La visita in ampiezza (BFS)

Algoritmi 2 – Lezione 7

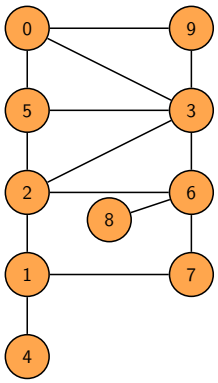
A. Monti
Sapienza Università di Roma

Corso: Algoritmi 2

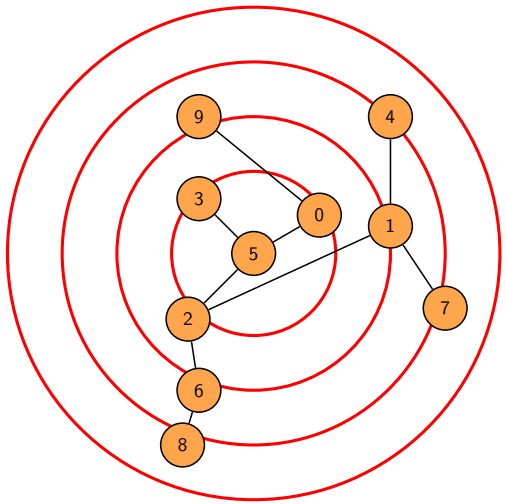
Indice

BFS: esplorazione livello per livello

- A differenza della visita in profondità (DFS) che esplora un singolo cammino il più a lungo possibile, la BFS esplora i nodi livello per livello.
- La BFS si comporta come un'onda che si espande, visitando prima i nodi vicini alla sorgente e poi, progressivamente, quelli più lontani.
- Con questo approccio, ogni nodo viene raggiunto seguendo il percorso più breve (in termini di numero di archi) dalla sorgente.



(a) Il grafo G.



(b) La visita $BFS(G, 5)$.

BFS: Implementazione (inefficiente) con lista Python

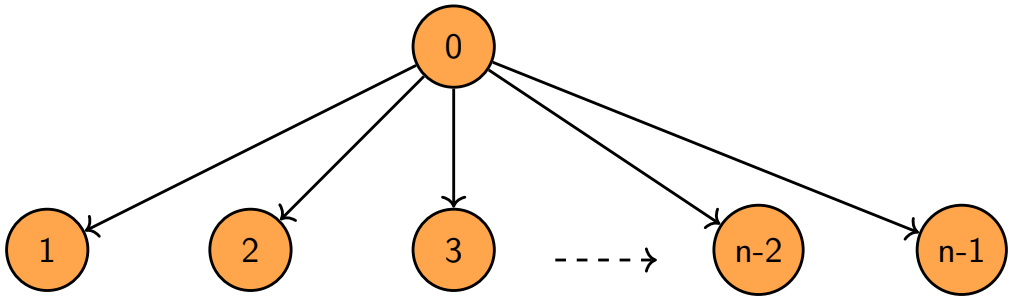
- Per gestire l'ordine di visita, la BFS utilizza una **coda** come struttura dati fondamentale. I nodi vengono aggiunti in coda e processati in base al principio "primo entrato, primo uscito".

```
def bfs_array(G, s):
    n = len(G)
    visitati = [0]*n
    coda = [s]
    visitati[s] = 1
    while coda:
        u = coda.pop(0) # Costo O(n)
        for v in G[u]:
            if visitati[v] == 0:
                visitati[v] = 1
                coda.append(v)
    return [ u for u in range(n) if visitati[u] == 1]
```

- L'operazione $coda.pop(0)$ ha un costo di $O(n)$ in quanto nel caso peggiore, la coda può arrivare a contenere fino a $O(n)$ elementi.
- Poiché $pop(0)$ viene eseguita per ogni nodo, la complessità totale di questa implementazione è $O(n^2)$.

Caso pessimo: lista Python

- Non è difficile convincersi che, nel caso pessimo, la complessità è $\Theta(n^2)$. Si consideri, ad esempio, il comportamento dell'algoritmo sul il grafo G in figura dove il nodo 0 funge da sorgente universale, mentre tutti gli altri nodi sono pozzi.



BFS: Implementazione efficiente con cancellazione logica

- Per ottenere la massima efficienza, è cruciale ricorrere ad un'implementazione della coda che consenta aggiunta e rimozione in tempo costante $O(1)$.
- Un modo per ottenere quest'effetto consiste nel ricorrere alla **Coda con cancellazioni logiche:** Si usa un indice *testa* che si sposta in avanti, simulando la rimozione senza spostare gli elementi.
- La complessità totale diviene $O(n + m)$.

```
def bfs_logico(G, s):
        n = len(G)
    visitati = [0] * n
    coda = [s]
    testa = 0
    visitati[s] = 1
    while testa < len(coda):
        u = coda[testa]
        testa += 1
        for v in G[u]:
            if  visitati[v] == 0:
                visitati[v] = 1
                coda.append(v)
    return [ u for u in range(n) if visitati[u] == 1]
```

BFS: Implementazione efficiente con *deque*

- In Python, la soluzione più efficiente consiste nell'utilizzare la struttura dati *deque* del modulo *collections* che è ottimizzata per operazioni a entrambe le estremità in tempo costante.
- La complessità totale risulta $O(n+m)$.

```
from collections import deque

def bfs_deque(G, s):
    n = len(G)
    visitati = [0]*n
    coda = deque( )
    coda.append(s)
    visitati[s] = 1
    while coda:
        u = coda.popleft( ) # O(1)
        for v in G[u]:
            if visitati[v] == 0:
                visitati[v] = 1
                coda.append(v)
    return [ u for u in range(n) if visitati[u] == 1]
```

Calcolo del vettore delle distanze

- La BFS visita i nodi in ordine di distanza dalla sorgente s. Questo permette di costruire un **vettore delle distanze** D, dove $D[i]$ è la distanza minima dal nodo sorgente s al nodo i. $D[i] = +\infty$ se i non è raggiungibile da s.

```
def VettoreDistanzeBFS(G, s):
    n = len(G)
    # inizializza le distanze a infinito
    D = [float('inf')]*n
    D[s] = 0
    coda = [s]
    testa = 0
    while testa < len(coda):
        u = coda[testa]
        testa += 1
        for v in G[u]:
            if D[v] == float('inf'): # se non visitato
                D[v] = D[u] + 1
                coda.append(v)
    return D
```

- Il passo cruciale è l'aggiornamento della distanza $D[v] = D[u] + 1$.

Albero BFS e cammini minimi

- La BFS costruisce implicitamente l' **albero BFS** ovvero il sottoinsieme del grafo formato da tutti i nodi raggiungibili dalla sorgente e dagli archi usati per raggiungerli.
  - Per ogni nodo v scoperto da un nodo u, l'arco (u, v) è un arco d'albero e u è il padre di v.
- L'albero BFS è l'**albero dei cammini minimi** dalla sorgente, una proprietà non condivisa dall'albero DFS.
- L'albero viene rappresentato dal **vettore dei padri** P, dove $P[v]$ è il nodo da cui si è raggiunto v per la prima volta durante la visita. $P[v] = -1$ se il nodo v non viene visitato.

Costruzione del vettore dei padri

```
def VettorePadriBFS(G, s):
    n = len(G)
    P = [-1] * n
    # la sorgente e' la  radice del cammino minimo
    P[s] = s
    coda = [s]
    testa = 0
    while testa < len(coda):
        u = coda[testa]
        testa += 1
        for v in G[u]:
            if P[vicino] == -1:
                #  v non visitato ha  predecessore u
                P[v] = u
                coda.append(v)
    return P
```

Definizione del diametro

Dato un grafo connesso G:

- la **distanza** $d(u, v)$ tra due nodi u e v del grafo è il numero minimo di archi da attraversare per andare dall'uno all'altro nodo.
- Il **diametro** del grafo è la massima distanza possibile tra due nodi qualsiasi del grafo. In altre parole, è il "percorso più lungo" che si può trovare all'interno del grafo.
- La formula matematica per il calcolo è la seguente:

$$\text{diametro}(G) = \max_{u,v \in V} d(u, v).$$

Algoritmo esaustivo in $O(nm)$

L'algoritmo dato qui di seguito calcola il diametro del grafo applicando direttamente la definizione.
Per ogni coppia di nodi, l'algoritmo calcola la loro distanza tramite una BFS e, infine, restituisce la distanza massima trovata.

```
def diametro_esaustivo(G):
    n = len(G)
    massimo = 0
    for nodo in range(n):
        D = VettoreDistanzeBFS(G, nodo)
        massimo = max(massimo, max(D))
    return massimo
```

- Questo algoritmo esaustivo ha una complessità di $O(n \cdot m)$ poiché esegue una BFS da ogni nodo.

Algoritmo efficiente per alberi)

Per gli **alberi**, esiste un algoritmo più efficiente di tempo $O(n)$ che utilizza solo due BFS:

- Esegui una prima BFS dal nodo 0 per trovare il nodo a lui più lontano, diciamo x.
- Esegui una seconda BFS a partire da x per trovare il nodo più lontano da x, diciamo y.
- La distanza $d(x, y)$ è il diametro dell'albero.

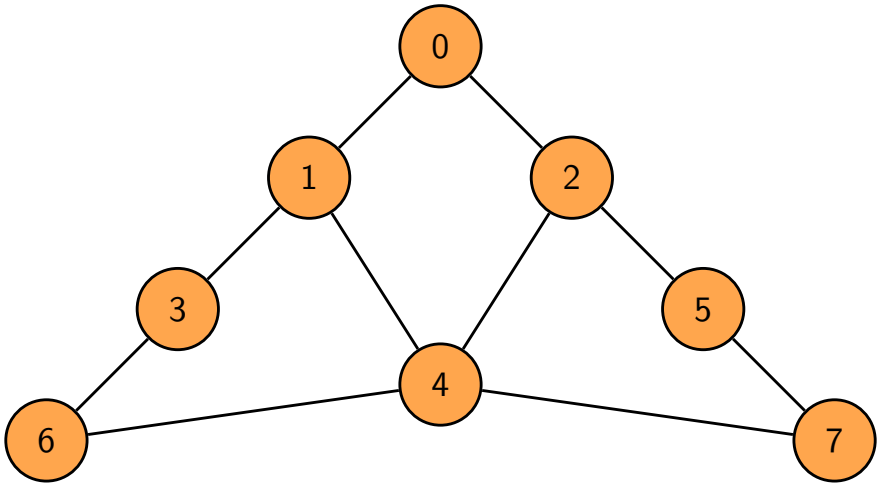
```
def DiametroAlbero(G):
    D = VettoreDistanzeBFS(G, 0)
    v = D.index(max(D))
    D = VettoreDistanzeBFS(G, v)
    return max(D)
```

- La complessità è $O(n)$ poiché una BFS su un albero costa $O(n+m) = O(n)$.

Controesempio: attenzione ai grafi

- **Importante:** Questo algoritmo funziona solo sugli alberi. Su grafi generali può restituire una distanza minore del diametro reale.

Considera ad esempio questo grafo:

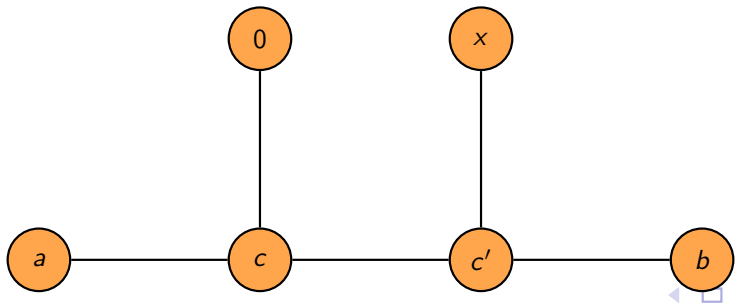


L'algoritmo delle due visite restituisce diametro 3 mentre il diametro del grafo è 4. Il cammino più lungo (di lunghezza 4) è quello che collega i nodi 3 e 5).

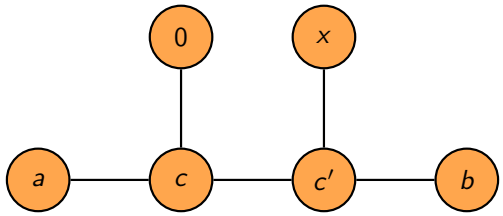
Dimostrazione di correttezza (1/2)

Teorema: Sia G un albero. Sia x un nodo a distanza massima dal nodo 0. Sia y un nodo a distanza massima da x. Allora, la distanza $d(x, y)$ è uguale al diametro di G.

Dimostrazione: Per dimostrare il teorema, procediamo per assurdo. Supponiamo che il diametro dell'albero non sia $d(x, y)$ da questo discende che 0 e x non sono estremi del diametro (altrimenti l'algoritmo sarebbe corretto). Siano a e b due nodi di G tali che $d(a, b)$ è il diametro dell'albero (a e b devono essere due foglie dell'albero). Sia c il nodo più vicino a 0 sul cammino da a a b e c' il nodo più vicino a x sul cammino da a a b.
Assumiamo, senza perdere di generalità, che c si trovi prima di c' sul cammino da a a b (nel caso opposto il ragionamento è analogo). La situazione è illustrata dalla seguente figura



Dimostrazione di correttezza (2/2)



Per definizione x è un nodo a distanza massima da 0. Pertanto

$$d(0, b) \leq d(0, x)$$

Sostituendo le distanze coi percorsi illustrati nella figura otteniamo

$$d(0, c') + d(c', b) \leq d(0, c') + d(c', x).$$

Questo implica che $d(c', b) \leq d(c', x)$.
A questo punto consideriamo il diametro $d(a, b)$. Abbiamo

$$d(a, b) = d(a, c') + d(c', b) \leq d(a, c') + d(c', x) = d(a, x).$$

Deve quindi aversi $d(a, b) \leq d(a, x)$. Poiché $d(a, b)$ è il diametro, deduciamo che $d(a, b) = d(a, x)$ ma allora x è un estremo di diametro contraddicendo l'ipotesi fatta.

ESERCIZI

Suggerimento: Prima di cercare le soluzioni in rete, prova a ragionarci da solo: gli esercizi che seguono sono pensati per aiutarti a mettere alla prova la tua comprensione, la tua capacità di sviluppare una prova di correttezza e costruire un algoritmo corretto.
Solo dopo averci riflettuto, confronta la tua soluzione con altre possibili versioni.

Esercizi

1. Sviluppare un algoritmo che, dato un grafo G connesso e due suoi nodi u e v, in tempo $O(n+m)$ trova i nodi che hanno la stessa distanza da u e v.

2. Dimostrare che gli archi di G non presenti in un albero BFS di G sono sempre archi trasversali, più precisamente archi che connettono nodi tali che:
   - nessuno dei due è antenato dell'altro e si trovano nello stesso livello o in due livelli adiacenti.

3. Che caratteristiche deve avere un grafo connesso perché le visite DFS e BFS del grafo producano alberi di visita uguali?

Esercizi

4 Un campo minato è rappresentato da una matrice binaria M di dimensione $a \times b$, dove:

- $M[i][j] = 0$ indica una zona bonificata (attraversabile).
- $M[i][j] = 1$ indica una zona ancora minata (non attraversabile).

Un robottino parte dalla cella $M[0][0]$ e deve raggiungere la cella $M[a-1][b-1]$, muovendosi solo in orizzontale o verticale tra zone bonificate.

Sapendo che la cella di partenza e quella d'arrivo sono bonificate, sviluppare un algoritmo che, data la matrice M, in tempo $O(a \cdot b)$ determini il numero minimo di zone bonificate da attraversare per raggiungere la destinazione, esclusa la zona di partenza. Se non esiste un percorso possibile, l'algoritmo deve restituire $+\infty$.

Ad Esempio, per la matrice

$$M = \begin{matrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{matrix}$$

la risposta deve essere 9:

$$\begin{matrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{matrix}$$

Esercizi

5 Data una matrice M binaria quadrata di lato n, si vuole calcolare una matrice D quadrata di lato n tale che, per ogni posizione (i, j) con $0 \leq i, j < n$:

$$D[i][j] = \begin{cases} 0 & \text{se } M[i][j] = 0 \\ \min_{(x,y) \text{ con } M[x][y]=0} (|i - x| + |j - y|) & \text{se } M[i][j] = 1 \end{cases}$$

In altre parole, per ogni cella contenente 1, si vuole determinare la distanza minima da una qualsiasi cella contenente 0, considerando come distanza il numero minimo di passi orizzontali o verticali necessari.

Vincoli:

- La matrice contiene almeno una cella con valore 0.
- I movimenti ammessi sono verso le 4 direzioni cardinali: alto, basso, sinistra, destra.

Input: M

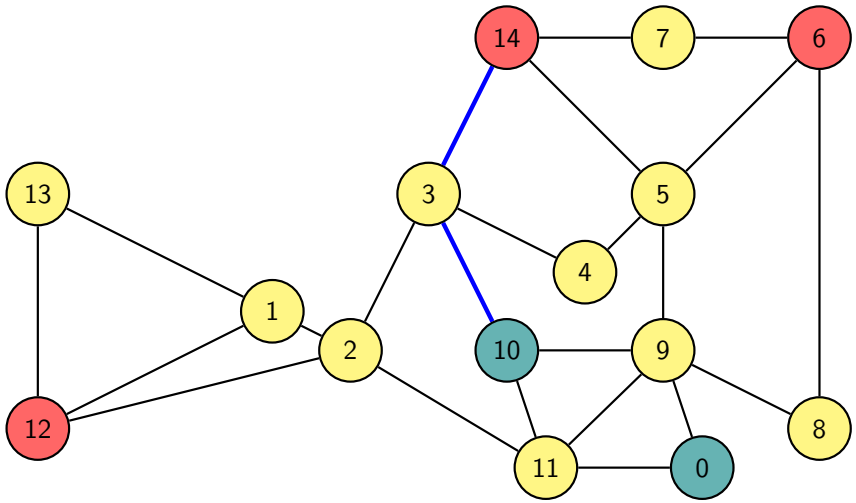
0	1	1
1	1	1
1	1	0

Output: D

0	1	2
1	2	1
2	1	0

Esercizi

6 La **distanza tra due sottoinsiemi disgiunti di nodi** V_1 e V_2 di un grafo connesso G è definita come la lunghezza del cammino minimo che collega un nodo qualsiasi dell'insieme V_1 ad un nodo qualsiasi dell'insieme V_2.
Ad esempio, considera il grafo G in figura con $V_1 = \{6, 12, 14\}$ (evidenziato in rosso) e $V_2 = \{0, 10\}$ (evidenziato in verde), la distanza tra i due sottoinsiemi è 2 (come evidenziato dal commino in blu).



Sviluppare un algoritmo che, dato un grafo G e due liste V_1 e V_2 coi sottoinsiemi di nodi di G, in tempo $O(n + m)$, calcoli la distanza tra V_1 e V_2.

Esercizi

7 Hai due parole s e t di uguale lunghezza l, e il tuo obiettivo è determinare il numero minimo di trasformazioni necessarie per passare dalla parola di partenza s alla parola di destinazione t, cambiando una lettera alla volta. Ogni parola intermedia deve essere una parola valida presente in una lista A contenente n parole di lunghezza l e comprendente s e t.
Progettare un algoritmo che, date le due parole s e t e la lista A, in tempo $O(l \cdot n^2)$, restituisca la sequenza di trasformazioni effettuate per trasformare s in t o una lista vuota se la trasformazione non è possibile.
Ad esempio: per

- $s =' cane'$,
- $t =' data'$
- $A =$
  $['dote',' rata',' cave',' data',' cate',' rapa',' cane',' core',' rate',' cose']$

l'algoritmo deve restituire $['cane',' cate',' rate',' rata',' data']$.

Esercizi

8 I' 8-puzzle è un gioco di scorrimento che consiste in una matrice 3×3 con 8 numeri e una casella vuota. L'obiettivo è muovere i numeri in modo da arrivare a una configurazione finale predefinita, partendo da una configurazione iniziale. Le mosse consentite sono spostare un numero nella casella vuota. Il numero deve essere adiacente alla casella vuota e le direzioni si spostamento ammesse sono: su, giù, sinistra e destra.
La configurazione iniziale è:

$$\begin{bmatrix} - & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{bmatrix}$$

Sviluppare un algoritmo che, data la configurazione finale sotto forma di permutazione dei 9 numeri da 0 a 8 dove lo zero rappresenta la casella vuota (ad esempio la configurazione iniziale è $(0, 1, 2, 3, 4, 5, 6, 7, 8)$):

1. Determini il numero minimo di mosse necessarie per portare il puzzle dalla configurazione iniziale a quella finale. L'algoritmo restituisce -1 se la configurazione finale è irraggiungibile.

2. Determini qual è il numero massimo di mosse necessarie per raggiungere una qualunque delle configurazioni raggiungibili.

Suggerimento: Costruire un grafo i cui nodi sono le possibili configurazioni del puzzle.