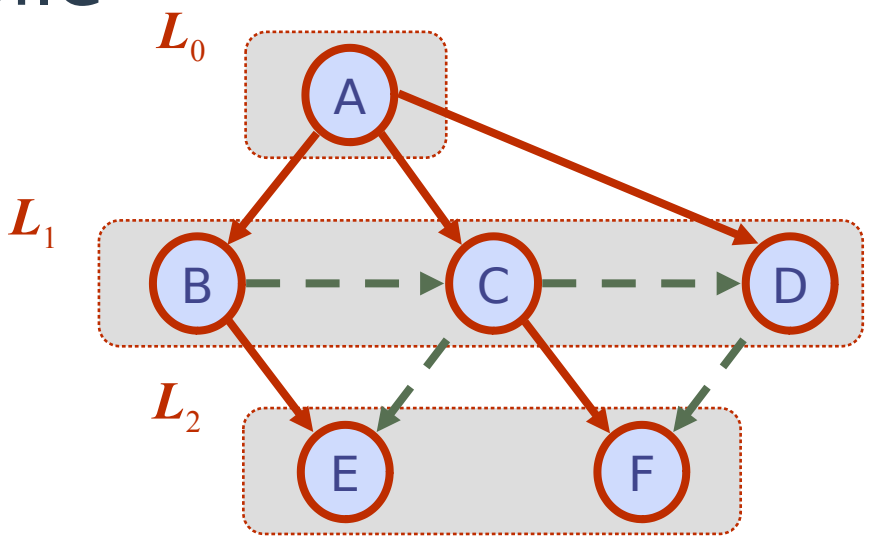
Cammini minimi

Luca Becchetti

Presentazione tratta dalle slide che accompagnano il testo Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

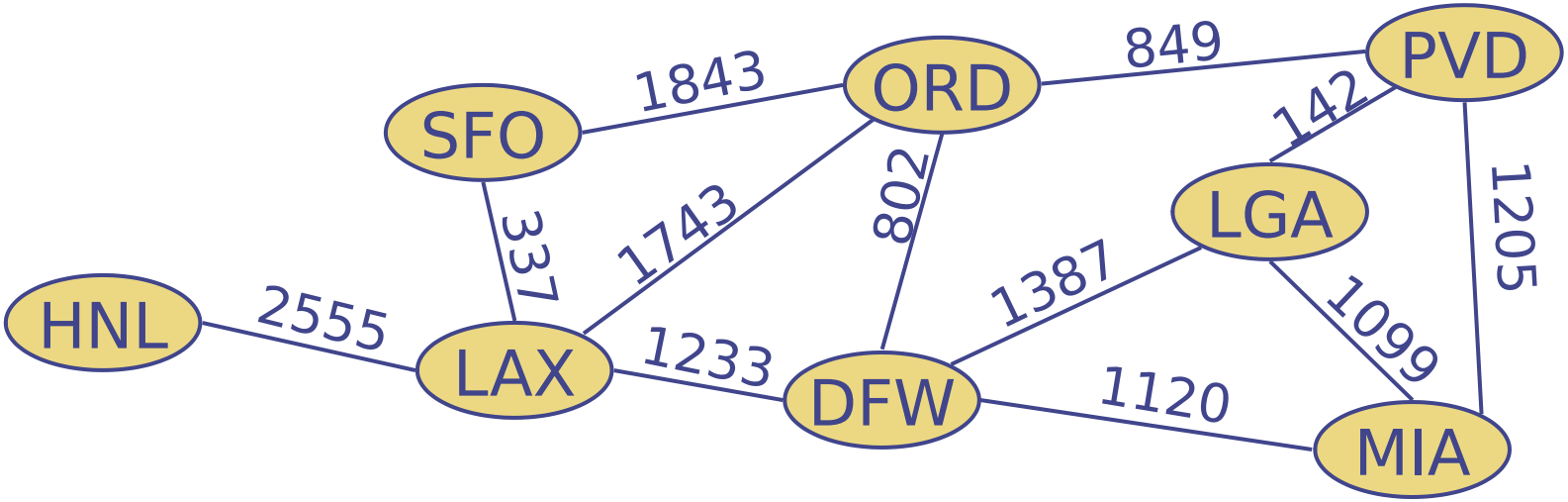
Visita in ampiezza (BFS)

- BFS di un grafo (orientato o meno) a partire da un nodo s
- Identifica cammini con il numero minimo di archi da s a qualsiasi altro vertice raggiungibile



Grafi pesati

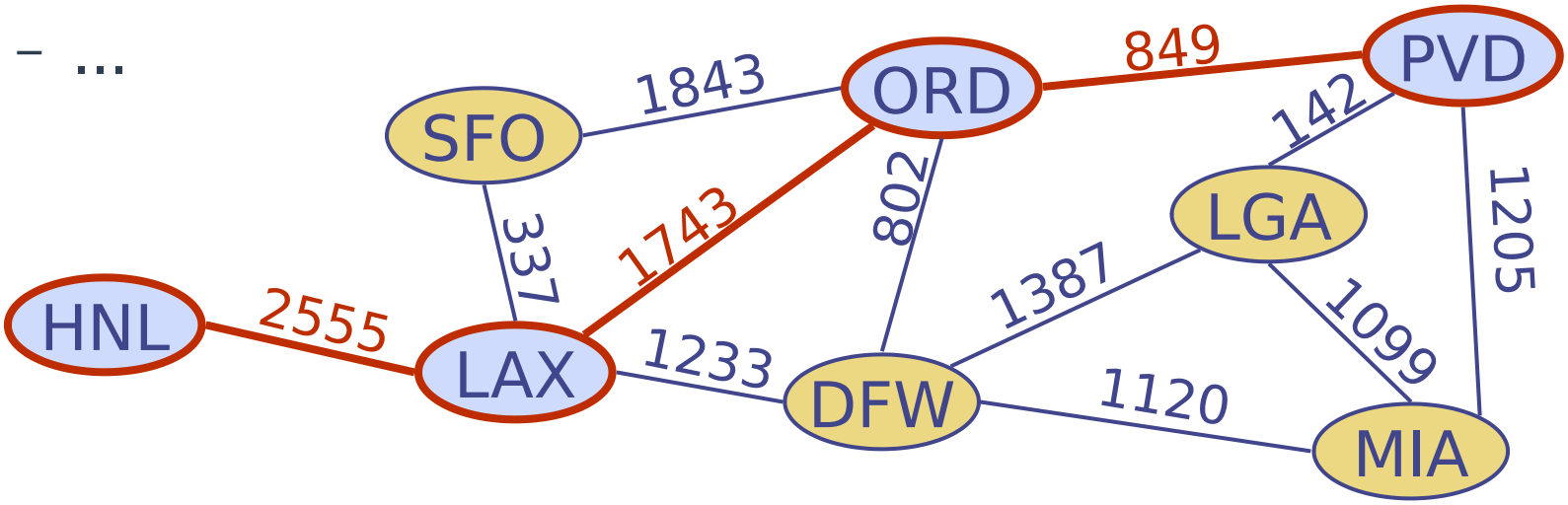
- Ogni arco ha un valore associato (peso)
  - Distanze, costi ecc.
- Esempio



Nel seguito: w(u, v) denota il peso del generico arco (u, v) appartenente a E

Cammino minimo

- Dati *u* e *v*: individuare un cammino di lunghezza complessiva minima da *u* a *v*
- Applicazioni
  - Instradamento di pacchetti IP
  - Navigatori automobilistici
  - …

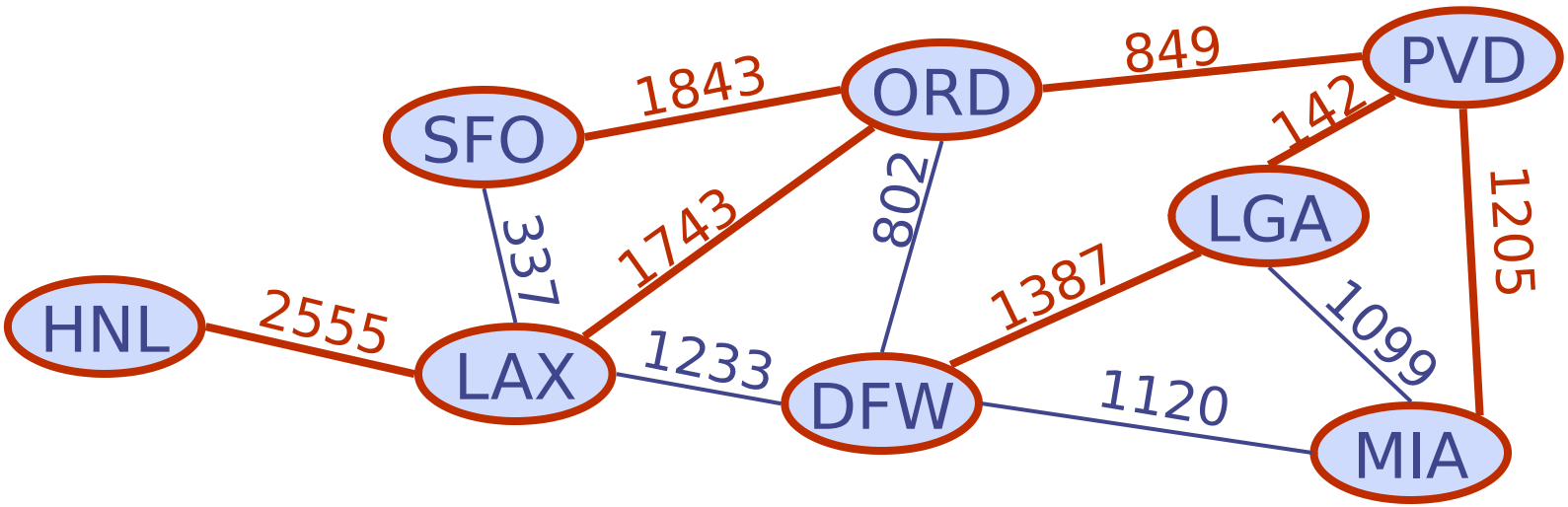


Grafi diretti

- Il cammino da *u* a *v* può non esistere
- Dato un vertice sorgente *s*
  - Sono definiti i cammini minimi da *s* a tutti i vertici *raggiungibili* da *s*
- In questa presentazione
  - Alcune proprietà valgono sempre
  - In altri casi: presentazione per il caso non diretto (diretto) e discussione delle differenze nel caso diretto (non diretto)

Proprietà dei cammini minimi

- *Ogni sottocammino di un cammino minimo è minimo (vera sempre)*
- *Per ogni v: esiste un albero di cammini minimi da v a tutti gli altri vertici (raggiungibili da v, se il grafo è diretto)*



Condizioni di ottimalità

- Valide per grafi diretti e indiretti
- Supponiamo di applicare un algoritmo per il calcolo della distanza minima da un nodo sorgente *s* a tutti gli altri nodi
  - In effetti l'algoritmo in generale calcola un albero di cammini minimi da *s* vertici da esso raggiungibili
- Il risultato è un'etichettatura d[], in cui d[u] è la lunghezza di un cammino da *s* a *u*
- Domanda: vi è una relazione tra i valori d[u] se, per ogni *u*, d[u] è la distanza minima di *u* da *s*?

Condizioni di ottimalità/cont.

Sia d[] un'etichettatura dei vertici, tale che d[u] sia la lunghezza di un percorso da una sorgente s al vertice u.

d[u] è la distanza minima da s per ogni u se e solo se le seguenti condizioni sono soddisfatte:

Per ogni u e per ogni (v, u) appartenente a E:

$$d(u) \leq d(v) + w(v, u)$$

Condizioni di ottimalità - prova

- *Necessità*
  - Si supponga $d(u) > d(v) + w(v, u)$ per qualche (v, u)
  - Possiamo trovare un cammino più breve da s a u → contraddizione
- *Sufficienza*
  - Sia $s = v_1, v_2, \ldots, v_k = u$ un cammino minimo da s a u
  - $d(u) = d(v_k) \leq d(v_{k-1}) + w(v_{k-1}, v_k)$
  - Se iteriamo
    - $d(u) \leq \sum_i w(v_{i-1}, v_i) = OPT_{su}$
    - ma $d(u) \geq OPT_{su}$ per definizione

Algoritmo di Dijkstra

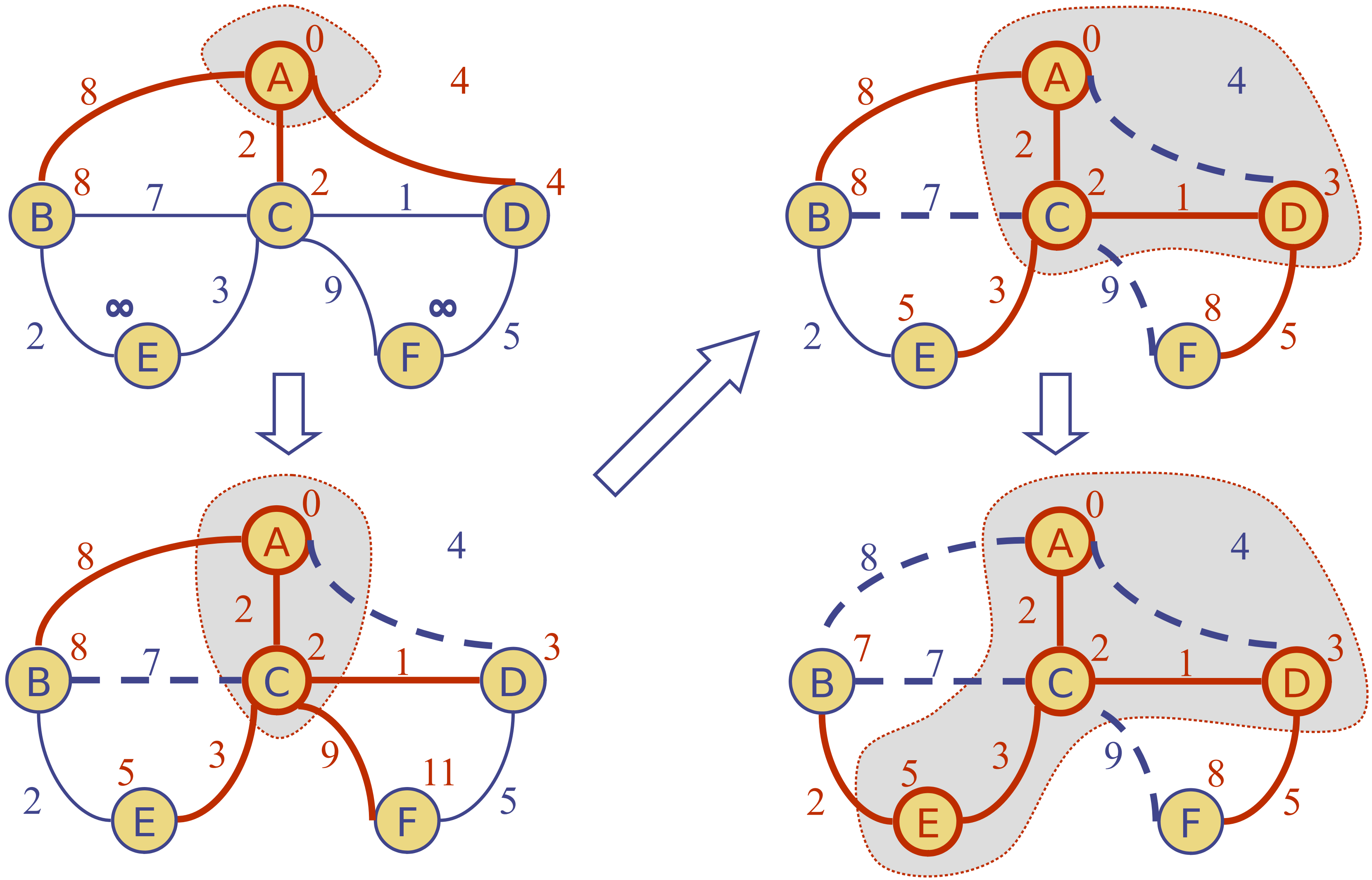
Algoritmo di Dijkstra

- Dato *s*: calcola albero di cammini da s a tutti gli altri vertici

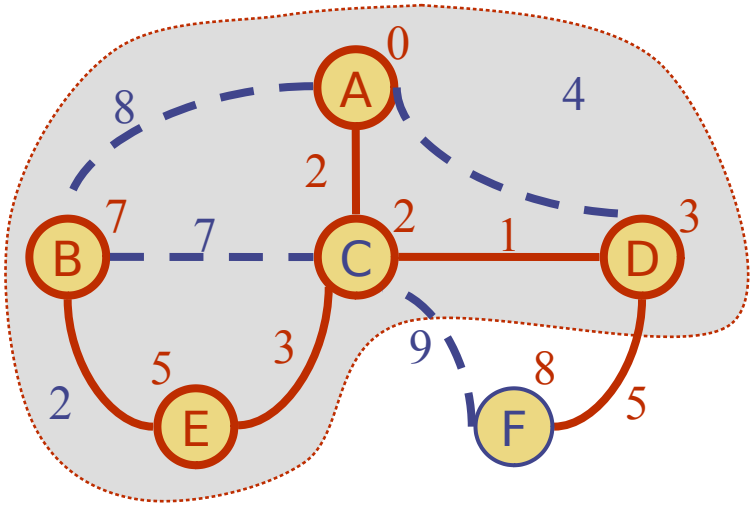
- Si fa crescere un sottoinsieme di S vertici che include *s*
  - All'inizio: S = {*s*}
  - Alla fine: S = V
- Per ogni vertice
  - Etichetta *d(v)*
    - Distanza minima da *s* calcolata finora
  - A ogni passo
    - Aggiungi a S vertice *u* tale che *d(u)* sia minima
    - Aggiorna le etichette dei vertici adiacenti a *u*

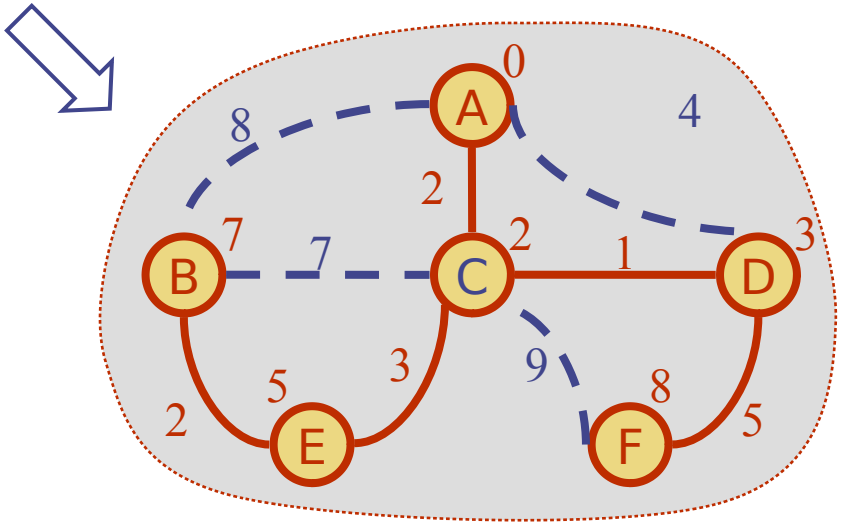
- Assunzioni
  - Grafo connesso
  - Pesi non negativi
  - Grafo non diretto (non necessaria, v. avanti)

Esempio



Esempio (cont.)



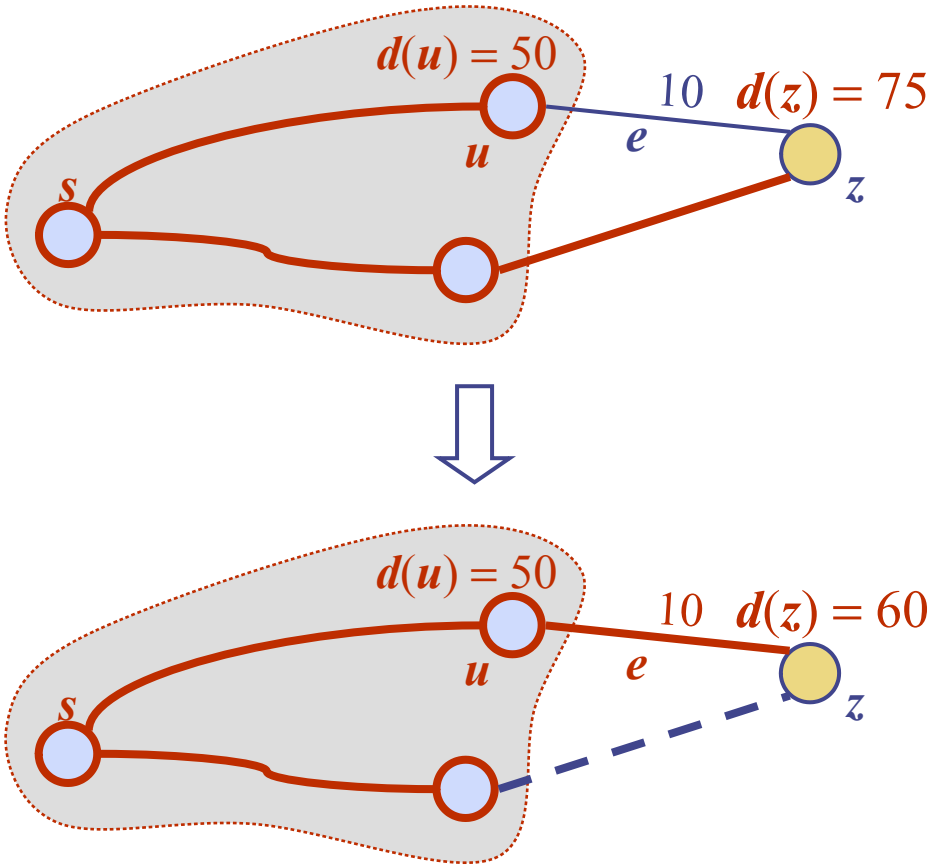


Passo chiave: rilassamento

- Sia $e = (u, z)$ tale che
  - u è stato appena aggiunto a S
  - z non appartiene a S
- Rilassamento

  $d(z) = min\{d(z), d(u) + w(u, z)\}$

$w(u, z)$: peso dell'arco (u, v)



Pseudo-codice

Algorithm ShortestPath(G,s):

Input: A weighted graph G with nonnegative edge weights, and a distinguished vertex s of G.

Output: The length of a shortest path from s to v for each vertex v of G.

Initialize $D[s] = 0$ and $D[v] = \infty$ for each vertex $v \neq s$.

Let a priority queue Q contain all the vertices of G using the D labels as keys.

while Q is not empty **do**

{pull a new vertex u into the cloud}

u = value returned by Q.remove_min()

for each vertex v adjacent to u such that v is in Q **do**

{perform the ***relaxation*** procedure on edge (u,v)}

if $D[u] + w(u,v) < D[v]$ **then**

$D[v] = D[u] + w(u,v)$

Change to $D[v]$ the key of vertex v in Q.

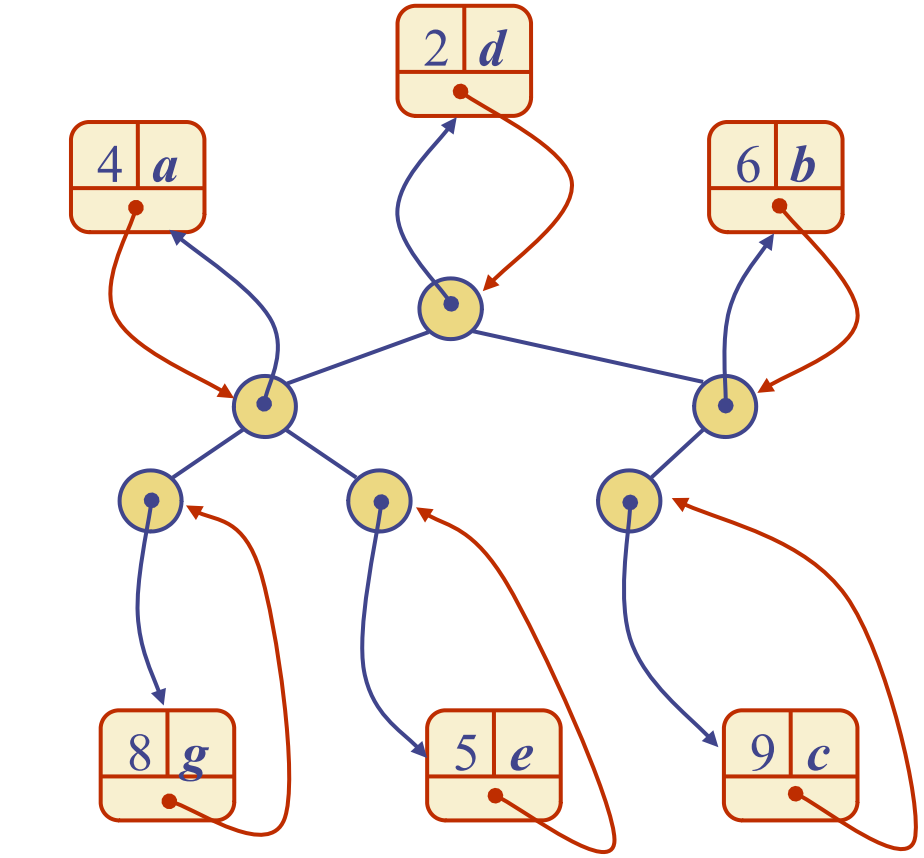
return the label $D[v]$ of each vertex v

Heap flessibile (v. prossima slide)

relax(u)

Coda di priorità (heap) flessibile (Sez. 9.5 del libro)

- Ogni entry contiene tre campi (invece di chiave e valore)
  - Chiave
  - Valore
  - Posizione
- Rappresentazione
  - Albero binario perfettamente bilanciato
  - In pratica: array



Vantaggio: modifica chiave/valore

- replace(e, k):
  - e è un oggetto (entry) dell'heap
  - Sostituisce k al valore corrente della chiave della entry e
  - Ripristina l'heap
- Complessità O(log n)

```
replace(e, k) {
        i = e.position;
        e.key = k;
        if (heap.get(i).key < heap.parent(i).key)
                upheap(i);
        else
                downheap(i); // Potrebbe non essere
                             // necessario
}
```

Analisi: correttezza

1) Condizioni di ottimalità

   $d(u) \leq d(v) + w(v, u)$, per ogni (u, v) in E

2) Siano $s = v_1, \ldots, v_i$ i primi i vertici estratti da Q (nell'ordine di estrazione)

   $d(v_1) \leq d(v_2) \leq \ldots \leq d(v_i)$

3) Dijkstra soddisfa le condizioni di ottimalità

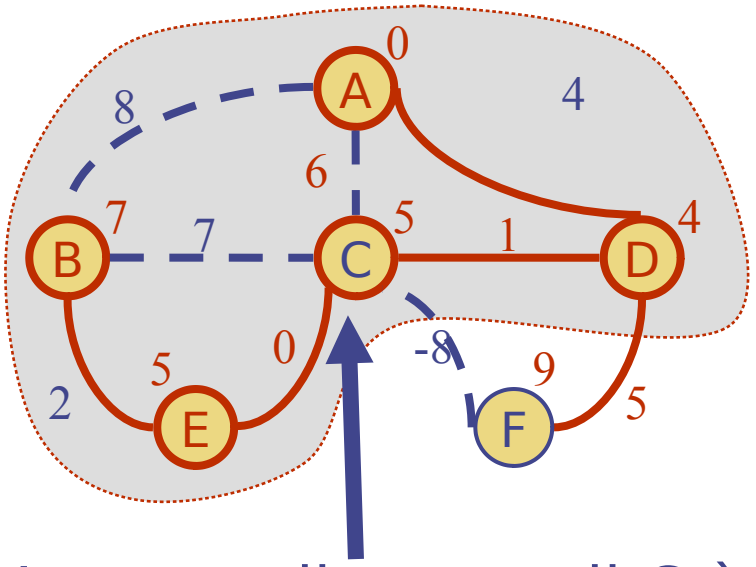
   Prova usa assenza di archi con peso negativo

Analisi: complessità

- Per ogni vertice *v*: si esaminano gli archi incidenti una volta
  - O(m) operazioni in totale
- Etichettatura
  - Per ogni *v*: *d(v)* aggiornata al più *deg(v)* volte
    - Una per ogni inserimento di un vicino di *v* in S nel caso peggiore
  - Ogni aggiornamento richiede O(1) ma
  - Modificare di conseguenza l'heap costa O(log n)
- Complessivamente: costo O((n + m)log n)

Dijkstra in presenza di pesi negativi

- Vertici aggiunti secondo distanze da s non decrescenti
- La presenza di archi con peso negativo potrebbe portare a modifiche delle distanze per vertici già rimossi da Q



La vera distanza di C è 1, ma quando è estratto da Q abbiamo d(C)=5!

Dijkstra: grafi diretti con pesi non negativi

- L’algoritmo e la sua analisi restano invariati
- Ovviamente sono calcolate le distanze dalla sorgente s a tutti e soli i vertici *raggiungibili* da s

Archi con pesi negativi in grafi *diretti*
Algoritmo di Bellman-Ford

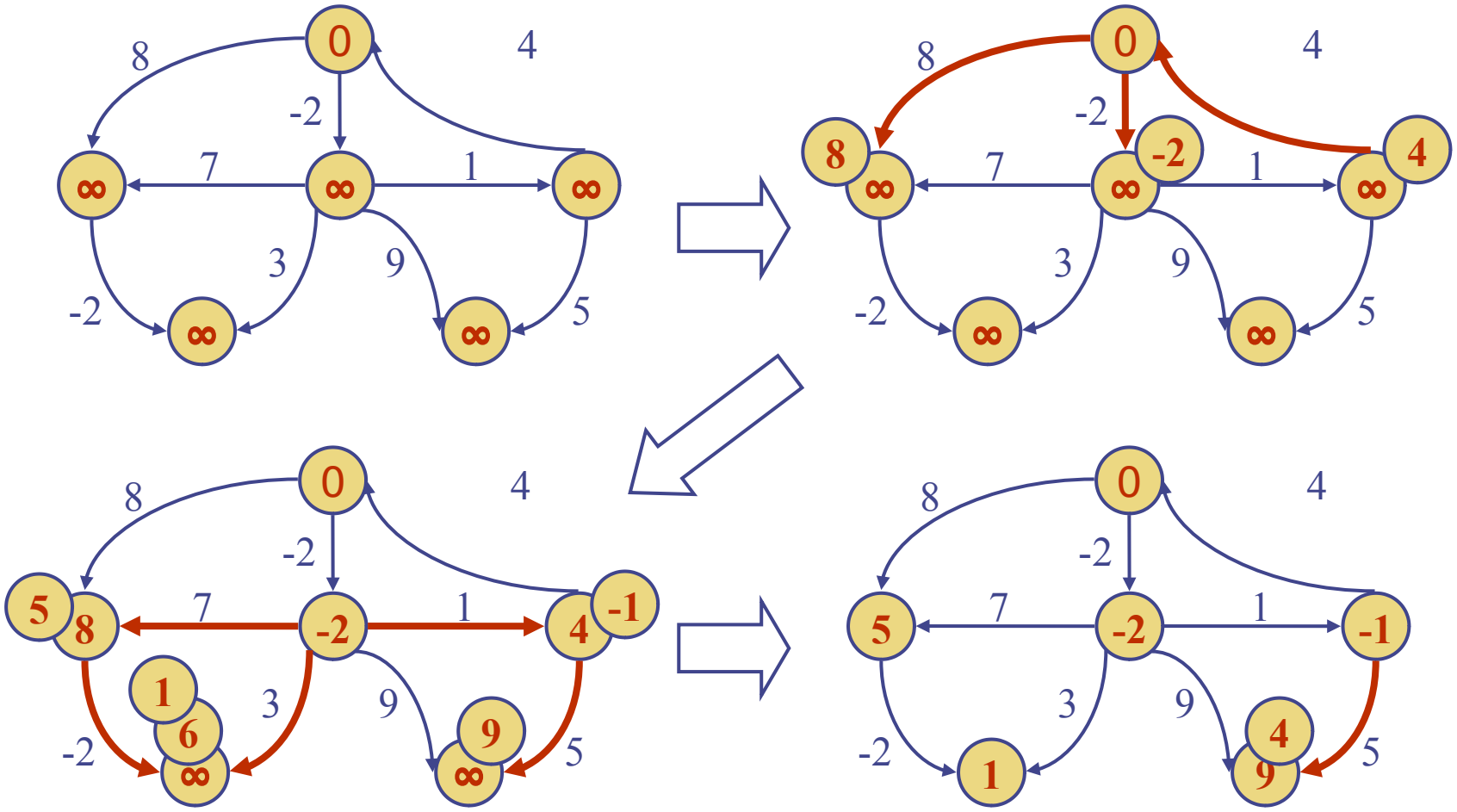
Algoritmo Bellman-Ford

- Su grafi non diretti con pesi negativi
  - Non termina
  - Dimostrare
- *Funziona su*
  - *Grafi diretti privi di cicli con peso negativo*
  - *Grafi non diretti con pesi non negativi*
- La i-esima iterazione individua i cammini minimi che usano i archi
- Complessità: O(mn)
- Può essere modificato per individuare l'esistenza di cicli con peso negativo → Come?

```
Algorithm BellmanFord(G, s)
    for all v ∈ G.vertices()
        if v = s
            setDistance(v, 0)
        else
            setDistance(v, ∞)
    for i = 1 to n − 1 do
        for each e ∈ G.edges()
            { relax edge e }
            u = G.origin(e)
            z = G.opposite(u,e)
            r = getDistance(u) + weight(e)
            if r < getDistance(z)
                setDistance(z,r)
```

Se nel ciclo n-esimo un arco è rilassato → ciclio di peso negativo
L'algoritmo può essere modificato per Individuare tali cicli

Bellman-Ford: esempio



Mantenimento albero dei cammini

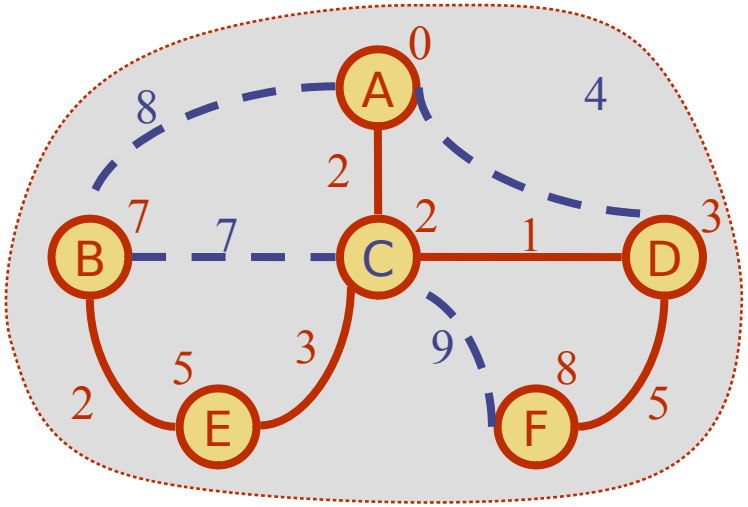
Implementazione albero dei cammini minimi

Per ogni vertice si memorizza il predecessore nell'albero dei cammini minimi

Ciò può essere realizzato in vari modi

Ad esempio, si associa un intero in {0, …, n-1} a ciascun vertice e si usa un array
L'entry i-esima dell'array è associata all'i-esimo Vertice e contiene l'intero associato al genitore nell'albero dei cammini minimi

Nell'esempio in figura si sono usate lettere



A	-
B	E
C	A
D	C
E	C
F	D

Mantenimento albero dei cammini minimi

- Quando si effettua il rilassamento di un nodo è necessario aggiornare i predecessori dei vertici la cui distanza dalla sorgente cambia
- Tale modifica può essere applicata sia all'algoritmo di Dijkstra che a quello di Bellman-Ford

```
relax(u) {
    for every v adjacent to u
        if d[v] > d[u] + w(u, v) {
            d[v] = d[u] + w(u, v)
            pred[v] = u
        }
```

Cammini minimi tra tutti i nodi

Problema

- Calcolare le distanza minime (e i corrispondenti cammini) tra ogni coppia di vertici
- Soluzione possibile
  - Eseguire l’algoritmo di Dijstra n volte (per ogni possibile vertice sorgente)
  - Complessità: $O(n(m + n)\log n)$