La visita in profondità (DFS)

Algoritmi 2 – Lezione 3

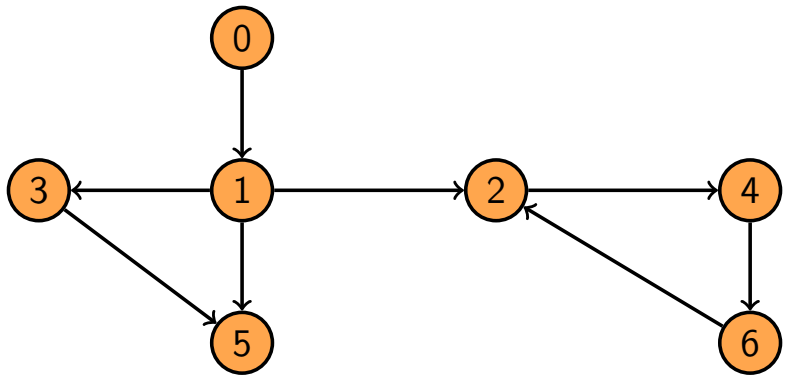
A. Monti
Sapienza Università di Roma

Corso: Algoritmi 2

Indice

Obiettivo: trovare i nodi raggiungibili

- **Obiettivo:** trovare tutti i nodi raggiungibili da un nodo sorgente s in un grafo G



- Per $s = 0$ l'insieme dei nodi raggiungibili è $\{0, 1, 2, 3, 4, 5, 6\}$
- Per $s = 6$ l'insieme dei nodi raggiungibili è $\{2, 4, 6\}$
- Per $s = 5$ l'insieme dei nodi raggiungibili è $\{5\}$

DFS: definizione e analogia

- L' algoritmo DFS segue un cammino profondo finché possibile, tornando indietro quando non trova più vicini non visitati (backtracking).
- E' come percorrere un labirinto: si va sempre avanti in una direzione finché non si trova un vicolo cieco, per poi tornare indietro.

DFS: algoritmo ricorsivo (panoramica)

- Visita il nodo di partenza s.
- Per ogni vicino non ancora visitato, si esegue una chiamata ricorsiva alla DFS su di esso.
- Quando tutti i vicini di un nodo sono già visitati, l'algoritmo fa back-tracking.

DFS Ricorsiva con Vettore Booleano

- Per evitare di visitare più volte lo stesso nodo, si usa un vettore *visitati*.
- La complessità è $O(n+m)$ (G rappresentato tramite liste di adiacenza).

```
def dfs(G, s):
    visitati =[False]* len(G)
    dfsR(G, s, visitati)
    return visitati

def dfsR(G, u, visitati):
    visitati[u] = True
    for v in G[u]:
        if not visitati[v] :
            dfsR(G, v, visitati)
```

Se si desidera ottenere l'elenco dei nodi visitati anziché un vettore booleano, si può restituire:

[i for i in range(len(visitati)) if visitati[i]].

Implementazione con matrice di adiacenza

- L'algoritmo ha complessità $O(n^2)$, poiché per ogni nodo visitato è necessario scorrere l'intera riga della matrice.

```
def dfs_matrice(M, s):
    visitati = [False] *n
    dfs_matriceR(M, s, visitati)
    return visitati

def dfs_matriceR(M, s, visitati):
    n= len(M)
    visitati[s] = True
    for v in range(n):
        # Condizioni per procedere:
        # 1. Esiste un arco da 's' a 'v'
        # 2. Il nodo 'v' non e' stato ancora visitato
        if M[s][v] == 1 and not visitati[v]:
            dfs_matriceR(M, v, visitati)
    return visitati
```

Implementazione con set

- Per tenere traccia dei nodi visitati si può usare un *set*.
- Le operazioni *add* e *in* su un set hanno un costo ammortizzato $O(1)$, rendendo l'algoritmo $O(n+m)$ in tempo medio. Il costo al caso pessimo è $O(n^2 + nm)$.

```
def dfs_set(G, s):
    visitati= set()
    dfs_setR(G,s, visitati)
    return visitati

def dfs_setR(G, u, visitati):
    visitati.add(u)
    for v in G[u]:
        if v not in visitati:
            dfs_setR(G, v, visitati)
```

Questo approccio è utile quando i nodi non sono rappresentati da interi, ad esempio se usano etichette stringa.

DFS iterativa: evitare il limite di ricorsione

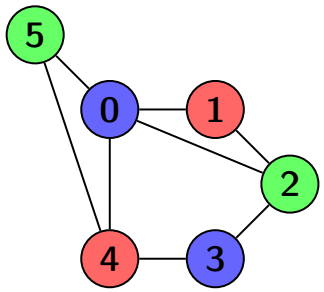
- Python ha un limite di ricorsione di default (1000) che può causare un 'RecursionError' per grafi molto grandi.
- Per evitare il limite di ricorsione, si può usare una versione iterativa basata su uno stack esplicito.

```
def dfs_iterativa(G, s):
    visitati = [False]*len(G)
    stack = [s]
    while stack:
        u = stack.pop()
        if not visitati[u]:
            visitati[u] = True
            for v in G[u]:
                if not visitati[v]:
                    stack.append(v)
    return visitati
```

Il problema della k-colorazione nei grafi

- Dato un intero k, ed un grafo G il problema consiste nel determinare se è possibile assegnare un "colore" a ogni vertice del grafo in modo che due vertici adiacenti abbiano sempre colori diversi.

Di seguito in figura un esempio di grafo 3-colorabile.



- Il problema della k-colorazione per $k \geq 3$ appare computazionalmente difficile mentre il caso $k = 2$ può essere risolto in tempo lineare.

Proprietà della 2-Colorazione

- Un grafo è 2-colorabile se e solo se non contiene cicli di lunghezza dispari.
  - $\Longrightarrow$ Un grafo 2-colorabile non può contenere cicli di lunghezza dispari: due vertici connessi da un ciclo dispari avrebbero lo stesso colore.
  - $\Longleftarrow$ Se un grafo non ha cicli dispari, è possibile colorarlo con due colori alternandoli lungo una DFS.

Algoritmo DFS per la 2-colorazione

- L'algoritmo usa una DFS per colorare i nodi alternando i colori 0 e 1.
- Se si trova un vicino già colorato che ha lo stesso colore del nodo corrente, significa che si è trovato un ciclo dispari e l'algoritmo fallisce.
- La complessità è quella di una visita DFS, quindi $O(n + m)$.

Implementazione: algoritmo DFS per la 2-colorazione

```
def Bicolorazione(G):
    Colore = [ -1]* len(G)
    if DFSr(0, G, Colore, 0):
        return Colore
    return [ ]

def DFSr(x, G, Colore, c):
  Colore[x] = c
  for y in G[x]:
    if Colore[y] == -1:
      if not DFSr(y, G, Colore, 1-c):
        return False
    elif Colore[y] == Colore[x]:
      return False
  return True
```

L'algoritmo può essere adattato per verificare la bipartizione di grafi non connessi, ripetendo la DFS per ogni componente.

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. **Visita in Profondità (DFS)** Sviluppa un algoritmo che, dato un grafo G tramite liste di adiacenza ed un nodo sorgente s, restituisce la lista dei nodi raggiunti a partire da s dalla visita DFS, nell'ordine in cui sono stati visitati, in tempo $O(n+m)$.

2. **Centri e Cammini in un Albero**. In un grafo connesso G **l'eccentricità** di un nodo è la sua massima distanza da tutti gli altri nodi del grafo. Un nodo di G è un **centro** se ha l'eccentricità minima.
   1. Dimostra che esistono grafi in cui tutti i nodi sono centri.
   2. Fornisci un esempio di un albero G che ha esattamente un solo centro.
   3. Fornisci un esempio di un albero G che ha esattamente due centri.
   4. Sia P un cammino di lunghezza massima in un albero G. Dimostra che i centri di G appartengono a questo cammino P.
   5. Dimostra che un albero non può avere 3 centri.
   6. Sviluppa un algoritmo che, dato un albero G di n nodi rappresentato con liste di adiacenza, trova i centri di G n tempo $O(n^2)$.
   7. Sviluppa un algoritmo più efficiente che, dato un albero G di n nodi rappresentato con liste di adiacenza, trova i centri di G in tempo $O(n)$.

Esercizi

3. **Distanza Massima in un Grafo**. Dato un grafo G, rappresentato tramite liste di adiacenza ed un suo nodo s, l'obiettivo è determinare la **massima distanza** di un nodo da s, cioè il numero massimo di archi che è necessario attraversare per raggiungere un qualsiasi nodo del grafo partendo da s.
   - Modifica la procedura di visita in profondità (DFS) per risolvere il problema in tempo $O(n + m)$.

4. **Grafi aciclici**
   1. Dimostra che un grafo aciclico ha al più $n - 1$ archi.
   2. Sviluppa un algoritmo che, dato un grafo G rappresentato con liste di adiacenza, verifica se G è aciclico in tempo $O(n)$.
   3. Sviluppa un algoritmo che, dato un grafo G tramite liste di adiacenza, verifica se G è un albero in tempo $O(n)$.

Esercizi

1. **Grafo Complementare** Dato un grafo G il **grafo complementare** G^c è un grafo che ha gli stessi nodi di G e per archi tutte le coppie che non sono connesse in G
   1. Dimostra che la costruzione di G^c a partire da G richiede un tempo $\Omega(n^2)$.
   2. Sviluppa un algoritmo che, dato il grafo G con liste di adiacenza, restituisce il grafo complementare in tempo $O(n^2)$.

2. Sviluppare un algoritmo che, dato un grafo connesso e aciclico A verifica se esiste in A un nodo che lo tripartisce in modo bilanciato, vale a dire un nodo la cui cancellazione crea tre grafi connessi aciclici disgiunti ciascuno con lo stesso numero di nodi.