Progettazione di Algoritmi **Anno Accademico 2021–2022**

Note per la Lezione 26

Ugo Vaccaro

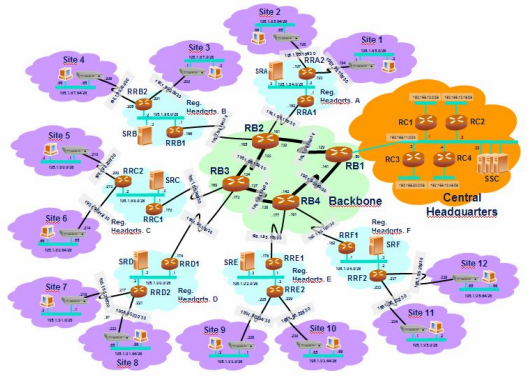
In questa lezione parleremo di problemi algoritmici relativi al calcolo ed alle applicazioni di *Cammini Minimi in Grafi*. Supporremo di avere in input al problema un grafo diretto $G = (V, E)$, un nodo sorgente s, ed una funzione $\ell : E \mapsto R_+$ che ci dice la "lunghezza" $\ell(e)$ di ogni arco $e \in E$ (per semplicità interpretiamo la quantità $\ell(e)$ come la lunghezza di un arco, ma nelle situazioni pratiche può rappresentare il costo di usare l'arco e, il tempo per attraversare l'arco e, etc.).. Il problema che vogliamo risolvere consiste nel voler determinazione di un cammino di *lunghezza minima* da s ad ogni altro nodo t nel grafo (dove la lunghezza di un cammino è pari alla somma delle lunghezze degli archi che lo formano). Più precisamente, vogliamo calcolare, per ogni nodo t in V un cammino $s, v_1, \ldots, v_k, t$ che parte da s e termina in t (quindi per cui vale che $(s, v_1), (v_1, v_2), \ldots, (v_k, t) \in E$), tale che la sua lunghezza, pari a $\ell(s, v_1) + \ell(v_1, v_2) + \ldots + \ell(v_k, t)$ sia la *minima possibile*, tra tutti i possibili cammini in G che partono da s e terminano in t. Tale (minima) somma delle lunghezze $\ell(s, v_1) + \ell(v_1, v_2) + \ldots + \ell(v_k, t)$ verrà anche chiamata la *distanza* di t da s.

Ricordiamo che nel caso in cui le lunghezze $\ell(e)$ degli archi sono tutte uguali tra di loro, allora il problema è risolvibile mediante una visita BFS del grafo. Infatti, minimizzare (in questo caso particolare!) la lunghezza totale del cammino equivale a minimizzare il numero di archi che attrversiamo per andare da s a t.

Per semplicità, un cammino di lunghezza totale minima da un nodo di partenza s ad un nodo arbitrario destinazione t verrà detto *cammino minimo da s a t*.

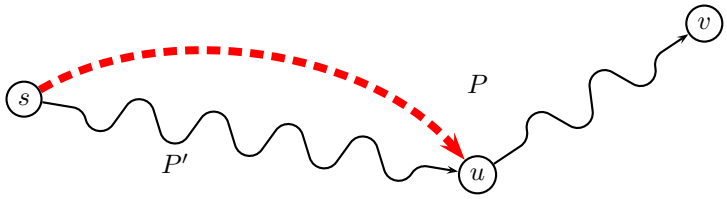
E perchè mai vogliamo calcolare cammini minimi in grafi? Perchè anche Google lo fà. Di sotto appare il calcolo del percorso *più breve* dalla stazione di Salerno all'Università mediante Google Maps.

Nelle reti di comunicazione, il protocollo *Open Shortest Path First* (OSPF) è un protocollo dinamico per l' instradamento di pacchetti, ampiamente usato in reti basate sull' Internet Protocol (IP). Esso fà uso dell'algoritmo oggetto di questa lezione.



Altri ambiti di applicazione sono in robotica, progetto di VLSI, etc.

Per il calcolo dei cammini minimi useremo la tecnica Greedy. Ricordiamo innanzitutto un fatto che notammo un pò di tempo fà: Se un cammino P di lunghezza minima dal nodo s al nodo v passa attraverso il nodo u, allora la parte di cammino P' da s a u è esso stesso un cammino di lunghezza minima da s a u



Perchè? Perchè se P' non lo fosse, allora esisterebbe un cammino differente s a v (linea rossa tratteggiata) di lunghezza inferiore. Ma ciò implicherebbe che si potrebbe sostituire il cammino P' con la linea rossa tratteggiata ed andare da s a v con un cammino di lunghezza totale inferiore a quella di P, contro l'ipotesi dell'ottimalità di P E perchè vogliamo ricordare questo fatto? Perchè esso ci dice che un cammino minimo da s ad un nodo generico v non ha una struttura arbitraria (e quindi potrebbe essere difficile da trovare), ma è una *estensione* di cammini minimi da s a nodi u più vicini ad s di quanto lo sia v.

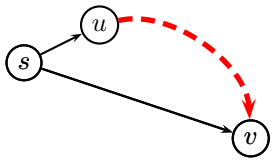
Questa osservazione dovrebbe suggerirci un'idea di tipo Greedy per calcolare cammini di lunghezza minima verso un numero sempre maggiore di nodi: *supposto di aver già costruito un insieme di cammini minimi che partono da s e raggiungono nodi $v_1, v_2, \ldots, v_k$, determiniamo l'estensione di lunghezza minore (visto che stiamo minimizzando) di uno di questi cammini, che raggiunge un nodo nuovo*, ovverosia tra tutti y che sono raggiungibili dai nodi $v_1, v_2, \ldots, v_k$ mediante l'uso di un solo arco del grafo, troviamo il nodo x che risulta essere a distanza minima da s. Poi iteriamo fin quando non abbiamo trovato i cammini di lunghezza minima da s ad *ogni* nodo del grafo

Di quest'idea almeno il punto di partenza è facile da realizzare: trova il nodo x adiacente a s che dista di meno da s e collegalo con il relativo arco.

Giustifichiamo un pò meglio l'idea, a partire dal primo passo, che possiamo riformulare nel seguente (equivalente) modo:

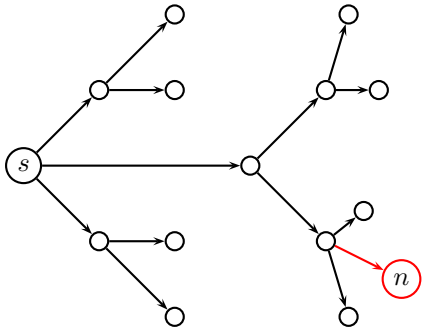
Tra tutti i nodi adiacenti al nodo di partenza s, trova il nodo v che che dista il meno possibile da s. Ovvero, tra tutti i nodi adiacenti al nodo di partenza s, sia v il nodo connesso a s con l'arco di lunghezza minima. Una volta determinato tale nodo v, collegalo ad s con il relativo arco del grafo.

E se abbiamo già sbagliato? Ovvero esisteva un altro cammino, di lunghezza totale minore di $\ell(s, v)$, che portava da s a v passando magari per un nodo u vicino di s, come nella figura di seguito:

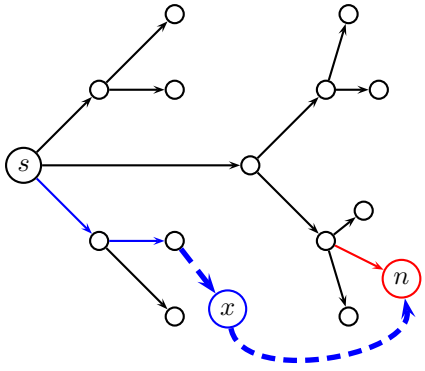


Non è possibile! Infatti ciò contraddirrebbe il fatto che v è il nodo adiacente ad s *che dista di meno* da s (u disterebbe ancora di meno). Quindi non abbiamo sbagliato già fin dall'inizio...

Potremmo però aver sbagliato nell'iterazione che consiste, ricordiamo, in: *dato un insieme di cammini minimi che partono da s e raggiungono nodi $v_1, v_2, \ldots, v_k$, trova l'estensione di lunghezza minore di uno di questi cammini, che raggiunge un nodo nuovo.* Ovvero, vorremmo procedere nel modo seguente:



Dati i cammini minimi già costruiti, tra *tutti* i vertici adiacenti dei nodi ○ già considerati (ed indicati con il ○) andiamo a sceglierci quello che dista di meno da s, e lo colleghiamo con il rispettivo vicino. Sia n il nodo che aggiungiamo.



Potrebbe esistere un percorso più breve da s ad n? Ad esempio, quello rappresentato in blu? No! Perchè se esistesse, *non* sarebbe n il nodo che (tra gli adiacenti dei nodi ○ già considerati) dista *di meno* da s, in quanto x dista da s ancora di meno.

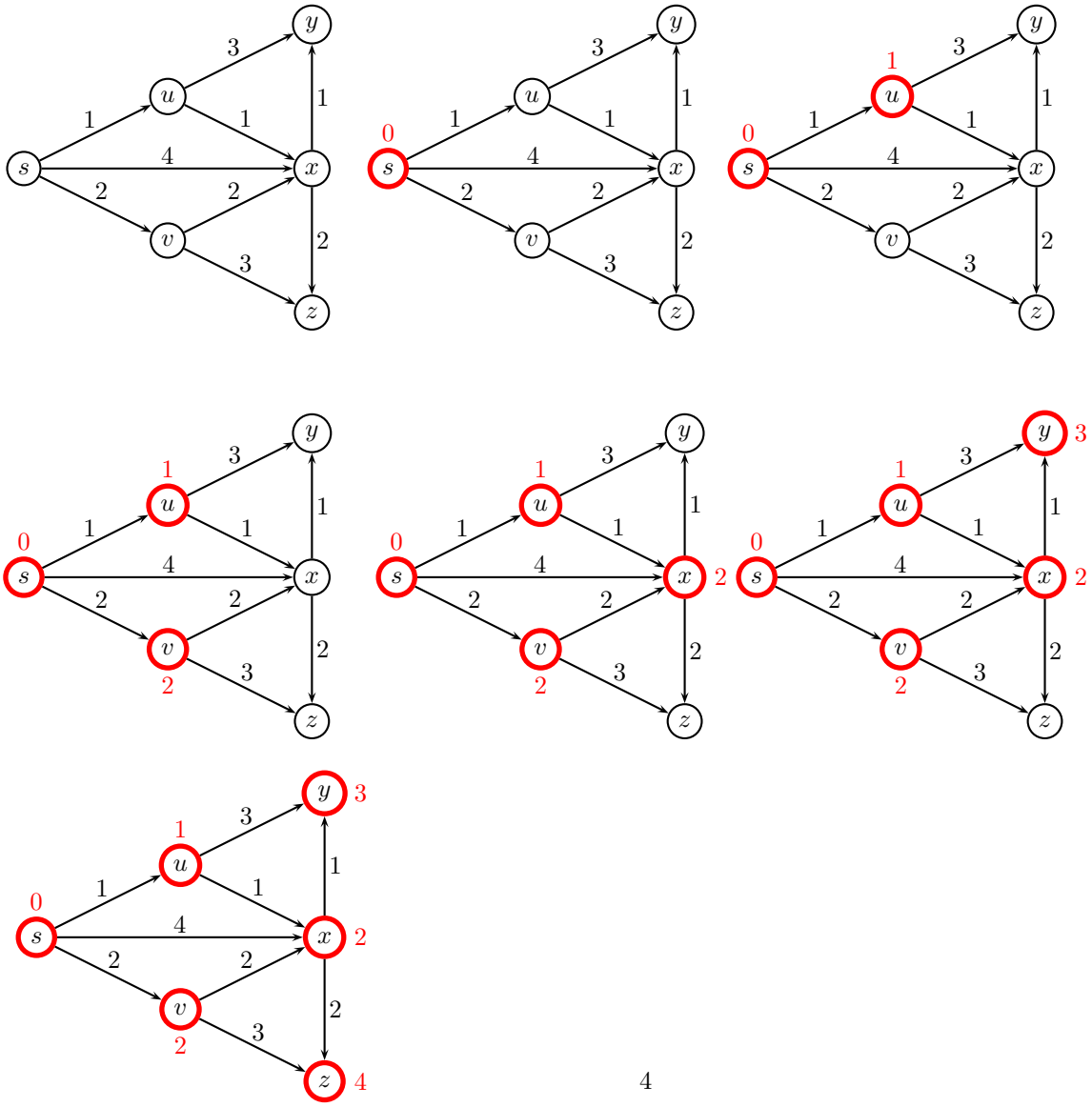
Vediamo quindi come costruire cammini minimi attraverso la tecnica Greedy (l'algoritmo risultante và sotto il nome di Algoritmo di Dijkstra, dal nome dell'ideatore.).

Iniziamo col formalizzare l'intuizione precedentemente guadagnata e, per il momento, preoccupiamoci solo di calcolare la lunghezza di un cammino di *lunghezza minima* dal nodo sorgente ad ogni nodo u nel grafo. Chiameremo tale lunghezza minima *la distanza del nodo u da s*. L'algoritmo può essere così descritto:

- Mantieni un insieme S di *nodi esplorati* per cui abbiamo già determinato la distanza $d[u]$ da s. Per i nodi v non ancora esplorati assumiamo che $d[v] = \infty$
- Inizializza $S = \{s\}$, $d[s] = 0$
- $\forall v \notin S$, calcola $d'[v] = \min_{e=(u,v):u\in S} d[u] + \ell(e)$
- Sia w il nodo che ha valore $d'[\cdot]$ minimo, aggiungi w in S e poni $d[w] = d'[w]$ (in altre parole w è, tra tutti i nodi $\notin S$ *ma* adiacenti ai nodi in S, quello più vicino a s).

L'algoritmo termina quando non ci sono più nodi raggiungibili da s mediante un cammino diretto. Da questo momento, assumeremo che tutti i nodi in G sono raggiungibili da s mediante un cammino diretto.

Vediamo un esempio di esecuzione: i nodi **rossi** sono quelli esplorati

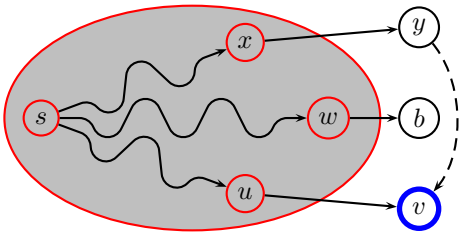


Analisi dell'algoritmo di Dijkstra.

Proviamo innanzitutto che i valori $d[u]$ calcolati dall'algoritmo corrispondono alla lunghezza minima di un cammino da s ad u. Ovvero, proviamo la seguente importante affermazione:

Ad ogni passo, $\forall v \in S$ $d[v]$ =distanza del nodo v da s

Vediamo come funziona l'algoritmo quando aggiunge un nodo v ad S:



Insieme S

Abbiamo calcolato $\forall t \notin S$, $d'[t] = \min_{e=(u,t):u\in S} d[u] + \ell(e)$, abbiamo scoperto che v ha il parametro d' *minore di tutti*, abbiamo posto $d[v] = d'[v] = d[u] + \ell(u,v)$ ed infine messo v in S. Potremmo aver sbagliato? (nel senso che esisteva un cammino da s a v più breve, di lunghezza $< d[v]$, ad es. passando da y). No! Infatti, innanzitutto vale che $d'[y] \geq d'[v]$, in quanto se fosse vero il contrario $d'[y] < d'[v]$, allora *non* sarebbe stato v il nodo con il parametro d' minore di tutti. Ma allora, se già $d'[y] \geq d'[v]$, *sicuramente* vale che la lunghezza del cammino che và da s a v (passando per y e per il percorso tratteggiato da y a v) non può essere di lunghezza di lunghezza $< d[v]$.

Quindi l'algoritmo di Dijkstra calcola correttamente le distanze dei nodi da s. *E se volessimo calcolarci anche i cammini da s di lungheza pari a tali distanze (ovvero i cammini di lunghezza minima)?*

Basterà memorizzarci, ogniqualvolta aggiungiamo un nodo v ad S, anche l'arco (u,v) che abbiamo usato, ovvero l'arco per cui

$$d[v] = d[u] + \ell(u,v)$$

In questo modo conosceremo l'ultimo arco del cammino di lunghezza minima da s a v, se lo abbiamo fatto anche per u allora conosciamo anche l'ultimo arco del cammino di lunghezza minima da s a u, *e quindi gli ultimi due archi* del cammino di lunghezza minima da s a v, e così via...

Implementazione dell'algoritmo di Dijkstra.

```
Inizializza S = {s}, d[s] = 0, d'[v] = ∞ ∀v ∈ V − {s}
While S ≠ V
    tra tutti i nodi w ∈ V − S per cui esiste un arco entrante (u, w), per qualche u ∈ S,
    seleziona un nodo v per cui d'[v] = min_{e=(u,v):u∈S} d[u] + ℓ(e) è più piccolo possibile
    Aggiungi v a S e poni d[v] = d'[v]
```

Ad ogni istante, strutturiamo l'insieme dei nodi in $V - S$ come una MinCoda a Priorità Q, in base ai valori $d'[\cdot]$ a loro associati.

Ad ogni iterazione estraiamo da Q il nodo v con $d'[v]$ minimo, lo mettiamo in S e aggiorniamo Q (come?).

Se $w \in Q$ è tale che $(v,w) \notin E$, allora $d'[w] = \min_{e=(u,w):u\in S} d[u] + \ell(e)$ rimane inalterato, se invece $(v,w) \in E$, allora $d'[w] = \min_{e=(u,w):u\in S} d[u] + \ell(e)$ può cambiare (diminuire) e occorrerà con un'operazione di tipo

`DecreaseKey` aggiustare la struttura della coda a priorità. Quindi, per ogni generico arco (x, y) chiamaremo `DecreaseKey` al più una volta, quando x viene aggiunto a S (il che avviene una volta sola).

Mettendo tutto insieme...

> Usando una coda a priorità, l'algoritmo di Dijkstra può essere implementato in un grafo con n nodi ed m archi in modo da richiedere tempo $O(m)$, più il tempo per eseguire n `ExtractMin` ed m `DecreaseKey`.

Se usiamo un min-heap per implementare la coda a priorità in questione, ogni operazione di `ExtractMin` e `DecreaseKey` richiede tempo $O(\log n)$, per un gran totale di $O(m \log n)$ operazioni per implementare l'algoritmo di Dijkstra.

Cliccare qui per un esempio di esecuzione dell'algoritmo.