APA Modulo 1 Lezione 5

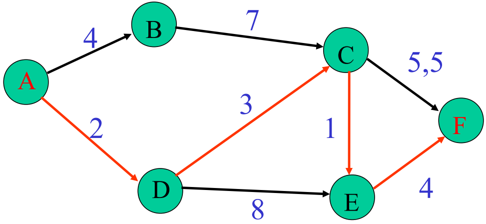
Elena Zucca

20 marzo 2020

Cammini minimi da nodo sorgente

- grafo pesato: $G = (V, E)$ + funzione $c_{\_} : E \to \mathbb{R}$
- i pesi (costi) possono rappresentare distanze, tempi di percorrenza, costi di attività, etc.
- costo di un cammino da s a t = somma dei pesi degli archi
- cammino da s a t minimo se ha peso minimo (detto distanza) fra tutti i cammini da s a t
  se pesi = 1 distanza nel senso usuale
- se non esistono cicli con peso negativo un cammino minimo fra due nodi connessi esiste sempre, può non essere unico
- problema: dato nodo di partenza s, trovare per ogni nodo u del grafo il cammino minimo da s a u

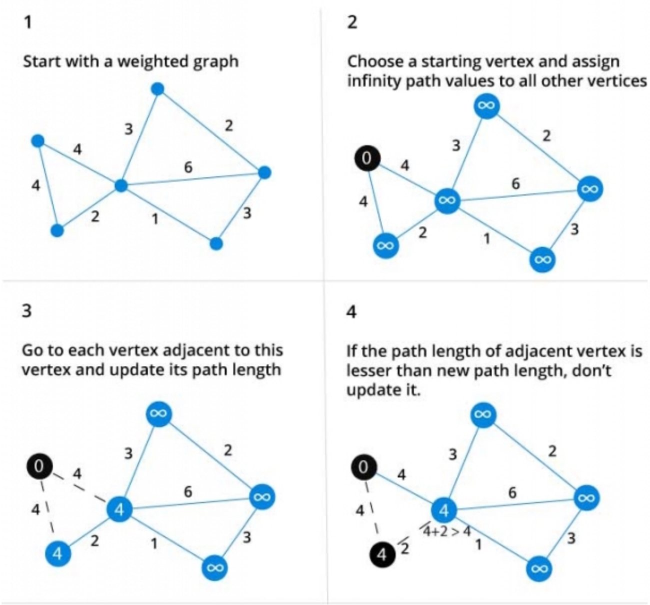
Esempio



A, D, C, E, F è un (il) cammino minimo da A ad F

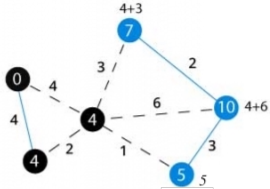
Algoritmo di Dijkstra (1959)

- è una visita in ampiezza in cui a ogni passo:
- per i nodi già visitati (neri) si ha la distanza
- per i non visitati si ha distanza provvisoria
  $=$ lunghezza minima di un cammino di soli nodi neri
- si estrae il nodo a distanza provvisoria minima (che risulta essere la distanza)
- si aggiornano le distanze provvisorie dei nodi adiacenti al nodo estratto tenendo conto del nuovo arco
- punto chiave: funziona se pesi non negativi
- conviene rappresentare l'insieme dei nodi ancora da visitare come coda a priorità (heap) - Johnson 1977
- per non avere caso a parte, conviene all'inizio avere distanza provvisoria "infinito"


1
Start with a weighted graph
2
Choose a starting vertex and assign infinity path values to all other vertices
3
Go to each vertex adjacent to this vertex and update its path length
4
If the path length of adjacent vertex is lesser than new path length, don't update it.
4+2 > 4

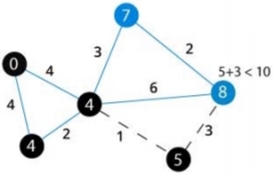
5

Avoid updating path lengths of already visited vertices



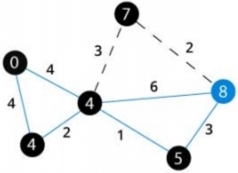
6

After each iteration, we pick the unvisited vertex with least path length. So we chose 5 before 7



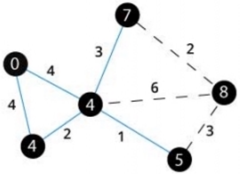
7

Notice how the rightmost vertex has its path length updated twice



8

Repeat until all the vertices have been visited



Pseudocodice

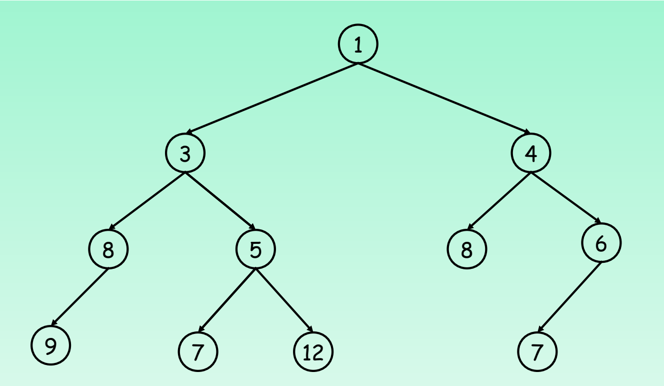
```
Djikstra(G,s)
  for each (u nodo in G)
    dist[u] = ∞// tutti i nodi sono bianchi
   parent[s] = null; dist[s] = 0 //s diventa grigio
  Q = heap vuoto
  for each (u nodo in G)  Q.add(u,dist[u])
  while (Q non vuota)
      u = Q.getMin()//nodo a distanza provvisoria minima
      //u diventa nero
       for each ((u,v) arco in G)
      //v diventa o resta grigio
           if (dist[u]+c_{u,v} < dist[v])
            parent[v] = u; dist[v] = dist[u] + c_{u,v}
            Q.changePriority(v, dist[v]) //moveUp
```

Non necessario marcare i nodi

```
if (dist[u]+c_{u,v} < dist[v])//se v nero falso
  parent[v] = u; dist[v] = dist[u] + c_{u,v}
  Q.changePriority(v, dist[v]) //moveUp
```

perché, come proveremo, se `v` nero `dist[v]` è già la distanza

Ripasso heap



Ripasso heap

- albero binario quasi completo, cioè completo fino al penultimo livello quindi bilanciato: altezza $O(\log n)$ con n numero dei nodi
- chiave nodo $\leq$ chiavi dei figli
- per fare in modo che l'albero rimanga quasi completo:
  - add: si inserisce elemento come foglia e poi si fa risalire al posto giusto mediante scambi successivi col genitore (move up)
  - getMin: si estrae il minimo, poi si sposta come radice una foglia dell'ultimo livello, e si fa scendere al posto giusto mediante scambi successivi col minore dei figli (move down)
  - changePriority: si cambia la priorità e poi si fa risalire al posto giusto mediante scambi successivi col genitore (move up)
- dato che l'albero è bilanciato queste operazioni sono $O(\log n)$

Correttezza

- $d(u)$ distanza (lunghezza di un cammino minimo) da s a u
- invariante composta di due parti:
  1. per ogni nodo u non in `Q`, ossia "nero"
     $\texttt{dist}[u] = d(u)$
  2. per ogni nodo u in `Q`
     $\texttt{dist}[u] = d_{\setminus\texttt{Q}}(u) =$ lunghezza di un cammino minimo da s a u i cui nodi, tranne u, non sono in `Q`, ossia sono nodi "neri"
     convenzione: se questo cammino non esiste (u è "bianco") $d_{\setminus\texttt{Q}}(u) = \infty$

L'invariante vale all'inizio

1. per ogni nodo u non in $\mathtt{Q}$, ossia "nero", $\mathtt{dist}[u] = d(u)$
   vale banalmente perché non ci sono nodi neri (tutti i nodi sono in coda)
2. per ogni nodo u in $\mathtt{Q}$ $\mathtt{dist}[u] = d_{\setminus \mathtt{Q}}(u)$
   vale banalmente perché la distanza è per tutti infinito, tranne che per s per cui vale 0

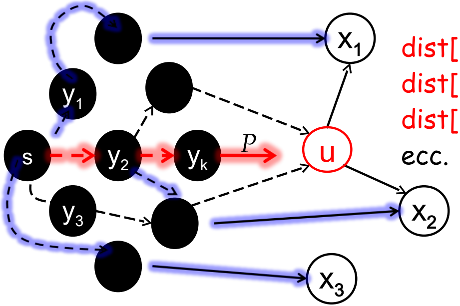
L'invariante si mantiene

- viene estratto u con $\mathtt{dist}[u]$ minima
- $\mathtt{dist}[u]$ è il costo di un cammino P da s a u, per invariante (2) minimo tra quelli di soli nodi neri (tranne l'ultimo)
- tesi: tale cammino è anche il minimo in assoluto da s a u
- per assurdo: esista un cammino da s a u di costo $\leq$ quello di P
- deve necessariamente contenere nodi non neri, sia w il primo
- quindi è della forma $\mathrm{P}_1\mathrm{P}_2$ con P_1 cammino da s a w e P_2 da w a u
- ma P_1, essendo di nodi neri tranne l'ultimo, ha costo $\geq$ quello di P
- quindi se pesi ≥ 0 a maggior ragione $\mathrm{P}_1\mathrm{P}_2$ ha costo $\geq$ quello di P

L'invariante si mantiene

- abbiamo provato che aggiungendo u l'invariante (1) vale ancora:
  per tutti i nodi neri, compreso il nuovo nodo nero u, è stato trovato il cammino minimo
- occorre però ripristinare l'invariante (2):
  per ogni nodo v in `Q`, $\texttt{dist}[v]$ = minima lunghezza cammino da s a v i cui nodi sono tutti, eccetto v, neri
- ora fra i nodi neri c'è anche u
  bisogna controllare se per qualche v adiacente a u il cammino da s a v passante per u è $\leq$ di quello trovato precedentemente e in tal caso aggiornare $\texttt{dist}[v]$ e $\texttt{parent}[v]$

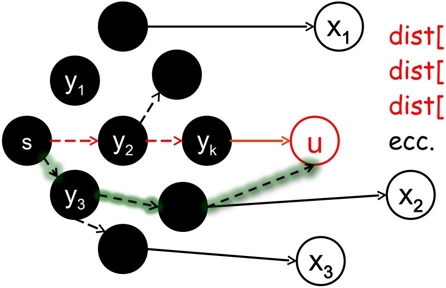
Graficamente



dist[u] < dist[x_1],
dist[u] < dist[x_2],
dist[u] < dist[x_3],
ecc.

il nodo u è "il più vicino a s" fra i nodi adiacenti a nodi neri, cioè
P è il "più corto" fra tutti i cammini da s a nodi adiacenti a nodi neri
tesi: P è il minimo fra tutti i cammini da s a u

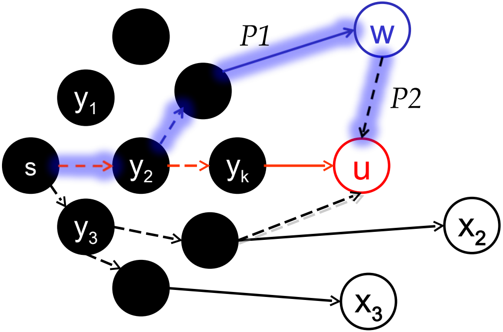
Graficamente



dist[u] < dist[x_1],
dist[u] < dist[x_2],
dist[u] < dist[x_3],
ecc.

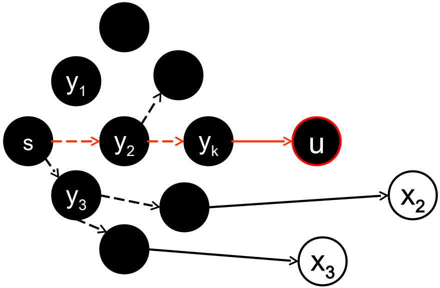
infatti: se ho un altro cammino da s a u che passa solo attraverso nodi neri, so già che è “più lungo” di P

Graficamente



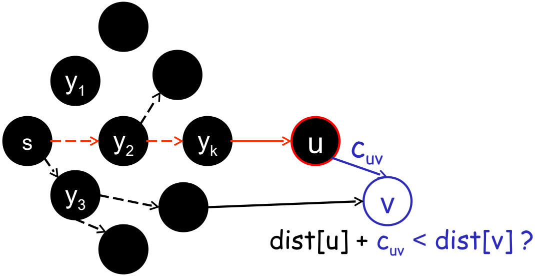
se ho un altro cammino da *s* a *u* che attraversa anche nodi non neri, sia *w* il primo, P_1 è “più lungo” di P, quindi anche $\mathrm{P}_1\mathrm{P}_2$ è “più lungo” di P

Graficamente



il nodo *u* estratto dalla coda viene correttamente aggiunto all'insieme dei nodi neri, e l'invariante (1) vale ancora
cioè è ancora vero che per tutti i nodi neri, compreso il nuovo nero *u*, è stato trovato il cammino minimo

Graficamente



bisogna controllare se per qualche v adiacente a u il cammino da s a v passante per u è $\leq$ di quello trovato precedentemente
e in tal caso aggiornare $\mathtt{dist}[v]$ e $\mathtt{parent}[v]$

Postcondizione e terminazione

- Al termine dell'esecuzione dell'algoritmo la coda è vuota, quindi tutti i nodi sono neri
- poiché vale l'invariante (1), per ogni nodo è stato trovato il cammino minimo da s
- funzione di terminazione: numero di nodi nello heap (non neri)

Osservazione importante

- per dimostrare che il ciclo mantiene l'invariante (1), che è quello che ci interessa, è necessario dimostrare che imantiene anche l'invariante (2)
- invariante ausiliaria: necessaria in molti algoritmi

Complessità: fase di inizializzazione

```
for each (u nodo in G)  dist[u]= ∞
parent[s]= null; dist[s]= 0;  Q= heap vuoto
for each (u nodo in G)  Q.add(u,dist[u])
```

$O(n)$, $O(1)$, $O(n \log n)$

in realtà

```
for each (u nodo in G)  Q.add(u,dist[u])
```

$O(n)$

Complessità: ciclo

```
while (Q non vuota)
  u = Q.getMin()    O(log n)
  for each ((u,v) arco in G)
    if (dist[u]+c_{u,v}<dist[v])
      parent[v]= u; dist[v]= dist[u]+c_{u,v}
      Q.changePriority(v, dist[v])    O(log n)
```

- totale `getMin`: $O(n \log n)$
- totale blocchi `if`: $O(m \log n)$

Complessità: calcolo adiacenti

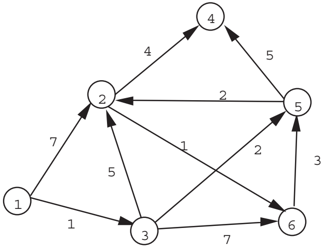
```
while (Q non vuota)
  u = Q.getMin()
  for each ((u,v) arco in G)
    ...
```

- liste di adiacenza: $O(\delta(u))$
  in tutto $O(m)$ (ogni lista di adiacenza viene scandita una volta sola)
- lista di archi: $O(m)$, in tutto $O(n \cdot m)$
- matrice di adiacenza: $O(n)$, in tutto $O(n^2)$

Complessità finale e vantaggi heap

- assumiamo liste di adiacenza
- complessivamente si ha $O((m+n)\log n)$
- se invece di heap sequenza non ordinata:
  inserimento in coda e cambio priorità costante, ma estrazione del minimo $O(n)$
- analogamente con sequenza ordinata: estrazione minimo costante, ma inserimento/cambio prioritá $O(n)$
- quindi complessità algoritmo $O(n^2)$
- se il grafo è denso, cioè $m = O(n^2)$, $O(n^2 \log n)$ è peggiore di $O(n^2)$

Esempio con tabella heap



	1	2	3	4	5	6
	0	∞	∞	∞	∞	∞
1	-	7	1	∞	∞	∞
3	-	6	-	∞	3	8
5	-	5	-	8	-	8
2	-	-	-	8	-	6
6	-	-	-	8	-	-
4	-	-	-	-	-	-

riga = iterazione; – se distanza definitiva, altrimenti provvisoria

prima colonna = nodo estratto