

Algoritmi (modulo di laboratorio)

Corso di Laurea in Matematica

Roberto Cordone

DI - Università degli Studi di Milano



- Lezioni: Martedì 8.30 - 10.30 in aula 8 Mercoledì 10.30 - 13.30 in aula 309
Giovedì 16.30 - 18.30 in aula 307 Venerdì 10.30 - 12.30 in aula 4
- Ricevimento: su appuntamento (Dipartimento di Informatica)
- E-mail: roberto.cordone@unimi.it
- Pagina web: <http://homes.di.unimi.it/~cordone/courses/2024-algo/2024-algo.html>
- Sito Ariel: <https://mgoldwurmasd.ariel.ctu.unimi.it>

Lezione 11: Visita di grafi e componenti connesse

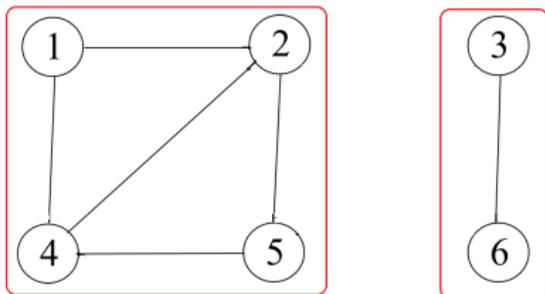
Milano, A.A. 2023/24

Componenti connesse di un grafo

Su un grafo non orientato $G = (V, E)$, la relazione di connessione è

- riflessiva (per convenzione)
- transitiva
- simmetrica

Quindi i vertici raggiungibili formano classi di equivalenza che sono dette **componenti connesse**



Determinare le componenti connesse

Determinare le componenti connesse di un grafo è utile a individuare

- posizioni geografiche mutuamente raggiungibili
- stati di funzionamento di un sistema mutuamente trasformabili
- individui che possono comunicare fra loro
- blocchi di equazioni indipendenti tra loro
- ...

Insomma, è un sottoproblema abbastanza comune

Le principali difficoltà da affrontare nel risolvere questo problema sono

- la presenza di **cammini alternativi** fra gli stessi vertici
- l'esistenza di **cicli**, che tornano indietro a vertici già raggiunti
- l'esistenza di **componenti separate**, per cui non si può scorrere l'intero insieme dei vertici V passando attraverso i lati

Per le componenti fortemente connesse, c'è anche l'**asimmetria degli archi**

Componenti connesse e visita di un grafo

Assumiamo di aver un **algoritmo di visita** dal **vertice sorgente** s , cioè un algoritmo che **enumera i vertici del grafo** (V, E) raggiungibili da s

$$U_s \leftarrow \text{visita}(V, E, s) \quad (\text{con } s \in U_s \subseteq V)$$

Per determinare le componenti connesse basta

- 1 definire un **insieme dei vertici visitati**, **inizialmente vuoto**
- 2 **considerare un vertice v non visitato**
 - visitare il grafo da v : U_v è una **componente connessa**, perché
 - sia C_v la componente connessa (unica) contenente v
 - $U_v \supseteq C_v$ perché i vertici di C_v sono raggiungibili da v
 - $U_v \subseteq C_v$ perché i vertici di U_v sono vicendevolmente raggiungibili passando per v e sfruttando la simmetria dei cammini
- 3 se esistono ancora vertici non visitati, tornare al punto 2

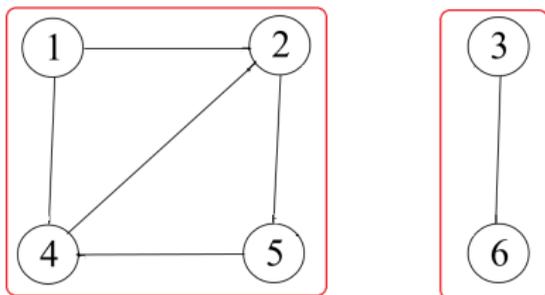
Al termine, i sottoinsiemi U_v ottenuti formano una partizione

Come rappresentare i sottoinsiemi U_v ?

Il **vettore di marcatura C** indica **la componente cui appartiene ogni $v \in V$** (con un indice c intero progressivo, con l'indice s della sorgente, o altro)

- inserire un vertice in una componente richiede $\Theta(1)$
- cercare la componente cui appartiene un vertice richiede $\Theta(1)$

Altre operazioni sono meno efficienti, ma non sono richieste



- tutti i vertici sono marcati come non assegnati: $C = [0 \ 0 \ 0 \ 0 \ 0 \ 0]$
- si parte dalla sorgente $s = 1$
- la visita da $s = 1$ restituisce $U_1 = \{1, 2, 4, 5\}$: $C = [1 \ 1 \ 0 \ 1 \ 1 \ 0]$
- si salta la sorgente $s = 2$, che è già marcata ($C_2 \neq 0$)
- la visita da $s = 3$ restituisce $U_3 = \{3, 6\}$: $C = [1 \ 1 \ 2 \ 1 \ 1 \ 2]$
- si saltano le sorgenti $s = 4, 5$ e 6 , che sono già marcate ($C_4 = C_5 = C_6 \neq 0$)

Componenti connesse di un grafo: pseudocodice

Qui adottiamo la marcatura con l'indice progressivo c

(al termine, fornisce il numero di componenti connesse)

```
ComponentiConnesse(V,E)
{
  for each s in V
    C[s] := 0;

  c := 0;
  for each s in V
  {
    if (C[s] = 0)
    {
      c := c+1;
      visita(V,E,s,C,c);
    }
  }
  return (C,c);
}
```

Visita di un grafo

L'idea fondamentale è di gestire due insiemi di vertici:

- l'insieme U dei **vertici visitati** (cioè raggiunti)
- l'insieme Q dei **vertici visitati, ma non usati per visitarne altri**
(ovviamente, $Q \subseteq U \subseteq V$)

Ogni vertice deve essere visitato e usato una sola volta per visitarne altri

Usarli più volte è

- inutile, perché i vertici adiacenti rimangono gli stessi
- dannoso, perché si cicla indefinitamente sugli stessi vertici

Gli algoritmi di visita

- **partono dalla sorgente**: $U \leftarrow \{s\}$, $Q \leftarrow \{s\}$
- per ogni vertice $v \in Q$ visitato, ma non ancora usato
 - **scorrono l'insieme $\text{Adj}(v)$ dei vertici adiacenti a v**
 - se un vertice adiacente w è **non ancora visitato** (test su U)
 - aggiungono w a U (ora è visitato)
 - aggiungono w a Q (non è ancora stato usato per visitarne altri)

Il grafo come vettore di *forward star* fornisce $\text{Adj}(v)$ efficientemente

Visita di un grafo

Come rappresentare i due sottoinsiemi U e Q ?

Al solito, dipende dalle operazioni richieste dall'algoritmo

L'insieme U dei vertici visitati richiede:

- 1 di aggiungere un elemento nuovo
- 2 di verificare l'appartenenza di un elemento dato

Con un vettore di incidenza entrambe le operazioni richiedono $\Theta(1)$

Per gestire diversi insiemi U , si può usare un vettore di marcatura C

L'insieme Q dei vertici visitati, ma non usati richiede:

- 1 di aggiungere un elemento nuovo
- 2 di estrarre un elemento qualsiasi

Con un vettore di incidenza, l'aggiunta richiede $\Theta(1)$, l'estrazione $\Theta(n)$!

Le liste consentono entrambe le operazioni in $\Theta(1)$
se si limitano l'aggiunta e l'estrazione alle posizioni estreme

Con questi vincoli, è possibile anche risparmiare spazio

Diverse strutture per Q danno luogo a diversi algoritmi di visita

Visita in ampiezza

La **visita in ampiezza** conserva i vertici visitati non usati in una coda Q

```
BFS(V,E,s,C,c)
{
  Q :=  $\emptyset$ ;
  C[s] := c;
  Enqueue(s,Q);
  while not IsEmpty(Q) do
  {
    v := Front(Q);
    Dequeue(Q);
    for each w in Adj(v) do
      if (C[w] = 0)
      {
        C[w] := c;
        Enqueue(w,Q);
      }
  }
}
```

Implementazione delle code come vettori

La **coda** è una **lista** con inserimento dal fondo ed estrazione dalla cima
Si parla di **gestione FIFO** , ovvero **First-In First-Out**
(il primo elemento inserito sarà il primo estratto)

Nel caso della visita di grafi, l'algoritmo ha tre proprietà molto utili

- 1 **la coda non conterrà mai più di $n = |V|$ elementi:**
si può gestire la coda come un vettore V , senza puntatori
- 2 **ogni elemento entra dalla coda ed esce dalla testa:**
si eseguono inserimenti e cancellazioni con due semplici indici interi:
 - **tail** è la **posizione in cui va inserito il prossimo elemento**
(dunque, la prima posizione libera)
 - **head** è la **posizione da cui si estrae il prossimo elemento**
(dunque, la prima posizione occupata)
- 3 **ogni elemento estratto dalla coda non vi rientrerà mai più**

Implementazione delle code come vettori

```
typedef struct _intqueue intqueue;
struct _intqueue
{
    int *V;
    int size; /* dimensione massima della coda */
    int head; /* indice dell'ultima posizione occupata */
    int tail; /* indice della prima posizione libera */
};
```

L'accesso in lettura al primo elemento è banale

```
Front(Q)
{
    Return Q.V[Q.head];
}
```

La coda è vuota quando la posizione in testa è la prima libera

```
IsEmpty(Q)
{
    Return (Q.tail == Q.head);
}
```

Implementazione delle code come vettori

L'inserimento usa la prima posizione libera, che scorre un passo in avanti

```
Enqueue(x, Q)
{
    Q.V[Q.tail] = x;
    Q.tail = (Q.tail+1) % Q.size;
}
```

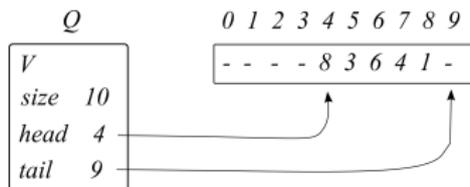
Quando la coda eccede il vettore si usano le celle liberate in testa con l'aritmetica modulo `Q.size` (dimensione allocata)

Lo spazio è esaurito quando `Q.head == Q.tail+1`

La cancellazione sposta un passo in avanti la prima posizione occupata

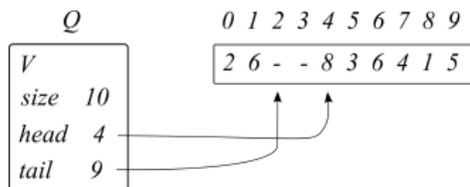
```
Dequeue(Q)
{
    Q.head = (Q.head+1) % Q.size;
}
```

La visita in ampiezza non richiede l'uso dell'aritmetica modulare, perché ogni nodo entra nella coda al massimo una volta

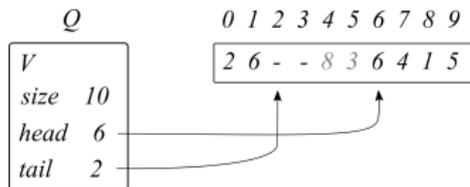


Front(Q) restituisce $Q.V[Q.head]$, cioè 8

IsEmpty(Q) restituisce $(Q.tail == Q.head)$, cioè *false*



Inserimento di tre elementi: Enqueue(5,Q), Enqueue(2,Q), Enqueue(6,Q)



Cancellazione di due elementi: Dequeue(Q), Dequeue(Q)

Visita in profondità (versione iterativa)

La **visita in profondità** conserva i vertici visitati non usati in una pila S

```
DFS(V,E,s,C,c)
{
  S :=  $\emptyset$ ;
  C[s] := c;
  Push(s,S);
  while not IsEmpty(S) do
  {
    v := Top(S);
    Pop(S);
    for each w in Adj(v) do
      if (C[w] = 0)
      {
        C[w] := c;
        Push(w,S);
      }
  }
}
```

```
BFS(V,E,s,C,c)
{
  Q :=  $\emptyset$ ;
  C[s] := c;
  Enqueue(s,Q);
  while not IsEmpty(Q) do
  {
    v := Front(Q);
    Dequeue(Q);
    for each w in Adj(v) do
      if (C[w] = 0)
      {
        C[w] := c;
        Enqueue(w,Q);
      }
  }
}
```

Implementazione delle pile come vettori

La **pila** è una **lista** con **inserimento ed estrazione dalla testa**
Si parla di **gestione LIFO**, ovvero **Last-In First-Out**
(l'ultimo elemento inserito è il primo estratto)

L'algoritmo ha le stesse tre proprietà molto utili della *BFS*:

- 1 **la pila non conterrà mai più di $n = |N|$ elementi:**
si può gestire la pila come un vettore V , senza puntatori
- 2 **ogni elemento entra ed esce dalla cima:**
si eseguono inserimenti e cancellazioni con un semplice indice intero:
 - **top** è la **posizione in cui è stato inserito l'ultimo elemento**
(dunque, l'ultima posizione occupata)
- 3 **ogni elemento estratto dalla pila non vi rientrerà mai più**

Implementazione delle pile come vettori

```
typedef struct _intstack intstack;
struct _intstack
{
    int *V;
    int size; /* dimensione massima della pila */
    int top; /* indice dell'ultima posizione occupata */
};
```

L'accesso in lettura al primo elemento è banale

```
Top(S)
{
    Return S.V[S.top];
}
```

La pila è vuota quando la posizione in cima è fuori del vettore

```
IsEmpty(Q)
{
    Return (S.top == -1);
}
```

Implementazione delle pile come vettori

L'inserimento usa la posizione dopo la cima, che scorre un passo avanti

```
Push(x,S)
{
  S.top = S.top+1;
  S.V[S.top] = x;
}
```

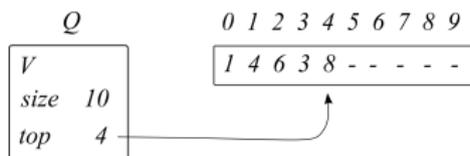
Lo spazio è esaurito quando $S.top == S.size$

La cancellazione sposta un passo indietro l'ultima posizione occupata

```
Pop(S)
{
  S.top = S.top-1;
}
```

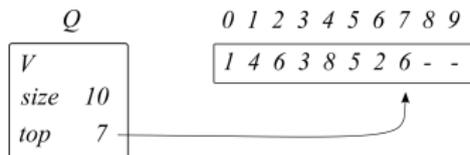
La gestione della pila non richiede l'uso dell'aritmetica modulare, perché inserimenti ed estrazioni avvengono dalla stessa parte del vettore

Esempio

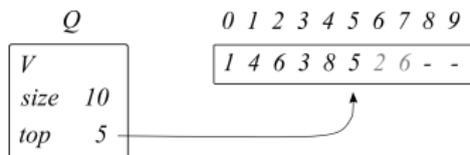


$\text{Top}(S)$ restituisce $S.V[S.\text{top}]$, cioè 8

$\text{IsEmpty}(S)$ restituisce $(S.\text{top} == -1)$, cioè *false*



Inserimento di tre elementi: $\text{Push}(5, S)$, $\text{Push}(2, S)$, $\text{Push}(6, S)$



Cancellazione di due elementi: $\text{Pop}(S)$, $\text{Pop}(S)$

Visita in profondità (versione ricorsiva)

Esiste una **versione ricorsiva** della visita in profondità che **usa la pila di sistema anziché una pila esplicita**

```
DFS(V,E,s,C,c)
```

```
{
```

```
    C[s] := c;
```

```
    for each w in Adj(s) do
```

```
        if (C[w] = 0)
```

```
        {
```

```
            DFS(V,E,w,C,c);
```

```
        }
```

```
}
```

```
DFS(V,E,s,C,c)
```

```
{
```

```
    S :=  $\emptyset$ ;
```

```
    C[s] := c;
```

```
    Push(s,S);
```

```
    while not IsEmpty(S) do
```

```
    {
```

```
        v := Top(S);
```

```
        Pop(S);
```

```
        for each w in Adj(v) do
```

```
            if (C[w] = 0)
```

```
            {
```

```
                C[w] := c;
```

```
                Push(w,S);
```

```
            }
```

```
        }
```

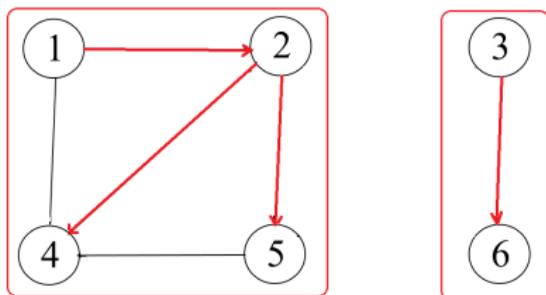
```
}
```

È molto più semplice ed elegante, ma un po' meno efficiente

Alberi di visita

La soluzione è descritta dal vettore di marcatura C , ma è anche possibile annotare per ogni vertice il lato che lo raggiunge e avere l'**albero di visita**

- sottografo del grafo di partenza $G = (V, E)$
- orientato in base all'ordine di visita
- rappresentabile con un vettore f che indica per ogni vertice v il vertice f_v da cui si è raggiunto v (**vertice padre**)



$$f = [1 \ 1 \ 3 \ 2 \ 2 \ 3]$$

Se vi sono più componenti connesse, è una foresta (un albero ciascuna)

Nelle dispense l'albero è descritto dalla lista dei lati, nei codici non è descritto

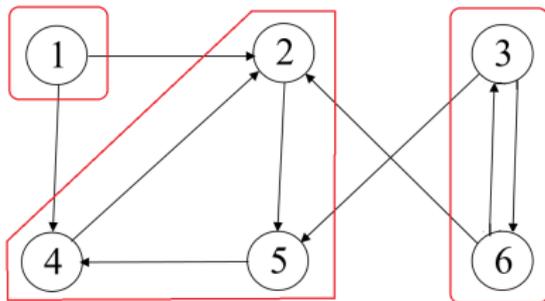
Componenti fortemente connesse di un grafo

Su un grafo orientato $G = (N, A)$, la relazione di connessione è

- riflessiva (per convenzione)
- transitiva
- in generale non simmetrica

La relazione di connessione forte è anche simmetrica (vale nei due versi)

I vertici reciprocamente raggiungibili formano classi di equivalenza che sono dette componenti fortemente connesse



Componenti fortemente connesse e visita di un grafo

Definiamo **nodi co-raggiungibili da v** i **nodi da cui v è raggiungibile**

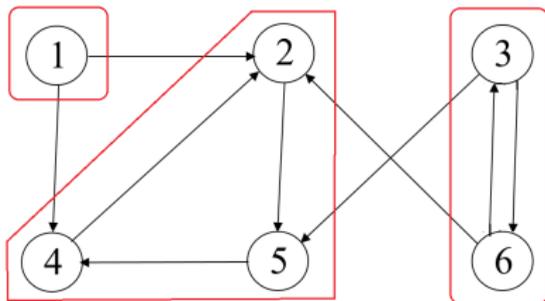
La componente fortemente connessa che contiene v è l'intersezione fra nodi raggiungibili e nodi co-raggiungibili da v :

- tutti i nodi della componente son raggiungibili e co-raggiungibili da v
- tutti i nodi raggiungibili e co-raggiungibili da v sono reciprocamente raggiungibili passando per v , e quindi appartengono alla componente

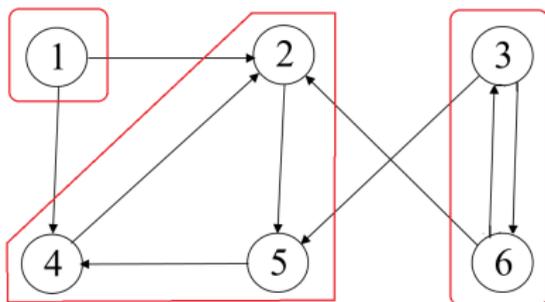
In un grafo orientato, la visita determina solo i nodi raggiungibili da v , ma **invertendo il verso degli archi, la visita determina i nodi co-raggiungibili**

Quindi, **per determinare le componenti fortemente connesse** basta scorrere i nodi come visto prima e per ciascun nodo v

- **visitare il grafo** da v
- **visitare il grafo trasposto** da v (nodi identici e archi invertiti)
(se c'è la *backward star*, basta un solo grafo)
- **calcolare l'intersezione** dei due sottoinsiemi di nodi visitati



- tutti i vertici sono marcati come non assegnati: $C = [0 \ 0 \ 0 \ 0 \ 0 \ 0]$
- si parte dalla sorgente $s = 1$
- la visita diretta da $v = 1$ restituisce $U_1^+ = \{1, 2, 4, 5\}$:
 $C_1 = [1 \ 1 \ 0 \ 1 \ 1 \ 0]$
- la visita inversa da $v = 1$ restituisce $U_1^- = \{1\}$: $C_2 = [1 \ 0 \ 0 \ 0 \ 0 \ 0]$
- l'intersezione è $U_1 = \{1\}$: $C = [1 \ 0 \ 0 \ 0 \ 0 \ 0]$



- la visita diretta da $v = 2$ restituisce $U_2^+ = \{2, 4, 5\}$: $C1 = [0 \ 2 \ 0 \ 2 \ 2 \ 0]$
- la visita inversa da $v = 2$ restituisce $U_2^- = \{1, 2, 3, 4, 5, 6\}$:
 $C2 = [2 \ 2 \ 2 \ 2 \ 2 \ 2]$
- l'intersezione è $U_2 = \{2, 4, 5\}$: $C = [1 \ 2 \ 0 \ 2 \ 2 \ 0]$
- la visita diretta da $v = 3$ restituisce $U_3^+ = \{2, 3, 4, 5, 6\}$:
 $C1 = [0 \ 3 \ 3 \ 3 \ 3 \ 3]$
- la visita inversa da $v = 3$ restituisce $U_3^- = \{3, 6\}$: $C2 = [0 \ 0 \ 3 \ 0 \ 0 \ 3]$
- l'intersezione è il sottoinsieme $U_3 = \{3, 6\}$: $C = [1 \ 2 \ 3 \ 2 \ 2 \ 3]$
- le sorgenti 4, 5 e 6 sono già marcate