Grafi

Anna Corazza

aa 2023/24

Dove studiare

- Sha'13, 11.1 - 11.3 inclusi

Sha'13 Clifford A. Shaffer, Data Structures & Algorithm Analysis in C++, (edition 3.2), 2013
`https://people.cs.vt.edu/shaffer/Book/C++3elatest.pdf`

Richiami della terminologia

Grafi

- Grafo $\mathbf{G} = (\mathbf{V}, \mathbf{E})$; $e \in \mathbf{E}, v_i, v_j \in \mathbf{V}, i \neq j : e = (v_i, v_j)$
- $0 \leq |\mathbf{E}| \leq |\mathbf{V}|^2 - |\mathbf{V}|$:
  - grafo **denso** se $|\mathbf{E}|$ vicino all'estremo superiore
  - grafo **sparso** se vicino all'estremo inferiore.
- Grafo **orientato** se gli archi sono orientati $((v_i, v_j) \neq (v_j, v_i))$; non orientato altrimenti.
- Grafo **etichettato** se i vertici sono etichettati.
- Se esiste l'arco (v_i, v_j), diciamo che v_i, v_j sono **adiacenti** (anche vicini) e che l'arco è **incidente** sui vertici v_i e v_j.
- Nei grafi **pesati** un **peso** (o costo) numerico viene associato ad ogni arco.

Richiami della terminologia
Cammini

- **Cammino**: sequenza di vertici $v_1, v_2, \ldots, v_n$ purché esistano tutti gli archi (v_i, v_{i+1}).
- Un cammino si dice **semplice** se tutti i vertici che lo compongono sono distinti.
- La lunghezza di un cammino è data dal numero di **archi** che lo compongono.
- In un grafo non orientato, se il primo e l'ultimo vertice coincidono e la lunghezza è ≥ 3 si parla di **ciclo**.
- Se il grafo è orientato, la restrizione sulla lunghezza cade.
- Un ciclo si dice semplice se il cammino è semplice eccetto che per il vertice iniziale e finale.

Richiami della terminologia
Sottografi

- Dato un grafo $\mathbf{G} = (\mathbf{V}, \mathbf{E})$n **sottografo** $\mathbf{S} = (\mathbf{V_S}, \mathbf{E_S})$ con $\mathbf{V_S} \subseteq \mathbf{V}$ e $\mathbf{E_S} \subseteq \mathbf{E}$ tali che per ogni arco $(v_i, v_j) \in \mathbf{E_S} : v_i, v_j \in \mathbf{V_S}$.
- Un grafo non orientato si dice **connesso** se per ogni coppia di vertici esiste almeno un cammino che li congiunge.
- I sottografi connessi **massimali** (non c'è un altro sottografo connesso che li contiene) sono detti **componenti connesse** del grafo.
- Un grafo senza cicli è detto **aciclico**.
- **DAG** sta per Directed Acyclic Graph, ovvero grafo aciclico orientato.
- **Albero libero**: grafo non orientato connesso con $|\mathbf{E}| = |\mathbf{V} - 1|$. Non può avere cicli (semplici).

Rappresentazione di un grafo

Matrice di adiacenza

- **Matrice di adiacenza**:
  - Matrice $|\mathbf{V}| \times |\mathbf{V}|$ di bit: 1 se esiste l'arco, 0 altrimenti.
  - Per rappresentare matrici pesate, ogni elemento contiene un numero.
  - In ogni caso, richiede spazio $\Theta(|\mathbf{V}|^2)$.
- **Lista di adiacenza**:
  - Vettore di $|\mathbf{V}|$ liste linkate (o altri contenitori più adeguati): la lista nella posizione i-esima contiene i vertici adiacenti a v_i (e in questo modo rappresenta gli archi).
  - Una posizione nell'array e quindi una lista per ogni vertice (anche se non ha archi incidenti) e una posizione in una lista per ogni arco: richiede spazio $\Theta(|\mathbf{V}| + \mathbf{E})$.
- Entrambe le rappresentazioni sono adatte sia a grafi orientati che a grafi non orientati.
- Un arco senza direzione è equivalente ai due archi con direzione corrispondenti.

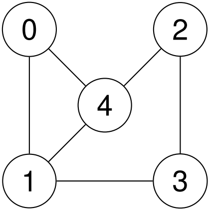
Richieste in termini di spazio

- Quale delle due rappresentazioni conviene in termini di spazio richiesto dipende dal numero di archi e quindi da quanto è denso il grafo.
- La lista di adiacenza usa spazio solo per gli archi che effettivamente appaiono nel grafo, mentre la matrice di adiacenza richiede lo stesso spazio per tutti i potenziali archi, sia che ci siano che non ci siano.
- D'altra parte, la matrice di adiacenza non richiede spazio aggiuntivo per i puntatori.
- Più il grafo è denso, più conviene la matrice di adiacenza.
- Più il grafo è sparso, più conviene la lista di adiacenza.

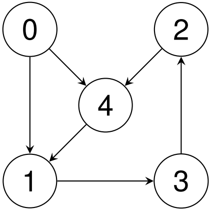
Spazio richiesto

Esempio

- Supponiamo di aver bisogno di:
  - due byte per l'indice dei vertici
  - quattro byte per un puntatore
  - due byte per il peso associato ad ogni arco



$4|\mathbf{V}| + 2 \cdot 6|\mathbf{E}| = 92$ byte



$4|\mathbf{V}| + 6|\mathbf{E}| = 56$ byte

Tempo richiesto

- Ovviamente dipende dall'algoritmo.
- Tuttavia molti algoritmi scorrono i nodi adiacenti al nodo considerato.
- Questa operazione risulta immediata con le liste di adiacenza (basta accedere alla lista corrispondente al nodo sotto esame),
- Mentre con la matrice di adiacenza occorre scorrere tutte le posizioni relative a ciascuno dei $|\mathbf{V}|$ vertici.
- Si ottiene quindi un costo di $\Theta(|\mathbf{V}|^2)$ con la matrice di adiacenza invece che $\Theta(|\mathbf{V}| + |\mathbf{E}|)$ con la lista di adiacenza.
- Se il grafo è sparso il vantaggio può essere considerevole.
- Se invece il grafo è denso il numero di archi viene ad essere simile a $|\mathbf{V}|^2$.

Implementazione

Classe astratta **Graph**

- I vertici verranno indicati da valori interi in $[0, |\mathbf{V}| - 1]$.
- Ovviamente in generale avremo dell'informazione associata ad ogni vertice, ma questa viene memorizzata altrove, a cura dell'utente.
- Quindi l'implementazione del grafo non farà uso di template.
- Supporremo che il grafo possa essere pesato.
- In quel caso, il metodo **weight** prende in ingresso due vertici e restituisce il peso dell'arco ad essi associato.

Implementazione
Metodi accessori

- In genere gli algoritmi prevedono di visitare i nodi adiacenti al nodo dato.
- Daremo supporto a questa necessità con due funzioni:

  first (v) restituisce il primo vertice adiacente al nodo v

  next (v,n) restituisce il vertice adiacente a v immediatamente dopo n nella lista dei nodi adiacenti; $n = |V|$ a fine lista.

```
for(w=G->first(v); w < G->n(); w=G->next(v,w))
```

- Le visite del grafo fanno spesso uso di marcatori per i nodi: offriremo supporto a tale operazione.

Visite del grafo

- Come per gli alberi, anche per i grafi esistono delle visite (**traversals**) standard.
- Ogni vertice viene visitato una ed una sola volta.
- Come per gli alberi, già conoscete (ma richiamiamo) l'implementazione ricorsiva, ma andremo a definire degli iteratori.
- Per cominciare la visita si sceglie un vertice di partenza.

Marcatori

- Due problemi principali:
  1. grafo **non connesso**: non è possibile raggiungere tutti i vertici da quello scelto per partire;
  2. presenza di **cicli**, che possono portare, se non controllati, a loop infiniti.
- Entrambi i problemi possono venir risolti con dei marcatori sui vertici (un bit per vertice):
  1. **cicli**: evito di visitare vertici già visitati;
  2. grafo **non connesso**: alla fine controllo per vedere se ci sono vertici ancora non visitati (se ci sono, riparto con la visita da uno di questi).

Implementazione della visita

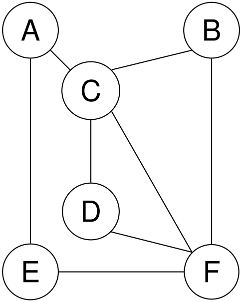
Funziona sia coi grafi orientati che con quelli non orientati.

```
void graphTraverse(Graph* G) {
  int v;
  for (v=0; v<G->n(); v++)
    // Initialize mark bits
    G->setMark(v, UNVISITED);
  for (v=0; v<G->n(); v++)
    if (G->getMark(v) == UNVISITED)
      doTraverse(G, v);
}
```

Dove **doTraverse()** può implementare una visita in ampiezza o in profondità.

Rappresentazione del grafo

Liste di adiacenza



Vertici	Adiacenti
A	C,E
B	C,F
C	A,B,D,F
D	C,F
E	A,F
F	B,E

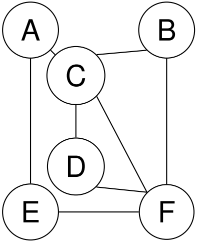
Visita in ampiezza

- **Breath-first search (BFS).**
- Prima di procedere visiti tutti i vertici collegati al vertice corrente.
- Struttura di supporto: **coda**.

```
void BFS(Graph* G, int start, Queue<int>* Q) {
    int v, w;
    Q->enqueue(start);
    // Initialize Q
    G->setMark(start, VISITED);
    while (Q->length() != 0) { // Process all vertices on Q
        v = Q->dequeue();
        PreVisit(G, v);       // Take appropriate action
        for (w=G->first(v); w<G->n(); w = G->next(v,w))
            if (G->getMark(w) == UNVISITED) {
                G->setMark(w, VISITED);
                Q->enqueue(w);
            }
    }
}
```

Visita in ampiezza

Marcatura nodi



Coda	Azioni	Mark
A	BFS(A),mEe(A)	A
C E	deq(A), pr**(A,C)**, mEe(C) pr**(A,E)**, mEe(E)	C E
E B D F	deq(C), pr(C,A), pr**(C,B)**, mEe(B), pr**(C,D)**, mEe(D), pr**(C,F)**, mEe(F)	B D F
B D F	deq(E), pr(E,A), pr(E,F)	
D F	deq(B), pr(B,C), pr(B,F)	
F	deq(D), pr(D,C), pr(D,F)	
	deq(F), pr(F,B), pr(F,C), pr(F,D)	
	end	

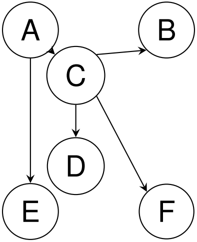
pr sta per *process*

mEe sta per *marca e elabora*

Visita in ampiezza

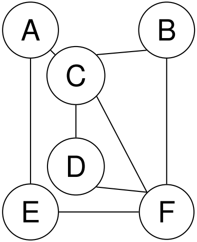
Albero di ricerca

Coda	**Azioni**
A	BFS(A),mEe(A)
C E	deq(A), pr**(A,C)**, mEe(C) pr**(A,E)**, mEe(E)
E B D F	deq(C), pr(C,A), pr**(C,B)**, mEe(B), pr**(C,D)**, mEe(D), pr**(C,F)**, mEe(F)
B D F	deq(E), pr(E,A), pr(E,F)
D F	deq(B), pr(B,C), pr(B,F)
F	deq(D), pr(D,C), pr(D,F)
	deq(F), pr(F,B), pr(F,C), pr(F,D)
	end



Visita in ampiezza

Simulazione

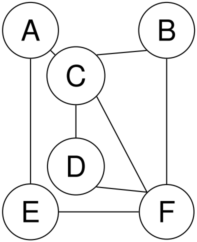


Coda	**Azioni**	**Out**
A	BFS(A),mEe(A)	
C E	deq(A), pr**(A,C)**, mEe(C) pr**(A,E)**, mEe(E)	A
E B D F	deq(C), pr(C,A), pr**(C,B)**, mEe(B), pr**(C,D)**, mEe(D), pr**(C,F)**, mEe(F)	C
B D F	deq(E), pr(E,A), pr(E,F)	E
D F	deq(B), pr(B,C), pr(B,F)	B
F	deq(D), pr(D,C), pr(D,F)	D
	deq(F), pr(F,B), pr(F,C), pr(F,D)	F
	end	

pr sta per *process*
mEe sta per *marca e elabora*

Visita in ampiezza

Dal punto di vista degli iteratori (esterni)



Coda	Azioni
A	enq(A)
C E	deq(A), enq(C), enq(E)
E B D F	deq(C), enq(B), enq(D), enq(F)
B D F	deq(E)
D F	deq(B)
F	deq(D)
	deq(F)
	end

Visita in profondità

- **Ricorsiva**: ogni volta che si visita un vertice v si visitano anche i suoi vicini non ancora visitati.
- **Iteratori**:
  - si inseriscono in uno **stack** tutti gli archi che escono da v;
  - per trovare il prossimo vertice da visitare, si estrae e segue un arco dallo stack.
- L'effetto è di seguire un ramo nel grafo fino alla sua conclusione prima di risalire le biforcazioni.
- Si costruisce così un **albero di ricerca in profondità**.
- Vale sia per i grafi orientati che per quelli non orientati.

Implementazione ricorsiva della ricerca in profondità

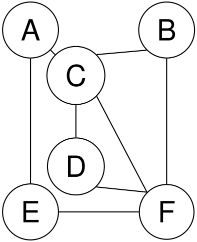
```
void DFS(Graph* G, int v) { // Depth first search
  PreVisit(G, v); // Take appropriate action
  G->setMark(v, VISITED);
  for (int w=G->first(v); w<G->n(); w =
      G->next(v,w))
    if (G->getMark(w) == UNVISITED)
      DFS(G, w);
  PostVisit(G, v); // Take appropriate action
}
```

Previsit Elaborazioni da fare sul nodo prima della visita.

Postvisit Elaborazioni da fare sul nodo dopo la visita.

Visita in profondità

Grafo non orientato



Stack	Azioni
| A	DFS(A)
| A C	m(A), pr**(A,C)**, DFS(C)
| A C B	m(C), pr(C,A), pr**(C,B)**, DFS(B)
| A C B F	m(B), pr(B,C), pr**(B,F)**, DFS(F)
| A C B F D	m(F), pr(F,B), pr(F,C), pr**(F,D)**, DFS(D)
| A C B F	m(D), pr(D,C), pr(D,F),pop(D)
| A C B F E	pr**(F,E)**,DFS(E)
| A C B F	m(E),pr(E,A),pr(E,F),pop(E)
| A C B	pop(F)
| A C	pop(B)
| A	pr(C,E), pr(C,F),pop(C)
|	pr(A,E), pop(A)
	end

Visita in profondità

Tipi di arco

- Tipi di arco:

  Nell'albero di ricerca se il secondo vertice non è ancora stato visitato quando scopro l'arco.

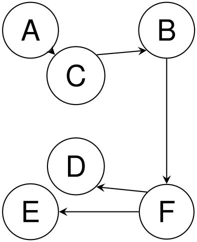
  All'indietro se il secondo vertice già appartiene all'albero di ricerca (ovvero è marcato come visitato): in altre parole, è un antenato.

  In avanti se il secondo vertice è un discendente nell'albero di ricerca.

- Se il grafo è orientato, la presenza di archi all'indietro segnala la presenza di un ciclo.

Visita in profondità

albero di ricerca



V	Adj
A	C,E
B	C,F
C	A,B,D,F
D	C,F
E	A,F
F	B,E

Stack	**Azioni**
| A	DFS(A)
| A C	m(A), pr**(A,C)**, DFS(C)
| A C B	m(C), pr(C,A), pr**(C,B)**, DFS(B)
| A C B F	m(B), pr(B,C), pr**(B,F)**, DFS(F)
| A C B F D	m(F), pr(F,B), pr(F,C), pr**(F,D)**, DFS(D)
| A C B F	m(D), pr(D,C), pr(D,F),pop(D)
| A C B F E	pr**(F,E)**,DFS(E)
| A C B F	m(E),pr(E,A),pr(E,F),pop(E)
| A C B	pop(F)
| A C	pop(B)
| A	pr(C,E), pr(C,F),pop(C)
|	pr(A,E), pop(A)
	end

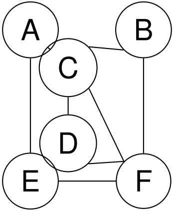
Visita in profondità

Implementazione iterativa

```
void DFS() {
  init(stack,visitedVertices,adjIterators);
  for(int v=vertices.first(); vertices.hasMore();
                                   v=vertices.Next())
    if(!v.isVisited())
      PreVisit(v);
      stack.push(v);
      while(!stack.isEmpty())
        v = stack.top();
        v.markVisited();
        while(adjIterators[v].hasNext() and
           (w = adjIterators[v].Next()).isVisited())
             // do nothing;
        if(!w.isVisited())
           PreVisit(w);
           stack.push(w);
        else
           stack.pop();
           PostVisit(v);
```

Visita in profondità

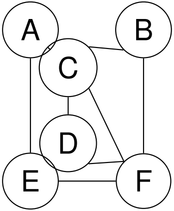
Pre-order



Stack	Azioni	Out
| A	push(A)	A
| A C	m(A), pr(**A,C**), push(C)	C
| A C B	m(C), pr(C,A), pr(**C,B**), push(B)	B
| A C B F	m(B), pr(B,C), pr(**B,F**), push(F)	F
| A C B F D	m(F), pr(F,B), pr(F,C), pr(**F,D**), push(D)	D
| A C B F	m(D), pr(D,C), pr(D,F),pop(D)	
| A C B F E	pr(**F,E**),push(E)	E
| A C B F	m(E),pr(E,A),pr(E,F),pop(E)	
| A C B	pop(F)	
| A C	pop(B)	
| A	pr(C,E), pr(C,F),pop(C)	
|	pr(A,E), pop(A)	
	end	

Visita in profondità

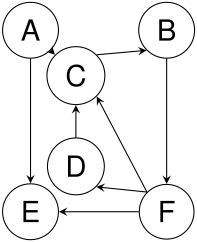
Post-order



<table>
<tr><th>Stack</th><th>Azioni</th><th>Out</th></tr>
<tr><td>| A</td><td>push(A)</td><td></td></tr>
<tr><td>| A C</td><td>m(A), pr(A,C), push(C)</td><td></td></tr>
<tr><td>| A C B</td><td>m(C), pr(C,A), pr(C,B), push(B)</td><td></td></tr>
<tr><td>| A C B F</td><td>m(B), pr(B,C), pr(B,F), push(F)</td><td></td></tr>
<tr><td>| A C B F D</td><td>m(F), pr(F,B), pr(F,C), pr(F,D), push(D)</td><td></td></tr>
<tr><td>| A C B F</td><td>m(D), pr(D,C), pr(D,F),pop(D)</td><td>D</td></tr>
<tr><td>| A C B F E</td><td>pr(F,E),push(E)</td><td></td></tr>
<tr><td>| A C B F</td><td>m(E),pr(E,A),pr(E,F),pop(E)</td><td>E</td></tr>
<tr><td>| A C B</td><td>pop(F)</td><td>F</td></tr>
<tr><td>| A C</td><td>pop(B)</td><td>B</td></tr>
<tr><td>| A</td><td>pr(C,E), pr(C,F),pop(C)</td><td>C</td></tr>
<tr><td>|</td><td>pr(A,E), pop(A)</td><td>A</td></tr>
<tr><td></td><td>end</td><td></td></tr>
</table>

Visita in profondità

Grafo orientato

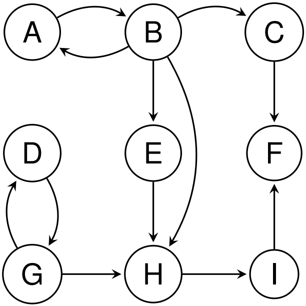


V	Adj
A	C,E
B	F
C	B
D	C
E	
F	C,D,E

Stack	**Azioni**
| A	DFS(A)
| A C	m(A), pr**(A,C)**, DFS(C)
| A C B	m(C), pr**(C,B)**, DFS(B)
| A C B F	m(B), pr**(B,F)**, DFS(F)
| A C B F D	m(F), $\boxed{\text{pr(F,C)}}$, pr**(F,D)**, DFS(D)
| A C B F	m(D), $\boxed{\text{pr(D,C)}}$,pop(D)
| A C B F E	pr**(F,E)**,DFS(E)
| A C B F	m(E),pop(E)
| A C B	pop(F)
| A C	pop(B)
| A	pop(C)
|	pr(A,E), pop(A)
	end

Grafo orientato

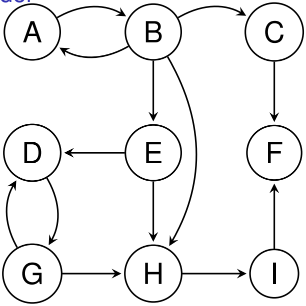
Liste di adiacenza



Vertici	Adiacenti
A	B
B	A,C,E,H
C	F
D	G
E	D,H
F	
G	D,H
H	I
I	F

Visita in profondità

Pre-order

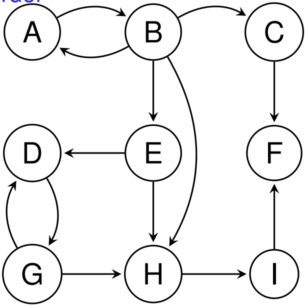


V	Adj
A	B
B	A,C,E,H
C	F
D	G
E	H, D
F	
G	D,H
H	I
I	F

t	Stack	V
0	| A	A
1	| A, B	B
2	| A, B, C	C
3	| A, B, C, F	F
4	| A, B, C	
5	| A, B	
6	| A, B, E	E
7	| A, B, E, H	H
8	| A, B, E, H, I	I
9	| A, B, E, H	
10	| A, B, E	
11	| A, B, E, D	D
12	| A, B, E, D, G	G
13	| A, B, E, H	
14	| A, B, E	
15	| A, B	
16	| A	
17	|	

Visita in profondità

Post-order



V	Adj
A	B
B	A,C,E,H
C	F
D	G
E	H, D
F	
G	D,H
H	I
I	F

t	Stack	V
0	| A	
1	| A, B	
2	| A, B, C	
3	| A, B, C, F	
4	| A, B, C	F
5	| A, B	C
6	| A, B, E	
7	| A, B, E, H	
8	| A, B, E, H, I	
9	| A, B, E, H	I
10	| A, B, E	H
11	| A, B, E, D	
12	| A, B, E, D, G	
13	| A, B, E, H	G
14	| A, B, E	H
15	| A, B	E
16	| A	B
17	|	A

Ordinamento topologico

- Organizzazione di task in un DAG.
- Arco significa che il vertice sorgente deve precedere quello destinazione.
- Si ottiene tramite DFS sul grafo con:

PreVisit	nulla
PostVisit	stampa il nodo

- Si ottiene un ordinamento topologico rovesciato.
- Non importa da quale nodo si parte, basta che sia garantito che tutti i nodi vengono visitati.

Ordinamento topologico

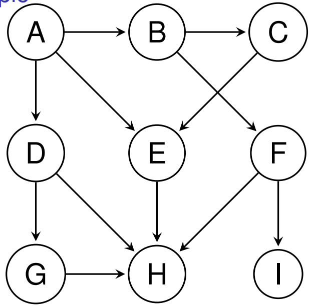
- Implementazione ricorsiva

```
void topsort(Graph* G) {
    for (int i=0; i<G->n(); i++) // Initialize Mark array
        G->setMark(i, UNVISITED);
    for (i=0; i<G->n(); i++) // Process all vertices
        if (G->getMark(i) == UNVISITED)
            tophelp(G, i); // Call recursive helper function
}

void tophelp(Graph* G, int v) { // Process vertex v
    G->setMark(v, VISITED);
    for (int w=G->first(v); w<G->n(); w = G->next(v,w))
        if (G->getMark(w) == UNVISITED)
            tophelp(G, w);
    printout(v); // PostVisit for Vertex v
}
```

Ordinamento topologico

Esempio



V	Adj
A	B, D, E
B	C, F
C	E
D	G, H
E	H
F	H, I
G	H
H	
I	

t	Stack	V
0	| C	
1	| C, E	
2	| C, E, H	
3	| C, E	H
4	| C	E
5	|	C
6	| A	
7	| A, B	
8	| A, B, F	
9	| A, B, F, I	
10	| A, B, F	I
11	| A, B	F
12	| A	B
13	| A, D	
14	| A, D, G	
15	| A, D	G
16	| A	D
17	|	A

Ordinamento topologico

Con l'appoggio di una coda

- Per ogni vertice: ogni arco entrante corrisponde ad un prerequisito.
- Se il grafo ha uno o più cicli, non ci sono soluzioni.
- Implementazione con una coda
- Scorri tutti gli archi e conti per ogni vertice il numero di archi entranti.
- Inserisci nella coda tutti i nodi che non hanno archi entranti.
- Per ogni nodo che estrai dalla coda:
  - lo stampi;
  - scorri i vertici adiacenti e decrementi di uno il relativo conteggio;
  - se ci sono conteggi che sono passati da 1 a 0, li aggiungi alla coda.
- Se la coda si svuota prima di stampare tutti i vertici, allora il problema non ha soluzione (ci sono dei cicli).

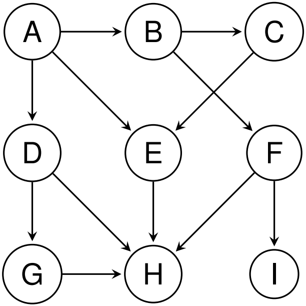
Implementazione

- Per ogni vertice, numero di archi entramti.
- In coda se $==$ 0. Per ogni nodo in coda:
  - stampi;
  - decrementi di uno il conteggio dei vertici adiacenti;
  - aggiungi alla coda se conteggio nullo.

```
void topsort(Graph* G, Queue<int>* Q) {
  int Count[G->n()];
  for (int v=0; v<G->n(); v++) Count[v] = 0; //  Initialize
  for (int v=0; v<G->n(); v++) // Process every edge
    for (int w=G->first(v); w<G->n(); w = G->next(v,w))
      Count[w]++;  // Add to v's prereq count
  for (int v=0; v<G->n(); v++)  //  Initialize  queue
    if (Count[v] == 0)  // Vertex has no prerequisites
      Q->enqueue(v);
  while (Q->length() != 0) {  // Process the vertices
    v = Q->dequeue();
    printout(v);    // PreVisit for "v"
    for (int w=G->first(v); w<G->n(); w = G->next(v,w)) {
      Count[w]--; // One less prerequisite
      if (Count[w] == 0)  // This vertex is now free
        Q->enqueue(w);
    }}}
```

Ordinamento topologico

Esempio: con l'appoggio di una coda



V	Adj	A	B	D	C	F	G	E	I
A	B, D, E	0	-						
B	C, F	1	0						
C	E	1	1	0					
D	G, H	1	0						
E	H	2	1	1	1	0			
F	H, I	1	1	0					
G	H	1	1	1	0				
H		4	4	4	3	3	2	1	0
I		1	1	1	1	1	0		

Coda	**V**
A	A
B, D	B
D, C, F,	D
C, F, G	C
F, G, E	F
G, E, I	G
E, I	E
I, H	I
H	H

Iteratori online e offline

- L'ordine di visita può essere precomputato una volta per tutte quando inizializzo l'iteratore.
- In questo caso, l'ordine viene salvato e poi non si fa che scorrerlo.

Componenti Fortemente Connesse (CFC)

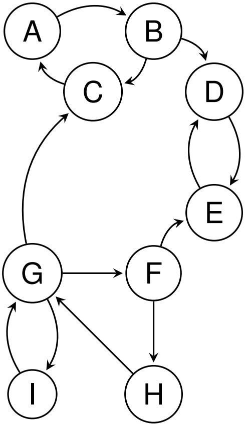
- Dato un grafo orientato $G = (V, E)$, un sottoinsieme massimale C di V: per ogni coppia di vertici $u, v \in C$, v è raggiungibile da u e viceversa.
- Vogliamo trovare **tutte** le CFC in un grafo orientato.
- Grafo trasposto: $G^T = (V, E^T)$: $E^T = \{(u, v) : (v, u) \in E\}$
- Due DFS:
  1. una sul grafo originale per trovare l'ordinamento post-order rovesciato
  2. una sul grafo trasposto, per trovare le CFC
- Il grafo originale e il suo trasposto hanno esattamente le stess CFC, visto che non faccio che rovesciare la direzione di **tutti** gli archi

Costruzione del grafo trasposto

- Trasposta della matrice di adiacenza.
- Per costruire la lista di adiacenza a partire da analoga rappresentazione:
  1. Costruisco l'elenco dei vertici per il grafo trasposto.
  2. Scorro la lista di adiacenza del grafo originale e per ogni edge, inserisco la versione rovesciata nel grafo trasposto.
- In questo caso, nel costruire l'elenco dei vertici per il grafo trasposto seguo l'ordine prodotto dalla visita post-order del grafo originale, ma dopo averlo rovesciato.
- Quindi associo ad ogni vertice il tempo di chiusura e poi li scorro per tempo di chiusura descrescente.
- Ogni volta che la DFS sul grafo rovesciato svuota lo stack, produco una CFC.

Grafo orientato

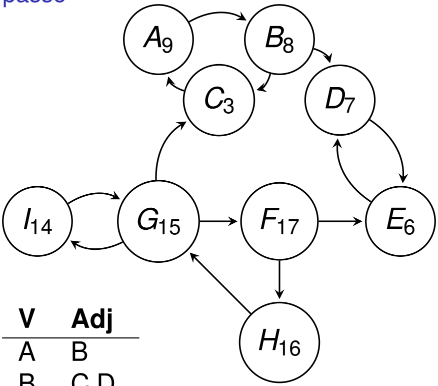
Liste di adiacenza



Vertici	Adiacenti
A	B
B	C,D
C	A
D	E
E	D
F	E,H
G	F,C,I
H	G
I	G

Ricerca CFC

Primo passo



V	Adj
A	B
B	C,D
C	A
D	E
E	D
F	E,H
G	F,C,I
H	G
I	G

	t	Stack	V
	0	| A	
	1	| A, B	
	2	| A, B, C	
	3	| A, B	C
	4	| A, B, D	
	5	| A, B, D, E	
	6	| A, B, D	E
	7	| A, B	D
	8	| A	B
	9	| ϵ	A
	10	| F	
	11	| F, H	
	12	| F, H, G	
	13	| F, H, G, I	
	14	| F, H, G	I
	15	| F, H	G
	16	| F	H
pari ->	17	| ϵ	F

Ricerca CFC

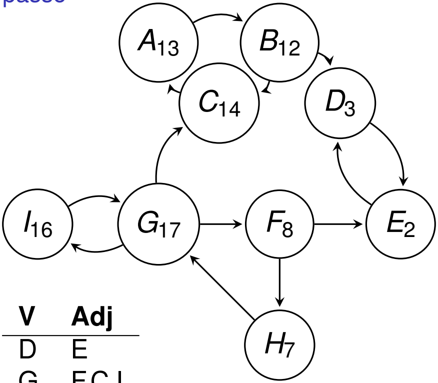
Secondo passo

Grafo originale	
V	**Adj**
A	B
B	C,D
C	A
D	E
E	D
F	E,H
G	F,C,I
H	G
I	G
Grafo trasposto	
F	G
H	F
G	H,I
I	G
A	C
B	A
D	B,E
E	F,D
C	G,B

Stack	V	CC
| F		
| F, G		
| F, G, H		
| F, G	H	
| F, G, I		
| F, G	I	
| F	G	
| ϵ	F	H,I,G,F
| A		
| A, C		
| A, C, B		
| A, C	B	
| A	C	
| ϵ	A	C, B, A
| D		
| D, E		
| D	E	
| ϵ	D	D,E

E da un vertice diverso?

Primo passo



V	Adj
D	E
G	F,C,I
E	D
A	B
C	A
I	G
F	E,H
H	G
B	C,D

t	Stack	V
0	| D	
1	| D, E	
2	| D	E
3	| ϵ	D
4	| G	
5	| G, F	
6	| G, F, H	
7	| G, F	H
8	| G	F
9	| G, C	
10	| G, C, A	
11	| G, C, A, B	
12	| G, C, A	B
13	| G, C	A
14	| G	C
15	| G, I	
16	| G	I
17	| ϵ	G

Da un vertice diverso

Secondo passo

Grafo originale	
V	**Adj**
D	E
G	F,C,I
E	D
A	B
C	A
I	G
F	E,H
H	G
B	C,D
Grafo trasposto	
G	H,I
I	G
C	G,B
A	C
B	A
F	G
H	F
D	B,E
E	F,D

Stack	V	CC
| G		
| G, H		
| G, H, F		
| G, H	F	
| G	H	
| G, I		
| G	I	
| ϵ	G	F,H,I,G
| C		
| C, B		
| C, B, A		
| C, B	A	
| C	B	
| ϵ	C	A, B, C
| D		
| D, E		
| D	E	
| ϵ	D	D,E