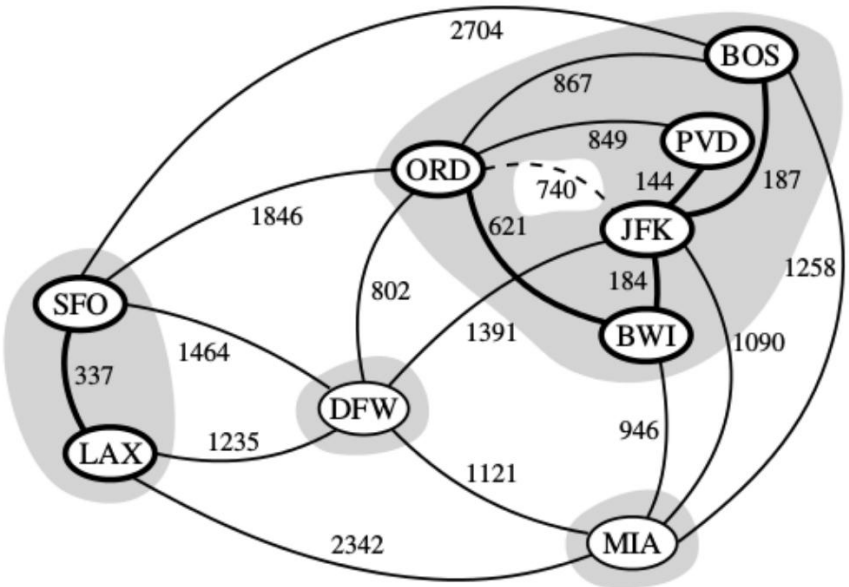
Algoritmo di Kruskal

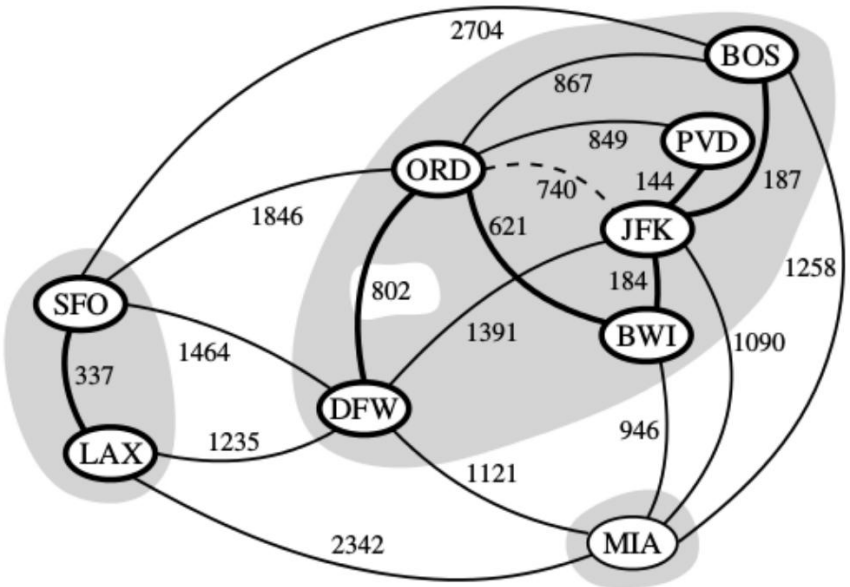




Alberi di copertura minimi

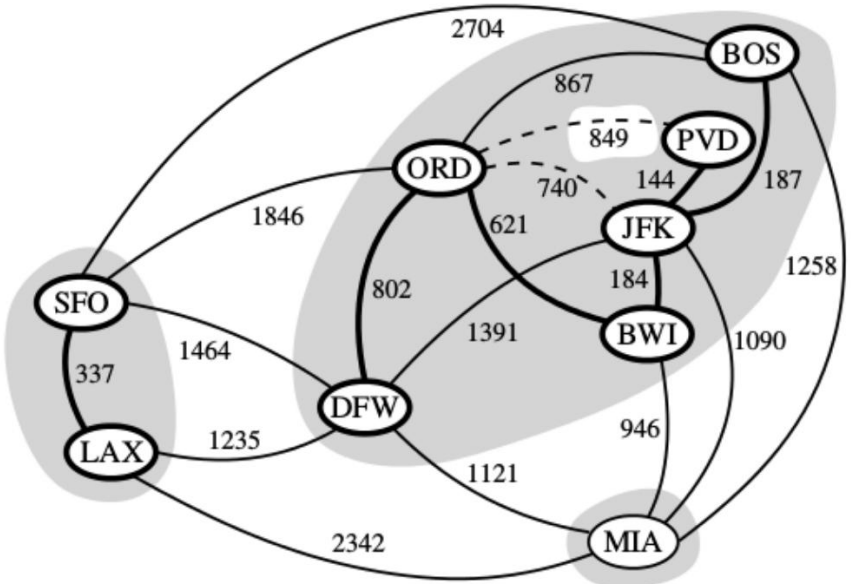
Algoritmo di Kruskal

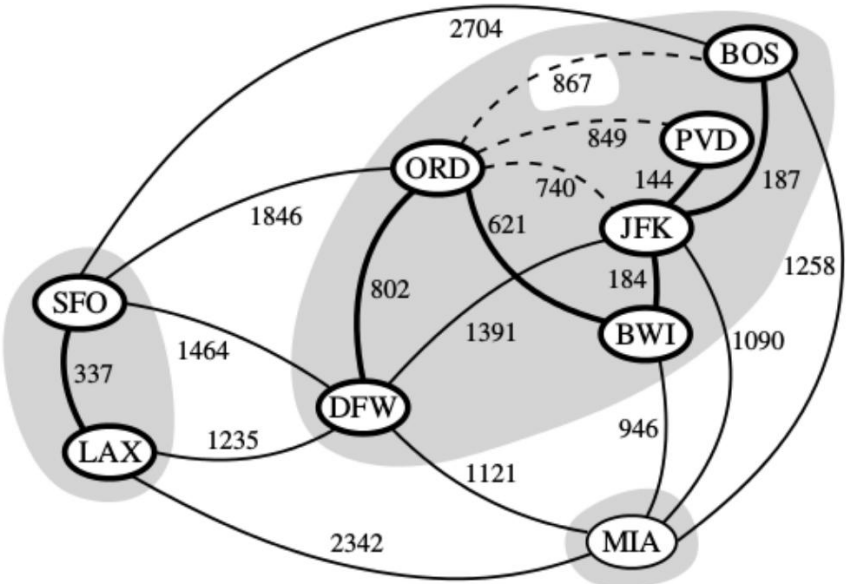




Alberi di copertura minimi

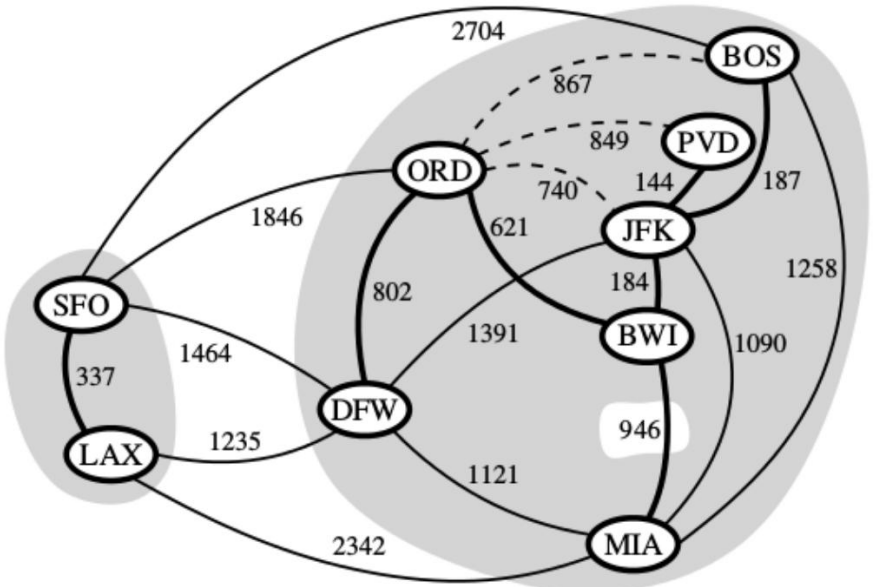
Algoritmo di Kruskal

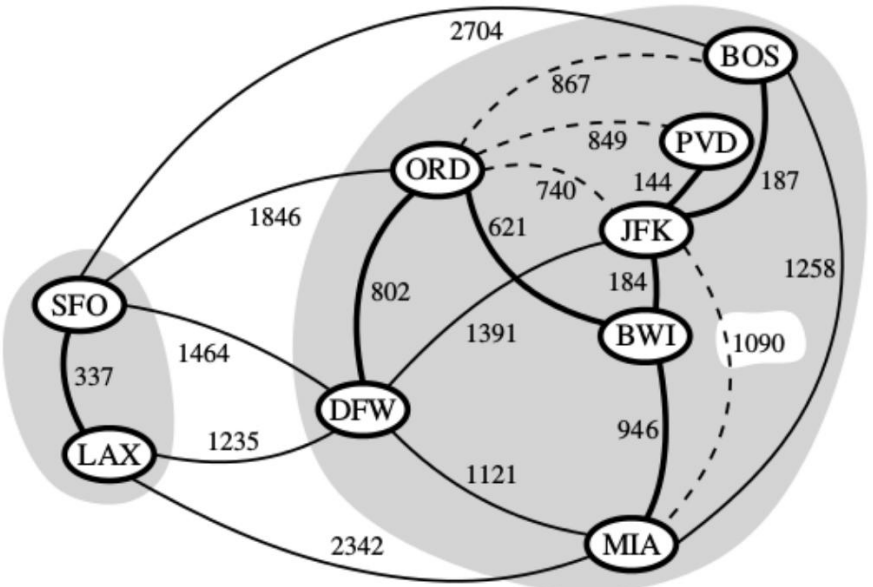




Alberi di copertura minimi

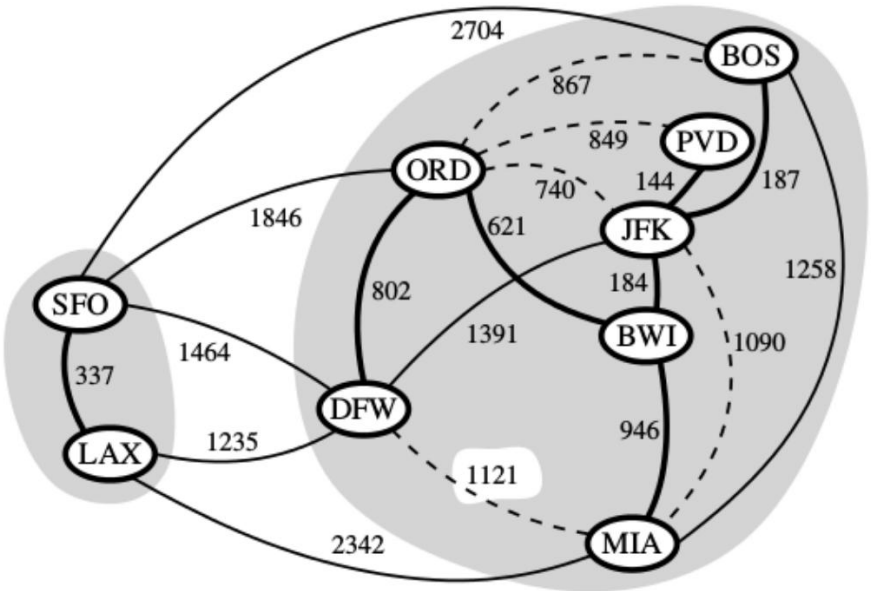
Algoritmo di Kruskal

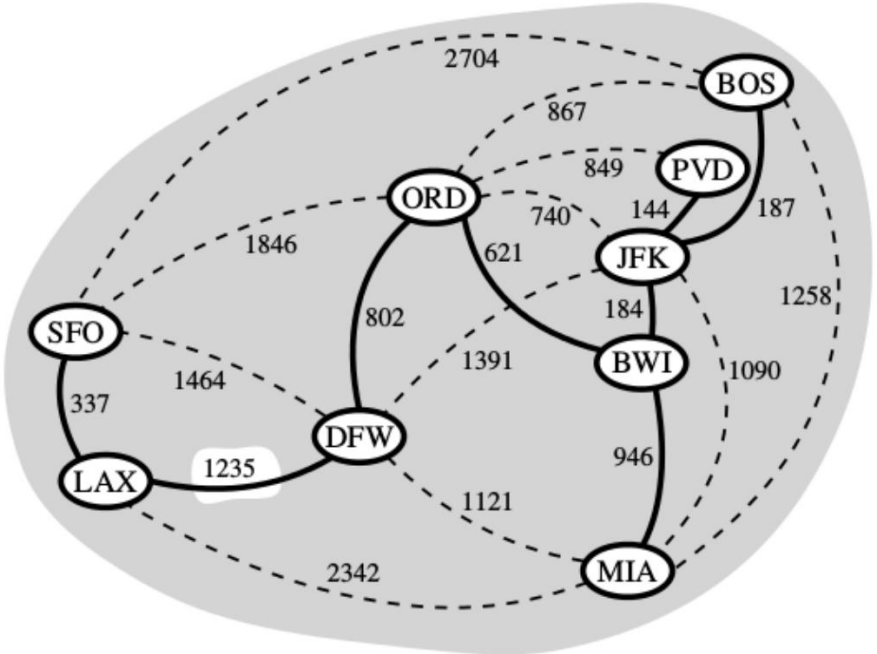




Alberi di copertura minimi

Algoritmo di Kruskal





Alberi di copertura minimi

Analisi dell'algoritmo di Kruskal

- L'ordinamento degli archi può essere implementato in O(mlogn) (utilizzando una coda prioritaria)
- Per implementare l'algoritmo di Kruskal, dobbiamo essere in grado di trovare il cluster per i vertici u e v che sono gli estremi di un bordo e, per verificare se questi due cluster sono distinti e, in tal caso, per unire questi due cluster in uno solo
- La gestione delle partizioni disgiunte può essere eseguita in O(m + nlogn)
- Per un grafo connesso abbiamo m ÿ n ÿ 1 e quindi il termine dominante è O(mlogn), che è il tempo di esecuzione di Algoritmo di Kruskal

Alberi di copertura minimi

Algoritmo di Kruskal

```
def MST_Kruskal(g): albero
    = [] pq =
    HeapPriorityQueue() foresta =
    Partition() posizione = {}


    for v in g.vertices(): position[v]
        = forest.make_group(v)

    for e in g.edges():
        pq.add(e.element(), e)

    size = g.vertex_count() while
    len(tree) != size - 1 e non pq.is_empty():
        peso,edge = pq.remove_min() u,v =
        edge.endpoints() a =
        forest.find(position[u]) b =
        forest.find(position[v]) if a != b:
```

Alberi di copertura minimi

Partizioni disgiunte

- Una struttura dati di partizione gestisce un universo di elementi organizzati in insiemi disgiunti (un elemento appartiene a uno e solo uno di questi insiemi)
- Per motivi di chiarezza, i cluster di una partizione vengono definiti gruppi
- Per distinguere tra un gruppo e un altro, assumiamo che in qualsiasi momento ogni gruppo abbia una voce designata chiamata leader del gruppo

Alberi di copertura minimi

Partizioni disgiunte

Metodo	Funzionalità
make_group(x)	Crea un gruppo singleton contenente il nuovo elemento x e restituisce la posizione che memorizza x
unione(p, q)	Unisci i gruppi che contengono posizioni p e q
trovare(p)	Restituisce la posizione del leader del gruppo contenente la posizione p