Applicazioni della visita in profondità – Parte I

Algoritmi 2 – Lezione 4

A. Monti
Sapienza Università di Roma

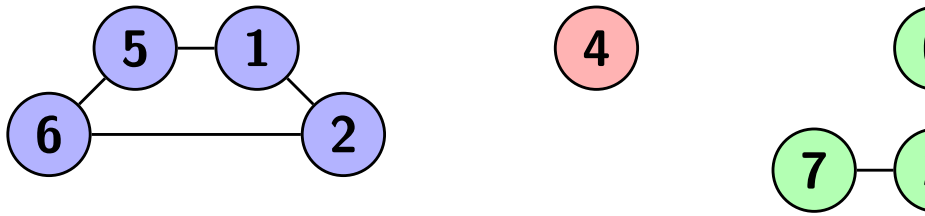
Corso: Algoritmi 2

Indice

Definizione di componente connessa

Dato un grafo G:

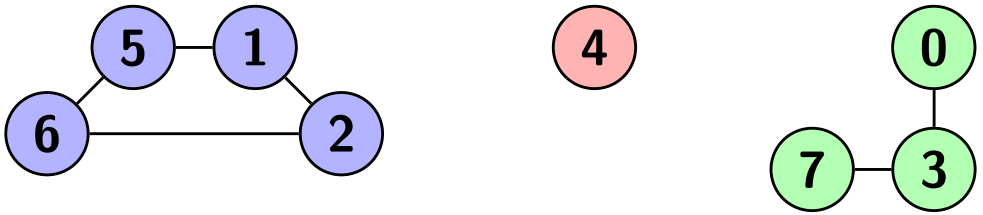
- Un **cammino** è una sequenza di nodi $v_1, v_2, \ldots v_k$ dove (v_i, v_{i+1}) è un arco del grafo.
- Due vertici u e v in un grafo si dicono **connessi** se esiste un cammino che li collega.
- Una **componente connessa** è un insieme massimo di nodi connessi tra loro.
- Un grafo è **connesso** se ha una sola componente connessa, altrimenti è **disconnesso**.



In figura un grafo disconnesso composto da tre componenti connesse.

Vettore delle componenti connesse

- Le componenti connesse partizionano i nodi del grafo.
- Per sapere rapidamente a quale componente appartiene ogni nodo, si usa un **vettore delle componenti connesse** CC.
- Il vettore CC ha lunghezza n, pari al numero di nodi del grafo.
- Per ogni nodo i, $CC[i]$ memorizza un identificatore unico della sua componente.



Un possibile vettore delle tre componenti connesse per il grafo in figura è $CC = [1, 2, 2, 1, 3, 2, 2, 1]$

Algoritmo per il vettore delle componenti

- L'algoritmo sfrutta la visita DFS per identificare i nodi di ogni componente.
- Si itera su tutti i nodi:
  - se un nodo non è ancora stato visitato (ID $= 0$), si avvia una DFS da quel nodo, assegnando un nuovo ID a tutti i nodi raggiungibili.
- La complessità è $O(n+m)$ perché ogni vertice e ogni arco sono visitati una sola volta in totale.

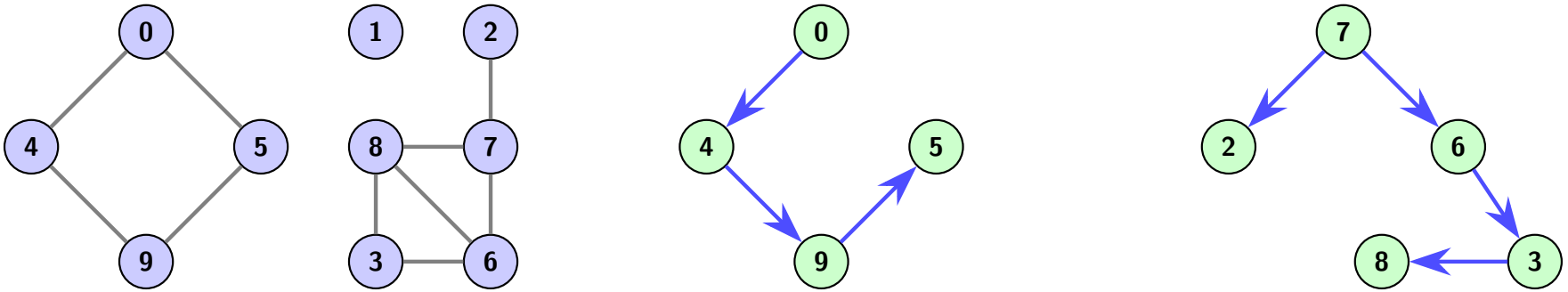
Implementazione: algoritmo per il vettore CC delle componenti connesse

```
def cc(G):
    n = len(G)
    CC = [0] * n
    c = 0
    for u in range(n):
        if CC[u] == 0:
            c += 1
            cc_dfs(G, u, CC, c)
    return CC


def cc_dfs(G, u, CC, c):
    CC[u] = c
    for v in G[u]:
        if CC[v] == 0:
            cc_dfs(G, v, CC, c)
```

Albero DFS e struttura della visita

- La DFS costruisce implicitamente una struttura chiamata **albero DFS**.
- Questo albero è il sottografo aciclico connesso e orientato del grafo originale, composto dai nodi raggiunti e dagli archi effettivamente "attraversati" nel corso della visita per raggiungere nuovi nodi.



(a) Grafo Originale G (b) Albero DFS con radice 0 (c) Albero DFS con radice 7

Il grafo in figura ha due componenti connesse. La DFS da un nodo esplora solo la componente che contiene quel nodo.

Rappresentazione tramite Vettore dei Padri

- Grazie all'albero DFS possiamo ricostruire facilmente il cammino dalla radice a ogni nodo visitato.
- Un modo compatto per rappresentare un albero DFS è tramite un **vettore dei padri** P.
  - Per ogni nodo i, $P[i]$ memorizza l'indice del suo nodo genitore.
  - La radice dell'albero avrà un valore speciale (ad esempio, la radice stessa), mentre i nodi non visitati avranno un valore predefinito (es. -1).

Ecco ad esempio il vettore dei padri ottenuto a partire dalla visita del nodo 0, che esplora la componente connessa contenente i nodi $\{0, 4, 5, 9\}$ del grafo prima visto:

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	-1	-1	-1	0	9	-1	-1	-1	4

Algoritmo per Generare il Vettore P in $O(n+m)$

- L'algoritmo usa una DFS modificata per tracciare i padri di ogni nodo.
- La complessità è $O(n+m)$.

```
def VettorePadri(G, s):
    P = [-1] * len(G)
    P[s] = s
    dfsPadri(G, s, P)
    return P


def dfsPadri(G, u, P):
    for v in G[u]:
        if P[v] == -1:
            P[v] = u
            dfsPadri(G, v, P)
```

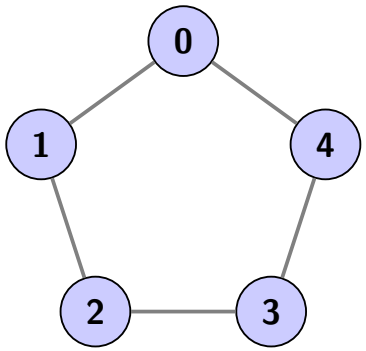
Come Ottenere i cammini in $O(n)$

Per ottenere il cammino da un nodo x alla radice s, risaliamo semplicemente il vettore dei padri. Partendo da x, aggiungiamo il nodo a una lista e ci spostiamo al suo genitore. Ripetiamo questo processo finché non raggiungiamo la radice. Poiché il percorso è costruito al contrario (da x a s), prima di restituirlo lo invertiamo.

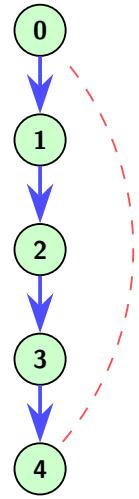
```
def Cammino(x, P):
    """
    Recupera il cammino dalla radice s al nodo x
    utilizzando il vettore dei padri P.
    Restituisce una lista vuota se il cammino non esiste.
    """
    C = [ ]
    if P[x] == -1:
        return C
    # Risali dal nodo x verso la radice
    while P[x] != x:
        # Continua finché non raggiungi la radice
        C.append(x)
        x = P[x]
    # Aggiungi la radice al cammino
    C.append(x)
    # Poiché il cammino è costruito al contrario (da x a s),
    # va invertito prima di restituirlo.
    C.reverse()
    return C
```

I limiti della DFS nei cammini minimi

- Il cammino ricostruito dall'albero DFS non è necessariamente il cammino più breve (in termini di numero di archi).
- Questo accade perché la DFS esplora in profondità e l'albero risultante dipende dall'ordine di visita dei vicini, non dalla loro distanza dalla radice.
- Un esempio classico è un grafo ciclico: la DFS può trovare un cammino lungo, mentre un cammino diretto potrebbe esistere.



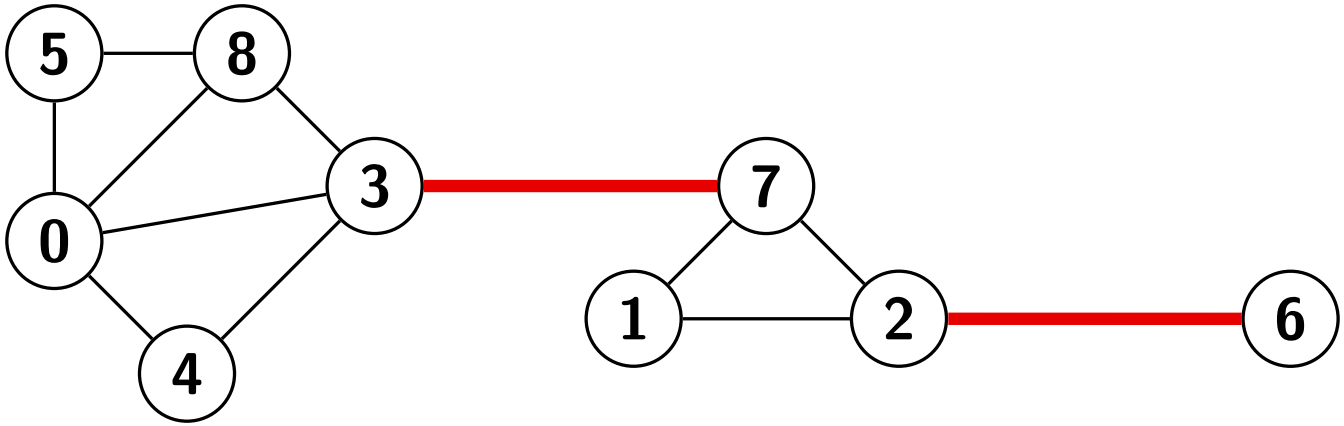
(a) Grafo Originale



(b) Albero DFS di G con radice 0

Definizione di Ponte

- In un grafo connesso, un **ponte** è un arco la cui rimozione sconnette il grafo.



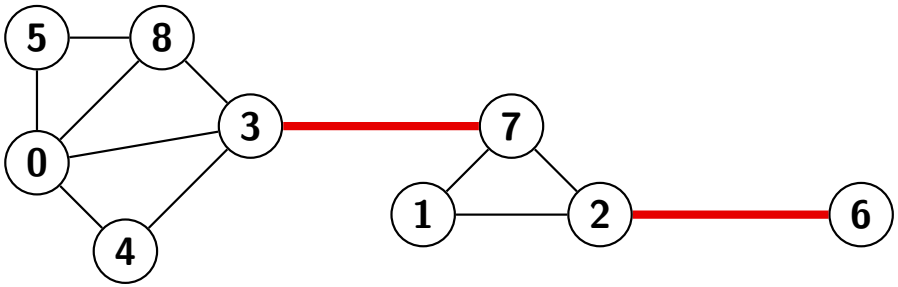
- I ponti rappresentano punti di vulnerabilità in una rete, come un collegamento singolo la cui interruzione causerebbe la separazione della rete.

Ricerca esaustiva dei ponti

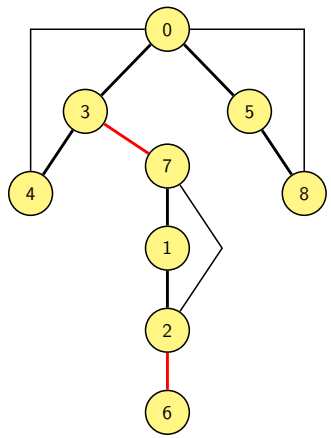
- Un approccio semplice consiste nel rimuovere ogni arco e verificare se il grafo rimane connesso.
- Per ogni arco (u, v), si esegue una DFS da u escludendo l'arco (u, v). Se v non viene raggiunto, l'arco è un ponte.
- Questo algoritmo, sebbene corretto nella sua logica, è estremamente inefficiente. La complessità temporale totale $O(m(n + m))$ può essere proibitiva per grafi di dimensioni significative.

Ricerca Efficiente dei Ponti in $O(n + m)$

- L'approccio efficiente si basa sulla relazione tra i ponti e l'albero DFS.
- Gli archi appartenenti a cicli non possono essere ponti.
- Un arco (u, v) è un ponte se fa parte dell'albero DFS e non è 'coperto' da alcun arco non appartenente all'albero. Un tale arco non dell'albero collegherebbe u (o un suo antenato) a v (o un suo discendente).



(a) Grafo G



(b) DFS del grafo G

A sinistra un grafo G e a destra l'albero DFS ottenuto da una visita a partire dal nodo 0. Gli archi dell'albero sono di colore nero se coperti da archi all'indietro, evidenziati in rosso in caso contrario.

Ricerca Efficiente dei Ponti in $O(n + m)$

- Possiamo dunque cercare i ponti del grafo solo tra gli archi dell'albero DFS che non risultano "coperti" da altri archi .
- Per implementare questa logica si usano due valori per ogni nodo u:
  - $TempoVisita[u]$: il **tempo di scoperta**, ovvero il momento in cui u viene visitato per la prima volta.
  - $Low[u]$: il minimo tempo di scoperta di un nodo che può essere raggiunto
    - partendo da u,
    - scendendo lungo zero o più archi dell'albero DFS (cioè passando eventualmente per i suoi discendenti),
    - e usando al più un arco all'indietro.

    In altre parole, $Low[u]$ rappresenta il più antico antenato dell'albero DFS che è raggiungibile dal sottoalbero radicato in u tramite un arco non dell'albero.

Condizione ponti:

- Un arco (u, v) dell'albero è un ponte se $Low[v] > TempoVisita[u]$.
- Questo significa che non esiste un cammino alternativo (usando un arco non dell'albero) dal sottoalbero di v a un antenato di u.

Algoritmo in O(n+m)

- L'algoritmo esegue una singola DFS, calcolando i valori 'Disc' e 'Low' per ogni nodo.
- La complessità è $O(n+m)$ perché ogni nodo e ogni arco sono visitati al massimo una volta.

```
def cerca_ponti(G):
    n = len(G)
    TempoVisita = [-1] * n
    Low = [-1] * n
    Ponti = []
    time = [0]
    dfs_ponti(0, G, TempoVisita, Low, -1, time, Ponti)
    return Ponti


def dfs_ponti(u, G, TempoVisita, Low, padre, time, Ponti):
    # scoperta del nodo
    TempoVisita[u] = Low[u] = time[0]
    time[0] += 1
    for v in G[u]:
        if v == padre:
            continue
        # gestione di un arco all'indietro
        if TempoVisita[v] != -1:
            Low[u] = min(Low[u], TempoVisita[v])
        else:
            dfs_ponti(v, G, TempoVisita, Low, u, time, Ponti)
            # controllo del ponte dopo la DFS del figlio
            if Low[v] > TempoVisita[u]:
                Ponti.append((u, v))
            Low[u] = min(Low[u], Low[v])
```

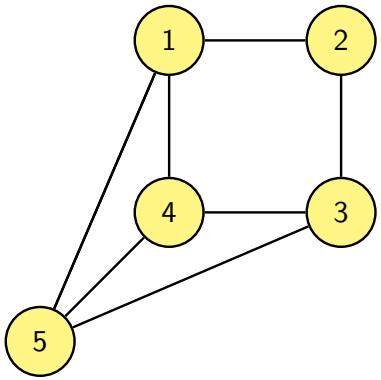
ESERCIZI

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 connesso G tramite liste di adiacenza, in tempo $O(m)$, restituisca un cammino che attraversa tutti gli archi di G una e una sola volta in entrambe le direzioni, (i nodi possono essere toccati anche più volte).
Ad esempio per il grafo in figura che ha 7 archi una possibile soluzione è il seguente cammino lungo 14:

$$1 - 4 - 5 - 4 - 1 - 5 - 1 - 2 - 3 - 4 - 3 - 5 - 3 - 2 - 1$$



2 Dimostrare che il numero di nodi di grado dispari in G è almeno il numero di componenti connesse di G.

Esercizi

3 **Conta le isole.** Immagina di avere una fotografia aerea di un arcipelago. Questa fotografia è rappresentata da una **matrice binaria (griglia 2D)**, dove ogni cella contiene un valore:

- 1 se la cella rappresenta una porzione di **terraferma** (un pezzo di isola).
- 0 se la cella rappresenta una porzione di **mare**.

Due pezzi di terraferma (1) fanno parte della stessa isola se sono **adiacenti orizzontalmente o verticalmente**. Le adiacenze diagonali *non* contano. Il vostro compito è implementare un algoritmo per **contare il numero totale di isole** presenti nell'arcipelago.
L'algoritmo prende dunque in input una matrice $n \times n$ che rappresenta la foto e deve restituire, in tempo $O(n^2)$ il numero di isole dell'arcipelago.
Ad esempio per

$$M = \begin{array}{|c|c|c|c|c|} \hline 1 & 1 & 0 & 1 & 0 \\ \hline 1 & 1 & 0 & 0 & 0 \\ \hline 0 & 0 & 1 & 0 & 0 \\ \hline 0 & 0 & 0 & 1 & 1 \\ \hline 0 & 0 & 1 & 1 & 1 \\ \hline \end{array}$$

l'algoritmo deve restituire 4.

Esercizi

4 **progetti condivisi**. Immaginate di dover organizzare gruppi di studio per un corso universitario. Avete a studenti cui sono stati proposti b progetti differenti identificati dai numeri che vanno da 1 a b. Ogni studente ha selezionato da 1 a 5 di questi progetti. Ricevete una lista di liste L, ogni sottolista $L[i]$ contiene gli identificativi dei progetti , a cui lo studente i sta lavorando. Due studenti devono far parte dello stesso gruppo di studio se condividono almeno un progetto, o se sono collegati indirettamente tramite una catena di studenti che condividono progetti.

Il vostro obiettivo è determinare quante "comunità" di studio indipendenti esistono.

Dovete sviluppare due procedure, entrambe prendono in input la lista L, ma e

1. la prima deve risolvere il problema in tempo $O(a^2)$
2. la seconda deve risolvere risolverlo in tempo $O(a + b)$.

In quali casi la prima procedura è preferibile alla seconda?

Esercizi

5 In un **grafo connesso** G un **punto di articolazione** è un nodo la cui rimozione sconnette il grafo.

1. Dimostrare o confutare che nel grafo G se (u, v) è un ponte allora almeno uno dei due nodi u e v è un punto di articolazione.
2. Dimostrare che in un grafo con $n \geq 2$ nodi il numero di punti di articolazione può andare da 0 a $n - 2$.
3. Sviluppare un algoritmo che, dato il grafo G ed un suo nodo u, in tempo $O(m)$, verifichi se u è un punto di articolazione di G.
4. Sviluppare un algoritmo che, dato il grafo G, in tempo $O(n \cdot m)$, restituisca la lista dei punti di articolazione di G.
5. Dimostrare che il nodo radice dell'albero DFS di G è un punto di articolazione di G se e solo se ha almeno due figli nell'albero.
6. Dimostrare che un nodo u dell'albero DFS di G diverso dalla radice è un punto di articolazione di G se e solo se non esiste in G un arco all'indietro (a, b) tale che nell'albero DFS il nodo a è un discendente di x (o x stesso) e b un antenato di x (o x stesso).
7. Sviluppare un algoritmo che, dato il grafo G, in tempo $O(m)$, restituisca la lista dei punti di articolazione di G.