Corso di laurea in Informatica
Progettazione d'algoritmi

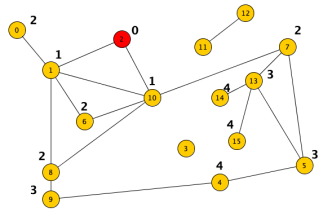
I grafi 7

Angelo Monti

SAPIENZA
UNIVERSITÀ DI ROMA

Il calcolo delle distanze in G e la visita in ampiezza (BFS)

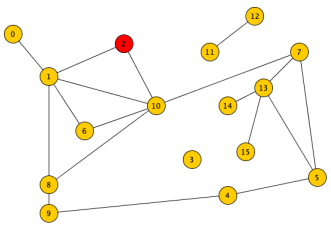
Dati due nodi a e b di un grafo G, definiamo distanza (minima) in G di a da b il numero minimo di archi che bisogna attraversare per raggiungere b a partire da a. La distanza è posta a $+\infty$ se b non è raggiungibile partendo da a.

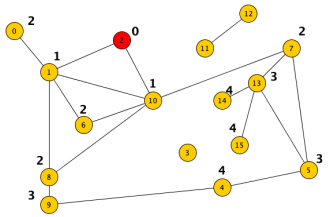


In figura sono riportate accanto ad ogni nodo le distanze dal nodo 2 in rosso

Problema: dato un grafo G ed un suo nodo x vogliamo conoscere le distanze di tutti i nodi di G da x.

Problema:
dato un grafo G ed un suo nodo x vogliamo conoscere le distanze di tutti i nodi di G da x.

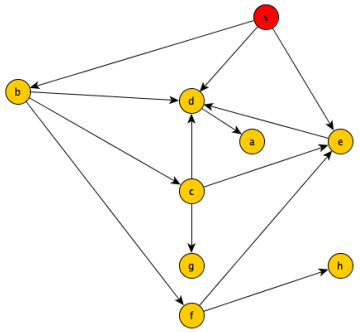


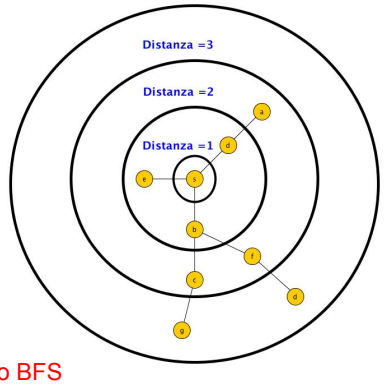


Più precisamente vogliamo calcolare il vettore delle distanze D, dove in $D[y]$ troviamo la distanza di y da x.

$$D = \boxed{2}\boxed{1}\boxed{0}\boxed{+\infty}\boxed{4}\boxed{3}\boxed{2}\boxed{2}\boxed{2}\boxed{3}\boxed{1}\boxed{+\infty}\boxed{+\infty}\boxed{3}\boxed{4}\boxed{4}$$

La visita in ampiezza (Breadth First Search) esplora i nodi del grafo partendo da quelli a distanza 1 dalla sorgente s. Poi visita quelli a distanza 2 e così via. L'algoritmo visita tutti i vertici a livello k prima di passare a quelli a livello $k+1$.

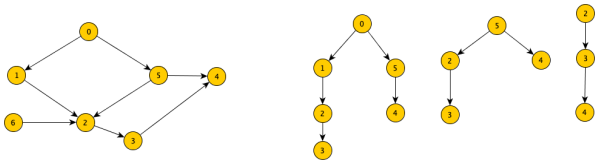




- Si genera un albero detto albero BFS
- Si visitano nodi sempre più distanti dalla radice

Per effettuare questo tipo di visita manteniamo in una coda i nodi visitati i cui adiacenti non sono stati ancora esaminati.
Ad ogni passo, preleviamo il primo nodo dalla coda, esaminiamo i suoi adiacenti e se scopriamo un nuovo nodo lo visitiamo e lo aggiungiamo alla coda.

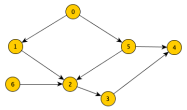
$G =$



sulla destra i tre alberi BFS risultanti da tre visite BFS del grafo G che partono da 0, 5 e 2, rispettivamente.

Cominciamo a prendere dimestichezza con la BFS applicando questo tipo di visita per ottenere i nodi raggiungibili a partire da un fissato nodo x.
Si comincia con una coda contenente il solo nodo di partenza x. Fino a che la coda non risulta vuota: ad ogni passo un nodo viene estratto dalla coda, tutti i suoi adiacenti vengono visitati e messi in coda:

```
def BFS(x,G):
    # restituisce i raggiungibili da x
    visitati=[0 for _ in G]
    visitati[x] = 1
    coda=[x]
    while coda:
        u = coda.pop(0)
        for y in G[u] :
            if visitati[y]==0: #Nota: y va in coda solo se non visitato
                visitati[y] = 1
                coda.append(y)
    return visitati
```



$visitati =$ | 1 | 1 | 1 | 1 | 1 | 1 | 0 |

G=[[1,5], [2], [3], [4], [ ], [2,4], [2]]
>>> BFS(0,G)
[1,1,1,1,1,1,0]

Alla fine $visitati[u] = 1$ se e solo se u è raggiungibile da x


```
def BFS(x,G):
    # restituisce i raggiungibili da x
    visitati=[0 for _ in G]
    visitati[x] = 1
    coda=[x]
    while coda:
        u = coda.pop(0)
        for y in G[u] :
            if visitati[y]==0: #Nota: y va in coda solo se non visitato
                visitati[y] = 1
                coda.append(y)
    return visitati
```

Correttezza:

Alla fine $visitati[u]$ contiene 1 se il nodo u è raggiungibile da x, 0 altrimenti.

- **se u raggiungibile a partire da x allora $visitati[u] = 1$.** Se u raggiungibile c'è cammino P da x a u. Supponiamo per assurdo che al termine $Visitati[u] = 0$. Sia b il primo nodo con $V[b] = 0$ che incontro nel cammino P e a il suo predecessore (questi due nodi esistono perché il cammino comincia con x e $visitati[x] = 1$ e termina con u e $visitati[u] = 0$). Un nodo che diventa 1 finisce in coda, quindi a è finito in coda, ma l'algoritmo si ferma quando la coda si svuota quindi c'è stato un momento in cui a è stato estratto dalla coda, in quel momento tutti i suoi vicini sono stati controllati e se non visitati vengono visitati (e poi inseriti in coda) quindi anche b è stato controllato e se non ancora visitato sarebbe stato visitato e posto a 1.
- **se u non raggiungibile a partire da x allora $visitati[u] = 0$.** Supponiamo per assurdo che al termine $visitati[u] = 1$. $Visitati[u]$ diventa 1 quando viene inserito in coda ma questo accade perché risulta adiacente ad un nodo che viene estratto dalla coda, andando all'indietro questo processo deve terminare perché ciascun nodo viene inserito al più una volta nella coda quindi si crea un cammino da x a u il che è assurdo avendo supposto che u non è raggiungibile da x.


```
def BFS(x,G):
    # restituisce i raggiungibili da x
    visitati=[0 for _ in G]
    visitati[x] = 1
    coda=[x]
    while coda:
        u = coda.pop(0)
        for y in G[u] :
            if visitati[y]==0: #Nota: y va in coda solo se non visitato
                visitati[y] = 1
                coda.append(y)
    return visitati
```

Complessità

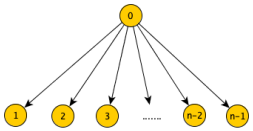
- un nodo finisce in coda al più una volta (poi risulterà visitato e non ci finisce più) quindi il $while$ verrà eseguito $O(n)$ volte.
- le liste di adiacenza dei nodi verranno scorse al più una volta quindi il costo totale dei $for\ \ y\ \ in\ \ G[u]$ sarà $O(m)$.

Se le operazione di inserimento e cancellazione dalla coda richiedessero tempo $\Theta(1)$ avremmo complessità $O(n+m)$.

Abbiamo implementato la coda tramite lista inserendo in coda e cancellando in testa.

- **inserimento in coda** ($append()$) costa $O(1)$
- **estrazione in testa** ($pop(0)$) ha un costo proporzionale al numero di elementi presenti in quel momento nella coda/lista e questi elementi possono essere anche $O(n)$.

Quindi il costo di questa implementazione dell'algoritmo è $O(n^2)$.



Non è difficile dimostrare che al caso pessimo la complessità è $\Theta(n^2)$ (si pensi ad esempio ad un grafo diretto in cui il nodo 0 è una sorgente universale e tutti gli altri nodi sono dei pozzi, in questo caso durante l'esecuzione dell'algoritmo, si assisterà all'inserimento in coda di tutti i nodi del grafo come adiacenti al nodo 0 e poi ad $n-1$ estrazioni consecutive di costo decrescente $n-1,\ n-2,\ n-3,\ \ldots\ 1$

Implementazione alternativa di costo $O(n+m)$.

Nella coda effettuo solo cancellazioni logiche e non effettive: uso un puntatore *inizio* che indica l'inizio della coda all'interno della lista. Il puntatore si incrementa ogni volta che si cancella dalla coda. (Il test di coda vuota va ora opportunamente modificato)

```
def BFS(x,G):
    # restituisce i raggiungibili da x
    visitati=[0 for _ in G]
    visitati[x] = 1
    coda=[x]
    inizio=0
    while len(coda)>inizio:
        u = coda[inizio]
        inizio+=1
        for y in G[u] :
            if visitati[y]==0: #Nota: y va in coda solo se non visitato
                visitati[y] = 1
                coda.append(y)
    return visitati
```


Implementazione di complessità $O(n+m)$ che fa uso della struttura dati deque.

Le collezioni in python sono contenitori di dati usati per registrare collezioni di dati. Python ne mette a disposizione diverse con le loro operazioni, ad esempio *list*, *dir*, *set*, *tuple*. Il modulo *collections* ne mette a disposizione altre.

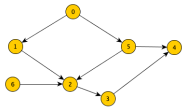
Dal modulo "collections" posso importare la struttura dati *deque* (double ended queue) che permette di effettuare inserzioni e le cancellazioni da entrambi i lati di una lista in tempo $O(1)$ in particolare posso estrarre a sinistra (*popleft*()) e inserire a sinistra (*appendleft*())

```
def BFS(x,G):
    # restituisce i raggiungibili da x
    visitati=[0 for _ in G]
    visitati[x] = 1
    from collections import deque
    coda = deque([x])
    while coda:
        u = coda.popleft()
        for y in G[u] :
            if visitati[y]==0: #Nota: y va in coda solo se non visitato
                visitati[y] = 1
                coda.append(y)
    return visitati
```


Modifichiamo ora leggermente la procedura di visita in modo che restituisca in $O(n+m)$ il vettore delle distanze D (anzicché il vettore dei visitati):

Al nodo x viene assegnata distanza zero e a ciascun nodo via via visitato viene assegnata la distanza corrispondente al padre incrementata di 1.

```
def DistanzeBFS(x,G):
    # restituisce il vettore delle distanze
    D=[ -1 for _ in G]
    D[x] = 0
    coda = [x]
    inizio = 0
    while len(coda) > inizio:
        u = coda[inizio]
        inizio+= 1
        for y in G[u]:
            if D[y] == -1: #Nota y va in coda solo se non visitato
                D[y] = D[u] + 1
                coda.append(y)
    return D
```



G=[[1,5], [2], [3], [4], [ ], [2,4], [2]]

```
>>> DistanzeBFS(0,G)
[0, 1, 2, 3, 2, 1, -1]
>>> DistanzeBFS(6,G)
[-1, -1, 1, 2, 3, -1, 0]
>>> DistanzeBFS(4,G)
[-1, -1, -1, -1, 0, -1, -1]
```

Al termine $D[u]$ conterrà -1 se il nodo u non è raggiungibile a partire da x, la distanza minima di u da x altrimenti.

Resta da dimostrare la correttezza della procedura distanzeBFS(x,G):

Cominciamo col dimostrare la seguente proprietà:

Siano v' e v due nodi raggiungibili a partire da x. Se v' finisce in coda prima di v allora $D[v'] \leq D[v]$.

La prova è per assurdo. Sia v' il primo nodo che contraddice la proprietà, vale a dire il primo nodo a finire in coda per cui esiste un nodo v finito in coda dopo con $D[v'] > D[v]$. I due nodi sono finiti in coda a causa di nodi-padre scorrendo le cui liste di adiacenza sono stati individuati, siano dunque $P_{v'}$ e P_v i loro due padri, nota che $P_{v'}$ e P_v sono diversi altrimenti $D[v'] = D[v]$ inoltre $P_{v'}$ è finito in coda prima di P_v e da $D[v'] > D[v]$ ricavo $D[P_{v'}] > D[P_v]$ ma questo contraddice il fatto che v' era il primo vertice a contraddire la proprietà.

siamo ora pronti a dimostrare la correttezza dell'algoritmo proposto:

Al termine della procedura per ogni nodo u raggiungibile a partire da x la distanza minima di u da x sarà $D[u]$.

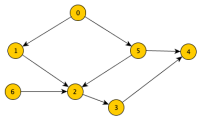
La prova è per induzione sulla distanza da x dei vari nodi. Vero per $d = 0$ (x è l'unico nodo la cui distanza vale 0 e $D[x] = 0$). Assumiamo $D[u]$ corretto per tutti i nodi con distanza al più $d > 0$ e consideriamo un qualunque nodo a distanza $d+1$. C'è dunque un cammino lungo $d+1$ che porta ad u, sia v il precedessore di u su questo cammino. Se ad u è stata assegnata la distanza scorrendo la lista di adiacenza di v allora poiché per ipotesi induttiva v ha distanza d il valore di $D[u]$ sarà proprio $d+1$. Sia dunque v' il nodo scorrendo la cui lista di adiacenza si è incontrato u. Per come funziona l'algoritmo v' è stato estratto prima di v dalla coda e **per la proprietà prima dimostrata** si ha $D[v'] \leq D[v] = d$. Se $D[v'] < D[v]$ avrei trovato un cammino che mi porta a u di costo $D[v'] + 1 < d + 1$ contro l'ipotesi che $d+1$ è la distanza minima, deve quindi aversi $D[v'] = D[v]$ ma allora si ha correttamente $D[u] = d + 1$.

Modifichiamo ancora la procedura in modo che restituisca in $O(n+m)$ l'albero di visita BFS rappresentato tramite il vettore dei padri.

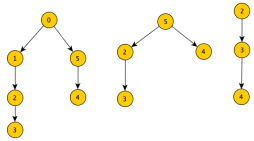
```
def PadreBFS(x,G):
    # restituisce il vettore delle distanze
    P=[ -1 for _ in G]
    P[x] = x
    inizio=0
    coda = [x]
    while len(coda) > inizio:
        u = coda[inizio]
        inizio+= 1
        for y in G[u]:
            if P[y] == -1: #Nota y va in coda solo se non visitato
                P[y] = u
                coda.append(y)
    return P
```

```
>>> PadreBFS(0,G)
[0, 0, 1, 2, 5, 0, -1]
>>> PadreBFS(5,G)
[-1, -1, 5, 2, 5, 5, -1]
>>> PadreBFS(2,G)
[-1, -1, 2, 2, 3, -1, -1]
```

$G =$



G=[[1,5], [2], [3], [4], [ ], [2,4], [2]]



$[0,0,1,2,5,0,-1]$ $[-1,-1,5,2,5,5,-1]$ $[-1,-1,2,2,3,-1,-1]$

Nota: grazie al vettore dei padri P con la procedura $cammini(x,P)$ in $O(n)$ posso ottenere un cammino minimo in G dalla radice dell'albero al nodo x (posto che x sia raggiungibile).