Algoritmi e Strutture Dati

Grafi

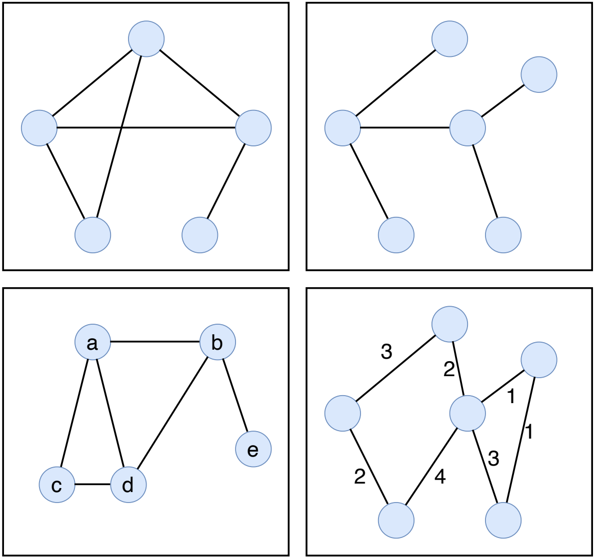
Alberto Montresor

Università di Trento

2024/06/16

Sommario

Esempi



Problemi relativi ai grafi

Problemi in grafi non pesati

- Ricerca del cammino più breve (misurato in numero di archi)
- Componenti (fortemente) connesse, verifica ciclicità, ordinamento topologico

Problemi in grafi pesati

- Cammini di peso minimo
- Alberi di copertura di peso minimo
- Flusso massimo

Problemi relativi ai grafi

Moltissimi problemi possono essere visti come problemi su grafi. Sebbene i problemi abbiano forma astratta, le loro applicazioni si trovano poi negli ambiti più disparati

Esempi

- Quando cercate qualcuno su LinkedIn, vi restituisce un "grado di conoscenza": e.g., la lunghezza del più breve cammino fra me e Bill Gates nella rete sociale di LinkedIn è pari a 3.
- L'ordinamento topologico viene utilizzato per stabilire un ordine di azioni in un grafo di dipendenze.
- Gli algoritmi di model checking utilizzati per la verifica formale del software sono basati sull'identificazione delle componenti fortemente connesse.

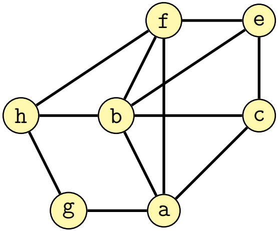
Un esempio di applicazione

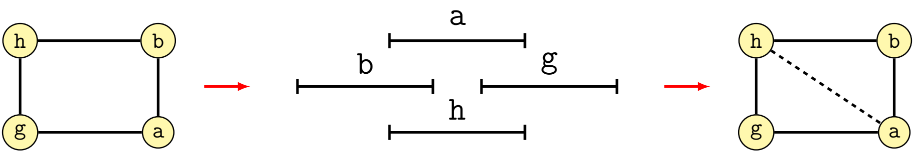
Watson e Holmes indagano sulla morte del duca MacPollock

- Watson: “Ci sono novità, Holmes: pare che il testamento, andato distrutto nell’esplosione, fosse stato favorevole ad una delle sette ‘amiche’ del duca.”
- Holmes: “Ciò che è più strano, è che la bomba sia stata fabbricata appositamente per essere nascosta nell’armatura della camera da letto, il che fa supporre che *l’assassino abbia necessariamente fatto più di una visita al castello*.”
- Watson: “Ho interrogato personalmente le sette donne, ma ciascuna ha giurato di essere stata nel castello *una sola volta nella sua vita*. Dagli interrogatori risulta che:
  - Ann ha incontrato Betty, Charlotte, Felicia e Georgia;
  - Betty ha incontrato Ann, Charlotte, Edith, Felicia e Helen;
  - Charlotte ha incontrato Ann, Betty e Edith;
  - Edith ha incontrato Betty, Charlotte, Felicia;
  - Felicia ha incontrato Ann, Betty, Edith, Helen;
  - Georgia ha incontrato Ann e Helen;
  - Helen ha incontrato Betty, Felicia e Georgia.

  Vedete, Holmes, che le testimonianze concordano. Ma chi sarà l’assassino?”
- Holmes: “Elementare, mio caro Watson: ciò che mi avete detto individua inequivocabilmente l’assassino!”

Un esempio di applicazione





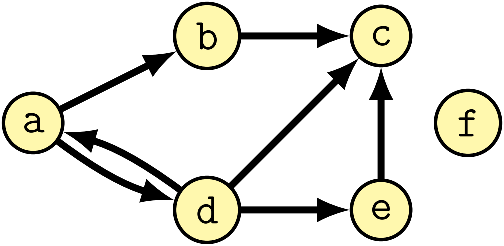
Grafi orientati e non orientati: definizioni

Grafo orientato (directed)

È una coppia $G = (V, E)$ dove:

- V è un insieme di nodi (node) o vertici (vertex)
- E è un insieme di coppie ordinate di nodi (u, v) dette archi o lati (edge)

```
V = { a,b,c,d,e,f }
E = { (a,b), (a,d),
      (b,c), (d,a),
      (d,c), (d,e),
      (e,c) }
```



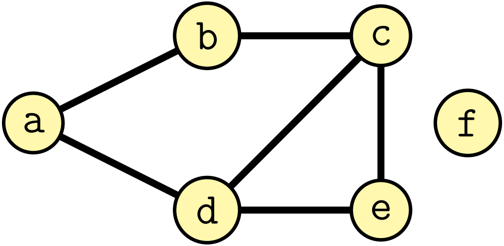
Grafi orientati e non orientati: definizioni

Grafo non orientato (undirected)

È una coppia $G = (V, E)$ dove:

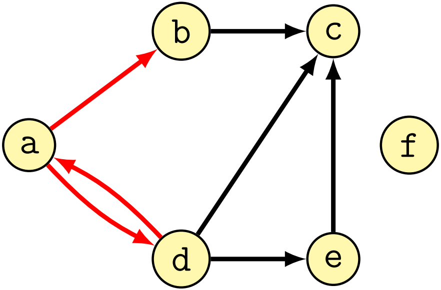
- V è un insieme di nodi (node) o vertici (vertex)
- E è un insieme di coppie non ordinate di nodi (u, v) dette archi o lati (edge)

```
V = { a,b,c,d,e,f }
E = { (a,b), (a,d),
      (b,c), (c,d),
      (d,e), (c,e)  }
```



Terminologia

- Un vertice v è detto adiacente a u se esiste un arco (u, v)
- Un arco (u, v) è detto incidente da u a v
- In un grafo indiretto, la relazione di adiacenza è simmetrica



- (a, b) è incidente da a a b
- (a, d) è incidente da a a d
- (d, a) è incidente da d a a
- b è adiacente a a
- d è adiacente a a
- a è adiacente a d

Dimensioni del grafo

Definizioni

- $n = |V|$: numero di nodi
- $m = |E|$: numero di archi

Alcune relazioni fra n e m

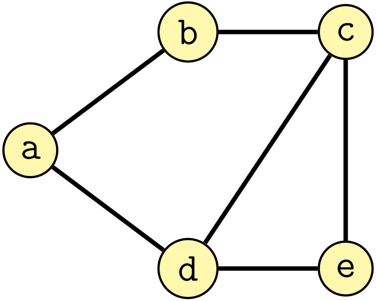
- In grafo non orientato, $m \leq \frac{n(n-1)}{2} = O(n^2)$
- In grafo orientato, $m \leq n^2 - n = O(n^2)$

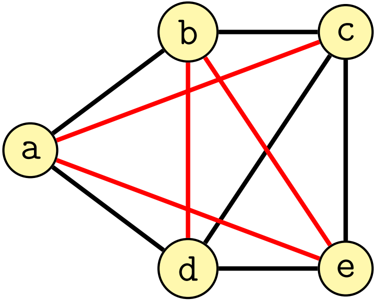
Complessità di algoritmi su grafi

- La complessità è espressa in termini sia di n che di m (ad es. $O(n + m)$)

Alcuni casi speciali

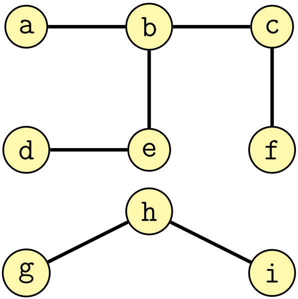
- Un grafo con un arco fra tutte le coppie di nodi è detto completo
- Informalmente (non c'è accordo sulla definizione)
  - Un grafo si dice sparso se ha "pochi archi"; grafi con $m = O(n)$, $m = O(n \log n)$ sono considerati sparsi
  - Un grafo si dice denso se ha "tanti archi"; e.g., $m = \Omega(n^2)$

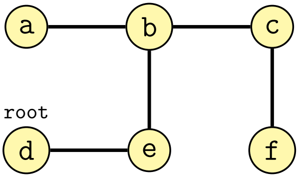




Alcuni casi speciali

- Un albero libero (free tree) è un grafo connesso con $m = n - 1$
- Un albero radicato (rooted tree) è un grafo connesso con $m = n - 1$ nel quale uno dei nodi è designato come radice.
- Un insieme di alberi è un grafo detto foresta





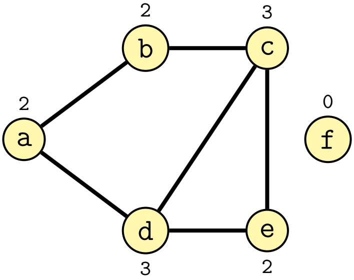
Definizioni: Grado

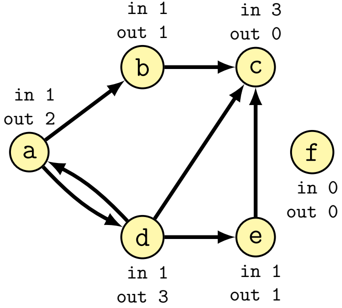
Grafi non orientati

Il grado (degree) di un nodo è il numero di archi incidenti su di esso.

Grafi orientati

Il grado entrante (in-degree) di un nodo è il numero di archi incidenti su di esso. Il grado uscente (out-degree) di un nodo è il numero di archi incidenti da esso.

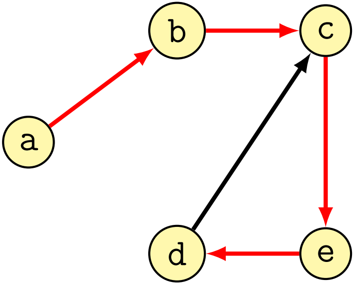




Definizioni: Cammino

Cammino (Path)

In un grafo $G = (V, E)$ (orientato oppure no), un cammino C di lunghezza k è una sequenza di nodi $u_0, u_1, \ldots, u_k$ tale che $(u_i, u_{i+1}) \in E$ per $0 \le i \le k-1$.



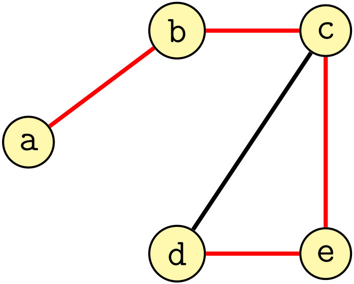
Esempio: a, b, c, e, d è un cammino nel grafo di lunghezza 4

Nota: un cammino è detto semplice se tutti i suoi nodi sono distinti

Definizioni: Cammino

Cammino (Path)

In un grafo $G = (V, E)$ (orientato oppure no), un cammino C di lunghezza k è una sequenza di nodi $u_0, u_1, \ldots, u_k$ tale che $(u_i, u_{i+1}) \in E$ per $0 \leq i \leq k - 1$.



Esempio: a, b, c, e, d è un cammino nel grafo di lunghezza 4

Nota: un cammino è detto semplice se tutti i suoi nodi sono distinti

Specifica – Grafi dinamici

Nella versione più generale, il grafo è una struttura di dati dinamica che permette di aggiungere/rimuovere nodi e archi.

GRAPH	
Graph( )	% Crea un nuovo grafo
SET V()	% Restituisce l'insieme di tutti i nodi
int size()	% Restituisce il numero di nodi
SET adj(NODE u)	% Restituisce l'insieme dei nodi adiacenti a u
insertNode(NODE u)	% Aggiunge il nodo u al grafo
insertEdge(NODE u, NODE v)	% Aggiunge l'arco (u, v) al grafo
deleteNode(NODE u)	% Rimuove il nodo u dal grafo
deleteEdge(NODE u, NODE v)	% Rimuove l'arco (u, v) dal grafo

Specifica ridotta (senza rimozioni)

- In alcuni casi, il grafo è dinamico ma sono possibili solo inserimenti
- Il grafo viene caricato all'inizio e poi non viene modificato
- Questo ha riflessi sull'implementazione sottostante

GRAPH	
Graph( )	% Crea un nuovo grafo
SET V()	% Restituisce l'insieme di tutti i nodi
int size()	% Restituisce il numero di nodi
SET adj(NODE u)	% Restituisce l'insieme dei nodi adiacenti a u
insertNode(NODE u)	% Aggiunge il nodo u al grafo
insertEdge(NODE u, NODE v)	% Aggiunge l'arco (u, v) al grafo

Memorizzare grafi

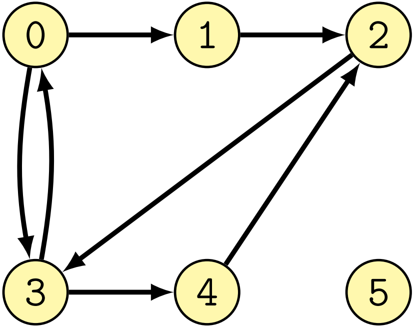
Due possibili approcci

- Matrici di adiacenza
- Liste di adiacenza

Matrice di adiacenza: grafi orientati

$$m_{uv} = \begin{cases} 1 & (u, v) \in E \\ 0 & (u, v) \notin E \end{cases}$$

Spazio $= n^2$ bit

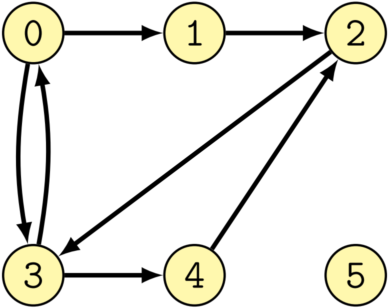


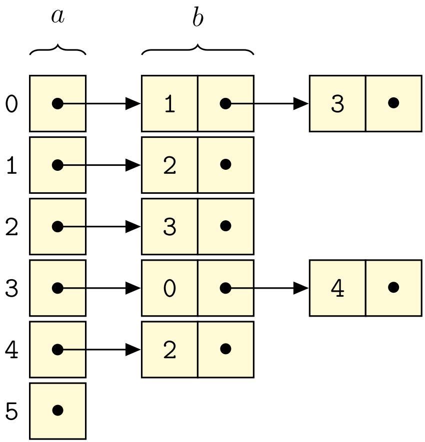
$$\begin{array}{c c} & \begin{array}{cccccc} \mathbf{0} & \mathbf{1} & \mathbf{2} & \mathbf{3} & \mathbf{4} & \mathbf{5} \end{array} \\ \begin{array}{c} \mathbf{0} \\ \mathbf{1} \\ \mathbf{2} \\ \mathbf{3} \\ \mathbf{4} \\ \mathbf{5} \end{array} & \begin{pmatrix} 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

Liste di adiacenza: grafi orientati

$G.\mathsf{adj}(u) = \{v | (u, v) \in E\}$

Spazio $= an + bm$ bit

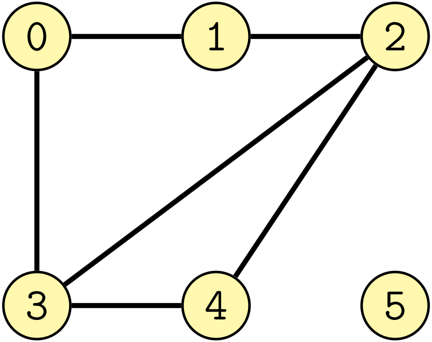




Matrice di adiacenza: grafi non orientati

$$m_{uv} = \begin{cases} 1 & (u,v) \in E \\ 0 & (u,v) \notin E \end{cases}$$

Spazio $= n^2$ oppure $n(n-1)/2$ bit

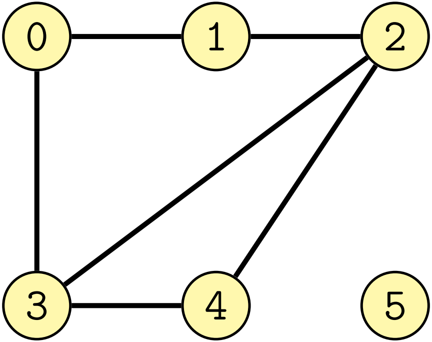


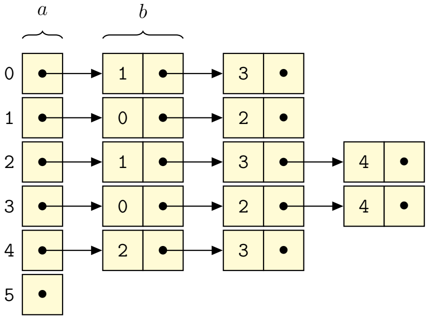
$$\begin{array}{c c} & \begin{array}{cccccc} \mathbf{0} & \mathbf{1} & \mathbf{2} & \mathbf{3} & \mathbf{4} & \mathbf{5} \end{array} \\ \begin{array}{c} \mathbf{0} \\ \mathbf{1} \\ \mathbf{2} \\ \mathbf{3} \\ \mathbf{4} \\ \mathbf{5} \end{array} & \begin{pmatrix} & 1 & 0 & 1 & 0 & 0 \\ & & 1 & 0 & 0 & 0 \\ & & & 1 & 1 & 0 \\ & & & & 1 & 0 \\ & & & & & 0 \\ & & & & & \end{pmatrix} \end{array}$$

Liste di adiacenza: grafo non orientato

$G.\mathsf{adj}(u) = \{v | (u, v) \in E\}$

Spazio $= an + 2 \cdot bm$

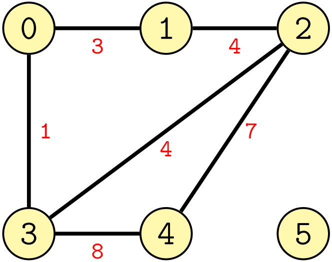




Matrice di adiacenza: grafi pesati

Grafi pesati

- Gli archi possono avere un peso (costo, profitto, etc.)
- Il peso è dato da una funzione di peso $w : V \times V \to \mathbb{R}$
- Se non esiste arco fra due vertici, il peso assume un valore che dipende dal problema - e.g. $w(u, v) = 0$ oppure $+\infty$

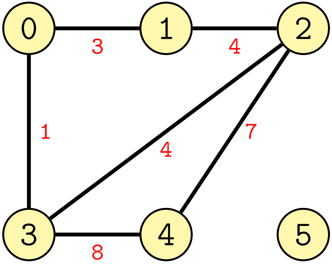


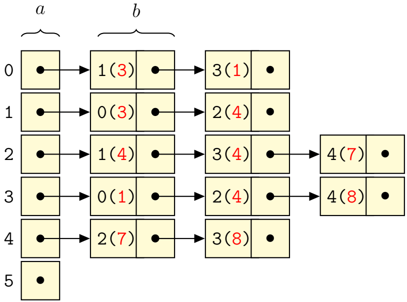
$$\begin{array}{c c}
 & \begin{array}{cccccc} \mathbf{0} & \mathbf{1} & \mathbf{2} & \mathbf{3} & \mathbf{4} & \mathbf{5} \end{array} \\
\begin{array}{c} \mathbf{0} \\ \mathbf{1} \\ \mathbf{2} \\ \mathbf{3} \\ \mathbf{4} \\ \mathbf{5} \end{array} &
\begin{pmatrix}
 & 3 & 0 & 1 & 0 & 0 \\
 & & 4 & 0 & 0 & 0 \\
 & & & 4 & 7 & 0 \\
 & & & & 8 & 0 \\
 & & & & & 0 \\
 & & & & &
\end{pmatrix}
\end{array}$$

Liste di adiacenza: grafi pesati

Grafi pesati

- Gli archi possono avere un peso (costo, profitto, etc.)
- Il peso è dato da una funzione di peso $w : V \times V \rightarrow \mathbb{R}$
- Se non esiste arco fra due vertici, il peso assume un valore che dipende dal problema - e.g. $w(u, v) = +\infty$ oppure 0





Liste di adiacenza - variazioni sul tema

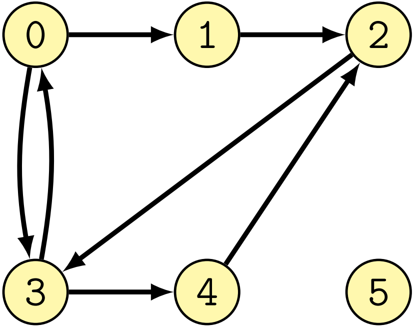
Sia il concetto di *lista di adjacenza* che il concetto di *lista dei nodi* possono essere declinati in molti modi:

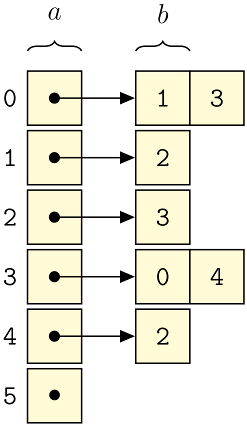
Struttura	**Java**	**C++**	**Python**
Lista collegata	`LinkedList`	`list`	
Vettore statico	`[]`	`[]`	`list`
Vettore dinamico	`ArrayList`	`vector`	`list`
Insieme	`HashSet` `TreeSet`	`set`	`set`
Dizionario	`HashMap` `TreeMap`	`map`	`dict`

Vettore di adiacenza: grafo orientato

$G.\mathsf{adj}(u) = \{v | (u, v) \in E\}$

Spazio $= an + bm$ bit





Dettagli sull'implementazione

Se non diversamente specificato, nel seguito:

- Assumeremo che l'implementazione sia basata su vettori di adiacenza, statici o dinamici
- Assumeremo che la classe NODE sia equivalente a **int** (quindi l'accesso alle informazioni avrà costo $O(1)$)
- Assumeremo che le operazioni per aggiungere nodi e archi abbiano costo $O(1)$
- Assumeremo che dopo l'inizializzazione, il grafo sia statico

Implementazione (pesata) con dizionari – Python

```
class Graph:

  def __init__(self):
    self.nodes = { }

  def V(self):
    return self.nodes.keys()

  def size(self)
    return len(self.nodes)

  def adj(self, u):
    if u in self.nodes:
      return self.nodes[u]

  def insertNode(self,u):
    if u not in self.nodes:
      self.nodes[u] = { }

  def insertEdge(self, u, v, w=0):
    self.insertNode(u)
    self.insertNode(v)
    self.nodes[u][v] = w
```

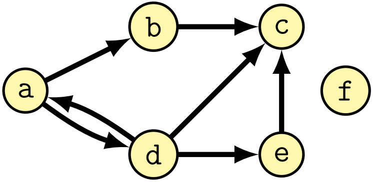
Implementazione (pesata) con dizionari – Python[1]

```
graph = Graph()

for u,v in [ ('a', 'b'), ('a', 'd'), ('b', 'c'),
     ('d', 'a'), ('d', 'c'), ('d', 'e'), ('e', 'c') ]:
  graph.insertEdge(u,v)

for u in graph.V():
  print(u, "->", graph.adj(u))
```

```
f -> {}
b -> {'c': 0}
e -> {'c': 0}
a -> {'b': 0, 'd': 0}
d -> {'e': 0, 'c': 0, 'a': 0}
c -> {}
```



[1] https://www.python.org/doc/essays/graphs/, Guido van Rossum

Iterazione su nodi e archi

Iterazione su tutti i nodi del grafo

foreach $u \in G.\mathsf{V}()$ **do**
  { Esegui operazioni sul nodo u }

Iterazione su tutti i nodi e archi del grafo

foreach $u \in G.\mathsf{V}()$ **do**
  { Esegui operazioni sul nodo u }
  foreach $v \in G.\mathsf{adj}(u)$ **do**
    { Esegui operazioni sull'arco (u, v) }

Costo computazionale

- $O(m + n)$ con liste di adiacenza
- $O(n^2)$ con matrici di adiacenza

Riassumendo

Matrici di adiacenza

- Spazio richiesto $O(n^2)$
- Verificare se u è adiacente a v richiede tempo $O(1)$
- Iterare su tutti gli archi richiede tempo $O(n^2)$
- Ideale per grafi densi

Liste di adiacenza

- Spazio richiesto $O(n+m)$
- Verificare se u è adiacente a v richiede tempo $O(n)$
- Iterare su tutti gli archi richiede tempo $O(n+m)$
- Ideale per grafi sparsi

Sebbene i matematici preferiscano la rappresentazione a matrice di adiacenza dei grafi per "maneggiarli" con algebra lineare, tale rappresentazione non è ideale per informatici interessati a implementazioni algoritmiche efficienti

Visite dei grafi

Definizione del problema

Dato un grafo $G = (V, E)$ e un vertice $r \in V$ (radice, sorgente), visitare una e una volta sola tutti i nodi del grafo che possono essere raggiunti da r

Visita in ampiezza (Breadth-first search) (BFS)

Visita dei nodi per livelli: prima si visita la radice, poi i nodi a distanza 1 dalla radice, poi a distanza 2, etc.

- Applicazione: calcolare i cammini più brevi da una singola sorgente

Visite dei grafi

Definizione del problema

Dato un grafo $G = (V, E)$ e un vertice $r \in V$ (radice, sorgente), visitare una e una volta sola tutti i nodi del grafo che possono essere raggiunti da r

Visita in profondità (Depth-First Search) (DFS)

Visita ricorsiva: per ogni nodo adiacente, si visita ricorsivamente tale nodo, visitando ricorsivamenti i suoi nodi adiacenti, etc.

- Applicazione: ordinamento topologico
- Applicazione: componente connesse, componenti fortemente connesse

Visita: leggermente più difficile di quanto sembri

Un approccio ingenuo alla visita di un grafo potrebbe essere il seguente:

```
visit(GRAPH G)
foreach u ∈ G.V() do
    { visita nodo u }
    foreach v ∈ G.adj(u) do
        { visita arco (u, v) }
```

- La struttura del grafo non è tenuta in considerazione
- Si itera su tutti i nodi e gli archi senza nessun criterio

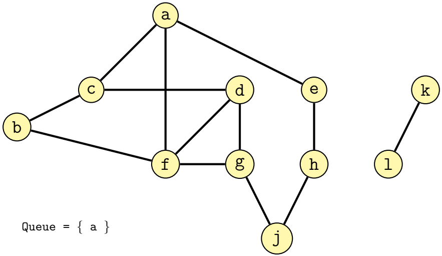
Visita: leggermente più difficile di quanto sembri

Un possibile approccio: utilizzare le visite degli alberi

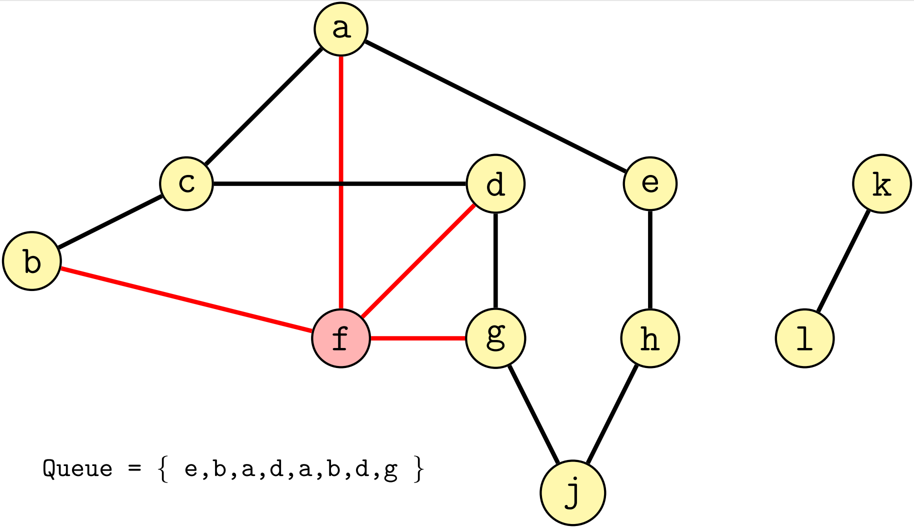
- Chiamare una BFS a partire da un nodo
- I nodi adiacenti sono trattati come figli

BFSTraversal(Graph G, **int** r)

```
QUEUE Q = Queue()
Q.enqueue(r)
while not Q.isEmpty() do
    NODE u = Q.dequeue()
    { visita il nodo u }
    foreach v ∈ G.adj(u) do
        Q.enqueue(v)
```



Esempio: Visita errata



`Queue = { e,b,a,d,a,b,d,g }`

Algoritmo generico di attraversamento

```
graphTraversal(GRAPH G, NODE r)
SET S = Set()                                   % Insieme generico
S.insert(r)                                     % Da specificare
{ marca il nodo r }
while S.size() > 0 do
    NODE u = S.remove()                         % Da specificare
    { visita il nodo u }
    foreach v ∈ G.adj(u) do
        { visita l'arco (u, v) }
        if v non è ancora stato marcato then
            { marca il nodo v }
            S.insert(v)                         % Da specificare
```

Breadth-first search - Obiettivi

Visitare i nodi a distanze crescenti dalla sorgente

- Visitare i nodi a distanza k prima di visitare i nodi a distanza $k+1$

Calcolare il cammino più breve da r a tutti gli altri nodi

- Le distanze sono misurate come il numero di archi attraversati

Generare un albero breadth-first

- Generare un albero contenente tutti i nodi raggiungibili da r, tale per cui il cammino dalla radice r al nodo u nell'albero corrisponde al cammino più breve da r a u nel grafo.

Breadth-first search

bfs(Graph G, Node r)

```
Queue Q = Queue()
S.enqueue(r)
boolean[] visited = new boolean[G.size()]
foreach u ∈ G.V() − {r} do
    visited[u] = false
visited[r] = true
while not Q.isEmpty() do
    Node u = Q.dequeue()
    { visita il nodo u }
    foreach v ∈ G.adj(u) do
        { visita l'arco (u, v) }
        if not visited[v] then
            visited[v] = true
            Q.enqueue(v)
```

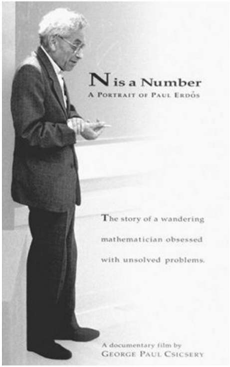
Applicazione BFS: Cammini più brevi

Paul Erdös (1913-1996)

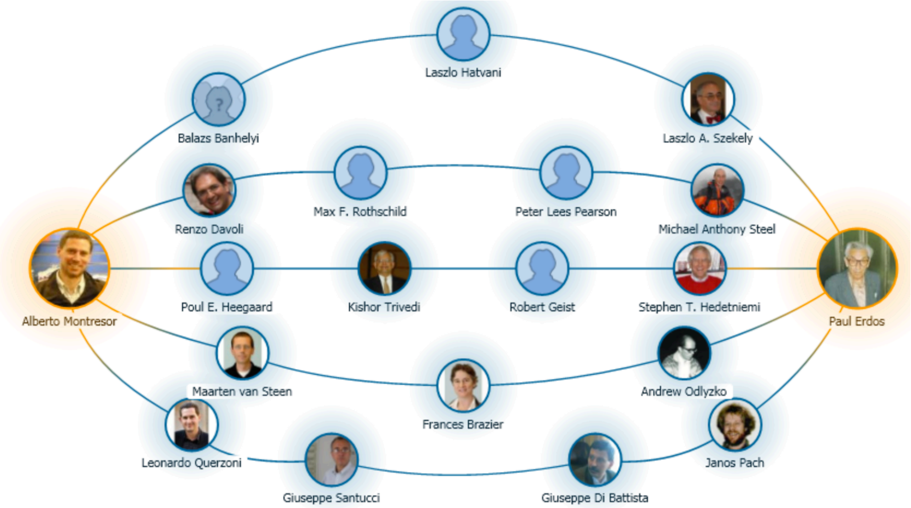
- Matematico
- 1500+ articoli, 500+ co-autori

Numero di Erdös

- Erdös ha valore $erdos = 0$
- I co-autori di Erdös hanno $erdos = 1$
- Se X è co-autore di qualcuno con $erdos = k$ e non è coautore con qualcuno con $erdos < k$, allora X ha $erdos = k + 1$
- Le persone non raggiunte da questa definizione hanno $erdos = +\infty$



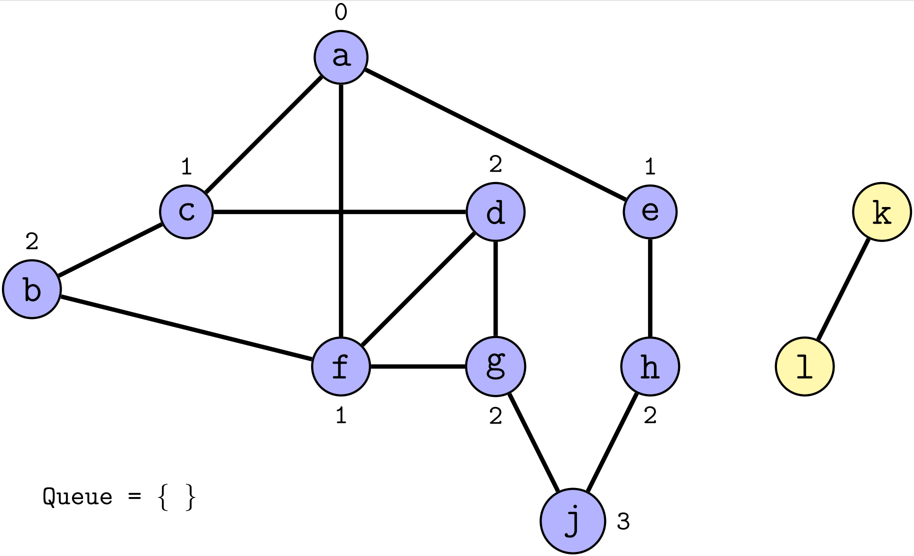
Alberto Montresor, *erdos* = 4



Calcolare il numero di Erdös

```
distance(GRAPH G, NODE r, int[] distance)
QUEUE Q = Queue()
Q.enqueue(r)
foreach u ∈ G.V() − {r} do
    distance[u] = ∞
distance[r] = 0
while not Q.isEmpty() do
    NODE u = Q.dequeue()
    foreach v ∈ G.adj(u) do
        if distance[v] == ∞ then % Se il nodo v non è stato scoperto
            distance[v] = distance[u] + 1
            Q.enqueue(v)
```

Esempio: Erdös



`Queue = { }`

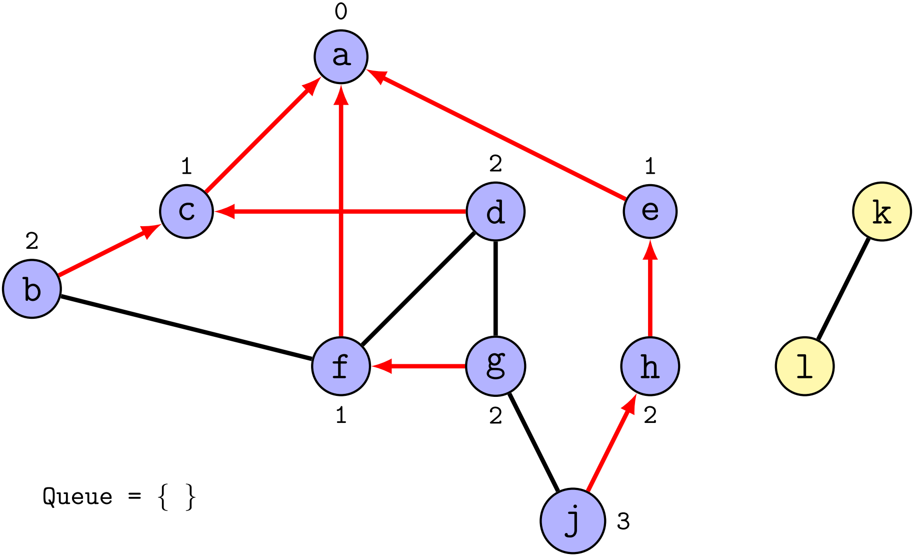
Albero BFS (BFS Tree)

- La visita BFS può essere usata per ottenere il cammino più breve fra due nodi (misurato in numero di archi)
- "Albero di copertura" con radice r
- Memorizzato in un vettore dei padri *parent*

```
distance([...], NODE[] parent)
[...]
parent[r] = nil
while not S.isEmpty() do
    NODE u = S.dequeue()
    foreach v ∈ G.adj(u) do
        if distance[v] == ∞ then
            distance[v] =
              distance[u] + 1
            parent[v] = u
            S.enqueue(v)
```

```
printPath(NODE r, NODE s, NODE[] parent)
if r == s then
    print s
else if parent[s] == nil then
    print "error"
else
    printPath(r, parent[s], parent)
    print s
```

Albero BFS (BFS Tree)



Queue = { }

Complessità BFS

Complessità: $O(m + n)$

- Ognuno degli n nodi viene inserito nella coda al massimo una volta
- Ogni volta che un nodo viene estratto, tutti i suoi archi vengono analizzati una volta sola
- Il numero di archi analizzati è quindi

$$m = \sum_{u \in V} d_{out}(u)$$

dove d_{out} è l'out-degree del nodo u

Depth-First Search (DFS)

Depth-First Search

- Spesso una subroutine della soluzione di altri problemi
- Utilizzata per esplorare un intero grafo, non solo i nodi raggiungibili da una singola sorgente

Output

- Invece di un albero, una foresta depth-first $G_f = (V, E_f)$
- Formata da una collezione di alberi depth-first

Struttura dati

- Stack implicito, attraverso la ricorsione
- Stack esplicito

Depth-First Search (Ricorsiva, stack implicito)

```
dfs(Graph G, Node u, boolean[] visited)
visited[u] = true
{ visita il nodo u (pre-order) }
foreach v ∈ G.adj(u) do
    if not visited[v] then
        { visita l'arco (u, v) }
        dfs(G, v, visited)
{ visita il nodo u (post-order) }
```

Complessità: $O(m + n)$

BFS vs DFS

- Eseguire una DFS basata su chiamate ricorsive può essere rischioso in grafi molto grandi e connessi
- È possibile che la profondità raggiunta sia troppo grande per la dimensione dello stack del linguaggio
- In tali casi, si preferisce utilizzare una BFS oppure una DFS basata su stack esplicito

Stack size in Java

Platform	**Default**
Windows IA32	64 KB
Linux IA32	128 KB
Windows x86_64	128 KB
Linux x86_64	256 KB
Windows IA64	320 KB
Linux IA64	1024 KB (1 MB)
Solaris Sparc	512 KB

DFS (Iterativa, stack esplicito, pre-order)

```
dfs(Graph G, Node r)
Stack S = Stack()
S.push(r)
boolean[] visited = new boolean[G.size()]
foreach u ∈ G.V() do
    visited[u] = false
while not S.isEmpty() do
    Node u = S.pop()
    if not visited[u] then
        { visita il nodo u (pre-order) }
        visited[u] = true
        foreach v ∈ G.adj(u) do
            { visita l'arco (u, v) }
            S.push(v)
```

https://11011110.github.io/blog/2013/12/17/stack-based-graph-traversal.html

Note

- Un nodo può essere inserito nella pila più volte
- Il controllo se un nodo è già stato visitato viene fatto all'estrazione, non all'inserimento
- Complessità $O(m + n)$
  - $O(m)$ visite degli archi
  - $O(m)$ inserimenti, estrazioni
  - $O(n)$ visite dei nodi

DFS (Iterativa, stack esplicito, post-order)

Visita post-order

- Quando un nodo viene scoperto:
  - viene inserito nello stack con il tag discovery
- Quando un nodo viene estratto dallo stack con tag discovery:
  - Viene re-inserito con il tag finish
  - Tutti i suoi vicini vengono inseriti
- Quando un nodo viene estratto dallo stack con tag finish:
  - Viene effettuata la post-visita

Componenti (fortemente) connesse

Motivazioni

- Molti algoritmi che operano sui grafi iniziano decomponendo il grafo nel sue componenti connesse.
- Tali algoritmi sono eseguiti su ognuna delle componenti
- I risultati sono ricomposti assieme.

Definizioni

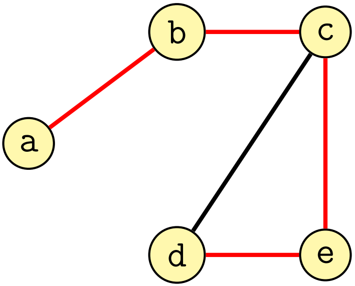
- Componenti connesse, definite su grafi non orientati (*Connected components*, CC)
- Componenti fortemente connesse, definite su grafi orientati (*Strongly connected components*, SCC)

Definizioni: Raggiungibilità

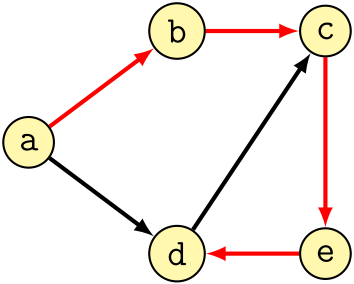
Definizione

Un nodo v è raggiungibile da un nodo u se esiste almeno un cammino da u a v.

Il nodo d è raggiungibile dal nodo a e viceversa



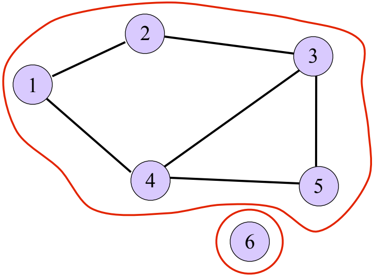
Il nodo d è raggiungibile dal nodo a, ma non viceversa



Grafi connessi e componenti connesse

Definizioni

- Un grafo non orientato $G = (V, E)$ è connesso $\Leftrightarrow$ ogni suo nodo è raggiungibile da ogni altro suo nodo
- Un grafo $G' = (V', E')$ è una componente connessa di $G \Leftrightarrow G'$ è un sottografo connesso e massimale di G
- G' è un sottografo di G $(G' \subseteq G) \Leftrightarrow V' \subseteq V$ e $E' \subseteq E$
- G' è massimale $\Leftrightarrow \nexists$ un altro sottografo G'' di G tale che G'' è connesso e più grande di G' (i.e. $G' \subseteq G'' \subseteq G$)



Applicazione DFS: Componenti connesse

Problema

- Verificare se un grafo è connesso oppure no
- Identificare le sue componenti connesse

Soluzione

- Un grafo è connesso se, al termine della DFS, tutti i nodi sono marcati
- Altrimenti, la visita deve ricominciare da capo da un nodo non marcato, identificando una nuova componente del grafo

Strutture dati

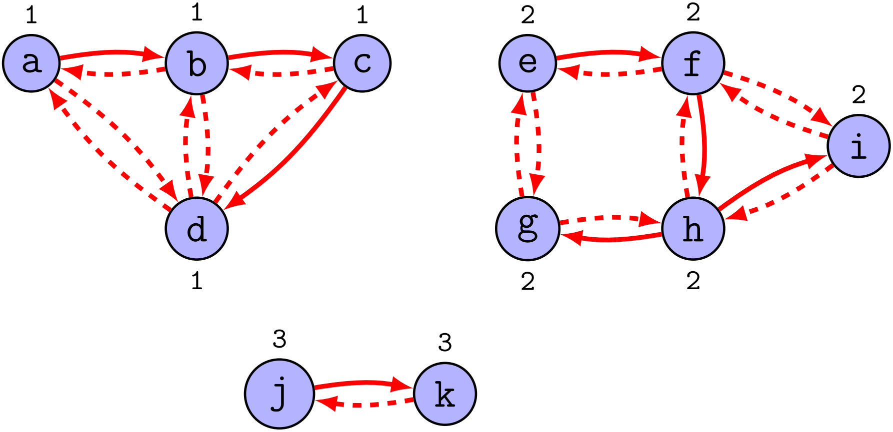
- Un vettore id, che contiene gli identificatori delle componenti
- $id[u]$ è l'identificatore della c.c. a cui appartiene u

Applicazione DFS: Componenti connesse

```
int[] cc(GRAPH G)
---------------------------------
int[] id = new int[G.size()]
foreach u ∈ G.V() do
    id[u] = 0
int counter = 0
foreach u ∈ G.V() do
    if id[u] == 0 then
        counter = counter + 1
        ccdfs(G, counter, u, id)
return id
```

```
ccdfs(GRAPH G, int counter, NODE u, int[] id)
---------------------------------
id[u] = counter
foreach v ∈ G.adj(u) do
    if id[v] == 0 then
        ccdfs(G, counter, v, id)
```

Esempio: Componenti connesse

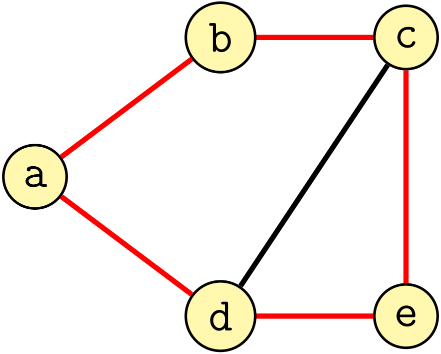


v

Definizioni: Ciclo

Ciclo (cycle)

In un grafo non orientato $G = (V, E)$, un ciclo C di lunghezza $k > 2$ è una sequenza di nodi $u_0, u_1, \ldots, u_k$ tale che $(u_i, u_{i+1}) \in E$ per $0 \leq i \leq k-1$ e $u_0 = u_k$.

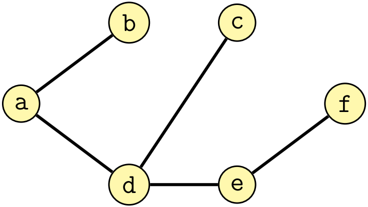


$k > 2$ esclude cicli banali composti da coppie di archi (u, v) e (v, u), che sono onnipresenti nei grafi non orientati.

Definizioni: Grafo aciclico

Grafo aciclico

Un grafo non orientato che non contiene cicli è detto aciclico.



Problema

Dato un grafo non orientato G, scrivere un algoritmo che restituisca **true** se G contiene un ciclo, **false** altrimenti.

Applicazione DFS: Grafo non orientato aciclico

```
boolean hasCycleRec(Graph G, Node u, Node p, boolean[] visited)
visited[u] = true
foreach v ∈ G.adj(u) − {p} do
    if visited[v] then
        return true
    else if hasCycleRec(G, v, u, visited) then
        return true
return false
```

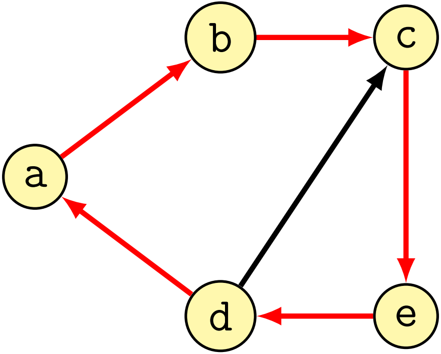
Applicazione DFS: Grafo non orientato aciclico

```
boolean hasCycle(Graph G)
boolean[] visited = new boolean[G.size()]
foreach u ∈ G.V() do
    visited[u] = false
foreach u ∈ G.V() do
    if not visited[u] then
        if hasCycleRec(G, u, null, visited) then
            return true
return false
```

Definizioni: Ciclo

Ciclo (cycle)

In un grafo orientato $G = (V, E)$, un ciclo C di lunghezza $k \geq 2$ è una sequenza di nodi $u_0, u_1, \ldots, u_k$ tale che $(u_i, u_{i+1}) \in E$ per $0 \leq i \leq k-1$ e $u_0 = u_k$.



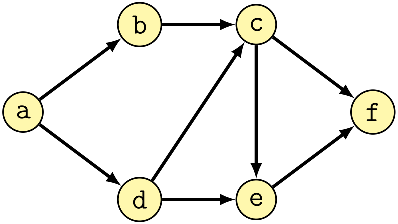
Esempio: a, b, c, e, d, a è un cammino nel grafo di lunghezza 5

Note: un ciclo è detto semplice se tutti i suoi nodi sono distinti (ad esclusione del primo e dell'ultimo)

Definizioni: Grafo orientato aciclico (DAG)

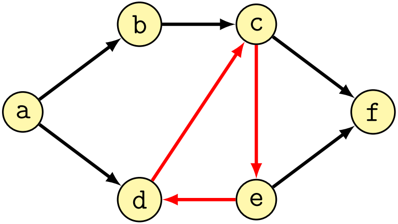
DAG

Un grafo orientato che non contiene cicli è detto DAG (directed acyclic graph).



Grafo ciclico

Un grafo è ciclico se contiene un ciclo.



Applicazione DFS: Grafo orientato aciclico

Problema

Dato un grafo orientato G, scrivere un algoritmo che restituisca **true** se G contiene un ciclo, **false** altrimenti.

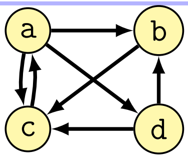
Problema

Riuscite a concepire un grafo orientato per cui l'algoritmo appena visto non si comporta correttamente?

Classificazione degli archi

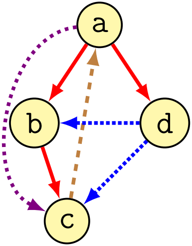
Albero di copertura DFS

Ogni volta che si esamina un arco da un nodo marcato ad un nodo non marcato, tale arco viene arco dell'albero



Gli archi (u, v) non inclusi nell'albero possono essere divisi in tre categorie

- Se u è un antenato di v in T, (u, v) è detto arco in avanti
- Se u è un discendente di v in T, (u, v) è detto arco all'indietro
- Altrimenti, viene detto arco di attraversamento



DFS Schema

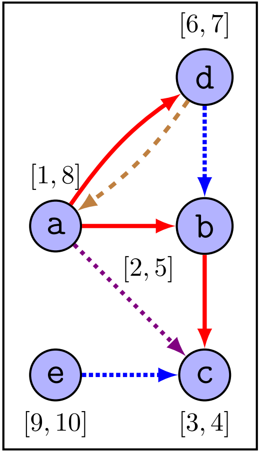
```
dfs-schema(GRAPH G, NODE u, int &time, int[] dt, int[] ft)
{ visita il nodo u (pre-order) }
time = time + 1;   dt[u] = time
foreach v ∈ G.adj(u) do
    { visita l'arco (u, v) (qualsiasi) }
    if dt[v] == 0 then
        { visita l'arco (u, v) (albero) }
        dfs-schema(G, v, time, dt, ft)
    else if dt[u] > dt[v] and ft[v] == 0 then
        { visita l'arco (u, v) (indietro) }
    else if dt[u] < dt[v] and ft[v] ≠ 0 then
        { visita l'arco (u, v) (avanti) }
    else
        { visita l'arco (u, v) (attraversamento) }
{ visita il nodo u (post-order) }
time = time + 1;   ft[u] = time
```

- *time*: contatore
- *dt*: discovery time (tempo di scoperta)
- *ft*: finish time (tempo di fine)

DFS Schema

dfs-schema(Graph G, Node u, **int** $\&time$, **int**[] dt, **int**[] ft)

```
time = time + 1;   dt[u] = time
foreach v ∈ G.adj(u) do
    if dt[v] == 0 then
        { visita l'arco (u, v) (albero) }
        dfs-schema(G, v, time, dt, ft)
    else if dt[u] > dt[v] and ft[v] == 0 then
        { visita l'arco (u, v) (indietro) }
    else if dt[u] < dt[v] and ft[v] ≠ 0 then
        { visita l'arco (u, v) (avanti) }
    else
        { visita l'arco (u, v) (attraversamento) }
time = time + 1;   ft[u] = time
```



Classificazione degli archi

Perchè classificare gli archi?

Possiamo dimostrare proprietà sul tipo degli archi e usare queste proprietà per costruire algoritmi migliori

Teorema

Data una visita DFS di un grafo $G = (V, E)$, per ogni coppia di nodi $u, v \in V$, solo una delle condizioni seguenti è vera:

- Gli intervalli $[dt[u], ft[u]]$ e $[dt[v], ft[v]]$ sono non-sovrapposti; u, v non sono discendenti l'uno dell'altro nella foresta DF
- L'intervallo $[dt[u], ft[u]]$ è contenuto in $[dt[v], ft[v]]$; u è un discendente di v in un albero DF
- L'intervallo $[dt[v], ft[v]]$ è contenuto in $[dt[u], ft[u]]$; v è un discendente di u in un albero DF

Teoria

Teorema

Un grafo orientato è aciclico $\Leftrightarrow$ non esistono archi all'indietro nel grafo.

Dimostrazione

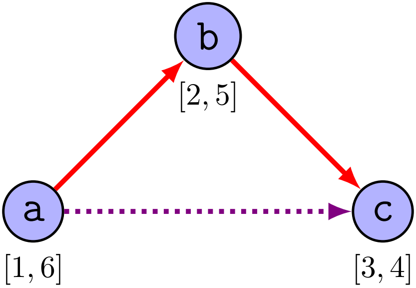
- **se**: Se esiste un ciclo, sia u il primo nodo del ciclo che viene visitato e sia (v, u) un arco del ciclo. Il cammino che connette u ad v verrà prima o poi visitato, e da v verrà scoperto l'arco all'indietro (v, u).
- **solo se**: Se esiste un arco all'indietro (u, v), dove v è un antenato di u, allora esiste un cammino da v a u e un arco da u a v, ovvero un ciclo.

Applicazione DFS: DAG

boolean hasCycleRec(GRAPH G, NODE u, **int** $\&time$, **int**[] dt, **int**[] ft)

```
time = time + 1;   dt[u] = time
foreach v ∈ G.adj(u) do
    if dt[v] == 0 then
        if hasCycleRec(G, v, time, dt, ft) then
            return true
    else if dt[u] > dt[v] and ft[v] == 0 then
        return true
time = time + 1;   ft[u] = time
return false
```

Applicazione DFS: DAG



Arco dell'albero	$dt[v] == 0$
Arco all'indietro:	$dt[u] > dt[v]$ **and** $ft[v] = 0$
Arco in avanti:	$dt[u] < dt[v]$ **and** $ft[v] \neq 0$
Arco attraversamento:	altrimenti

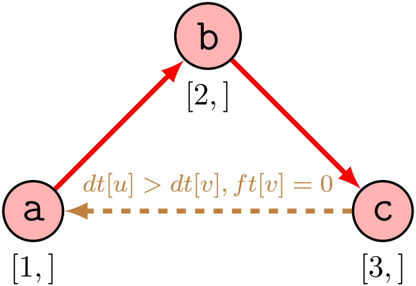
Applicazione DFS: DAG

Non viene individuato nessun arco all'indietro, quindi tutte le chiamate ricorsive arriveranno al termine e ritorneranno **false**.

boolean hasCycleRec(GRAPH G, NODE u, **int** &$time$, **int**[] dt, **int**[] ft)

```
time = time + 1;   dt[u] = time
foreach v ∈ G.adj(u) do
    if dt[v] == 0 then
        if hasCycleRec(G, v, time, dt, ft) then
            return true
    else if dt[u] > dt[v] and ft[v] == 0 then
        return true
time = time + 1;   ft[u] = time
return false
```

Applicazione DFS: DAG



Arco dell'albero	$dt[v] == 0$
Arco all'indietro:	$dt[u] > dt[v]$ **and** $ft[v] = 0$
Arco in avanti:	$dt[u] < dt[v]$ **and** $ft[v] \neq 0$
Arco attraversamento:	altrimenti

Applicazione DFS: DAG

Viene individuato un arco all'indietro, che causa la restituzione di **true** in una chiamata e la conseguente restituzione di **true** da parte di tutte le chiamate ricorsive precedenti.

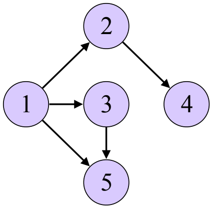
```
boolean hasCycleRec(GRAPH G, NODE u, int &time, int[] dt, int[] ft)
time = time + 1;    dt[u] = time
foreach v ∈ G.adj(u) do
    if dt[v] == 0 then
        if hasCycleRec(G, v, time, dt, ft) then
            return true
    else if dt[u] > dt[v] and ft[v] == 0 then
        return true
time = time + 1;    ft[u] = time
return false
```

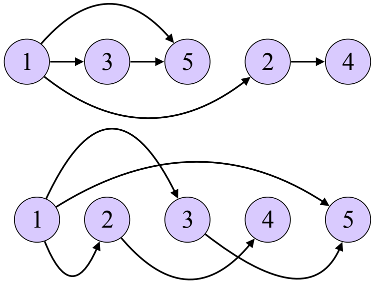
Ordinamento topologico

Definizione

Dato un DAG G, un ordinamento topologico di G è un ordinamento lineare dei suoi nodi tale che se $(u, v) \in E$, allora u appare prima di v nell'ordinamento.

- Esistono più ordinamenti topologici
- Se il grafo contiene un ciclo, non esiste un ordinamento topologico.





Ordinamento topologico

Problema

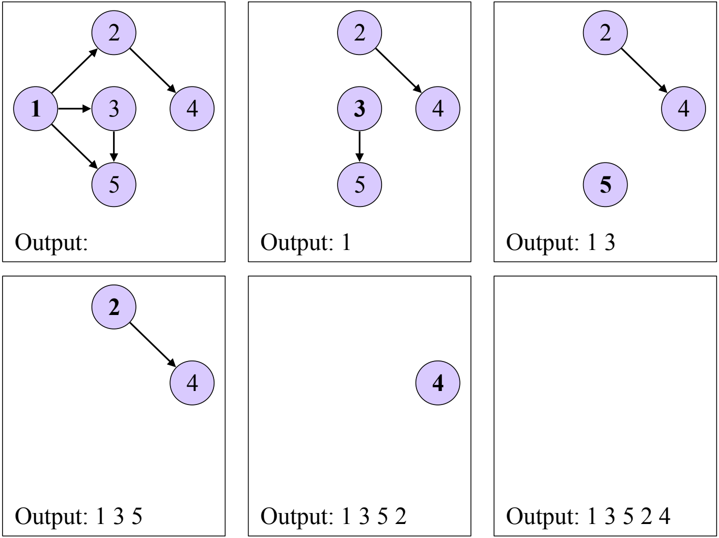
Scrivere un algoritmo che prende in input un DAG e ritorna un ordinamento topologico per esso.

Naive solution

- Trovare un nodo senza archi entranti
- Aggiungere questo nodo nell'ordinamento e rimuoverlo, insieme a tutti i suoi archi
- Ripetere questa procedura fino a quando tutti i nodi sono stati rimossi

Arthur B. Kahn. *Topological sorting of large networks.* Communications of the ACM, 5(11):558–562, 1962.

Ordinamento topologico - Algoritmi naive



Ordinamento topologico basato su DFS

Algoritmo

- DFS dove l'operazione di visita consiste nell'aggiungere il nodo in testa ad una lista, "a tempo di fine" (post-ordine)
- Restituire la lista così ottenuta.

Output

- La sequenza dei nodi, ordinati per tempo decrescente di fine.

Perchè funziona?

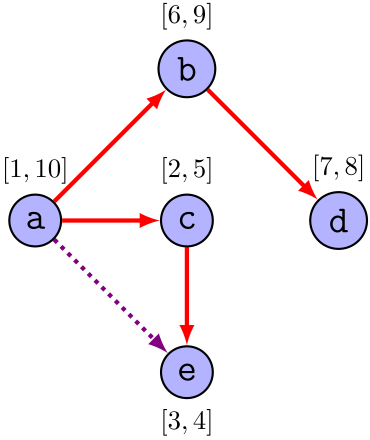
- Quando un nodo è "finito", tutti i suoi discendenti sono stati scoperti e aggiunti alla lista.
- Aggiungendolo in testa alla lista, il nodo è in ordine corretto.

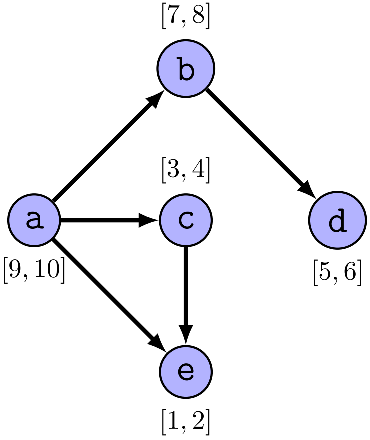
Ordinamento topologico - L'algoritmo

```
STACK topSort(GRAPH G)
---------------------------------
STACK S = Stack()
boolean[] visited = boolean[G.size()]
foreach u ∈ G.V() do visited[u] = false
foreach u ∈ G.V() do
    if not visited[u] then
        ts-dfs(G, u, visited, S)
return S
```

```
ts-dfs(GRAPH G, NODE u, boolean[] visited, STACK S)
---------------------------------
visited[u] = true
foreach v ∈ G.adj(u) do
    if not visited[v] then
        ts-dfs(G, v, visited, S)
S.push(u)
```

Ordinamento topologico – Esempio





`Stack = { a, b, d, c, e }`

`Stack = { a, b, d, c, e }`

Reality check

Applicazioni dell'ordinamento topologico

- Ordine di valutazione delle celle in uno spreadsheet
- Ordine di compilazione in un `Makefile`
- Risoluzione delle dipendenze nei linker
- Risoluzione delle dipendenze nei gestori di pacchetti software

Grafi e componenti fortemente connessi

Definizioni

- Un grafo orientato $G = (V, E)$ è fortemente connesso $\Leftrightarrow$ ogni suo nodo è raggiungibile da ogni altro suo nodo
- Un grafo $G' = (V', E')$ è una componente fortemente connessa di $G \Leftrightarrow G'$ è un sottografo connesso e massimale di G

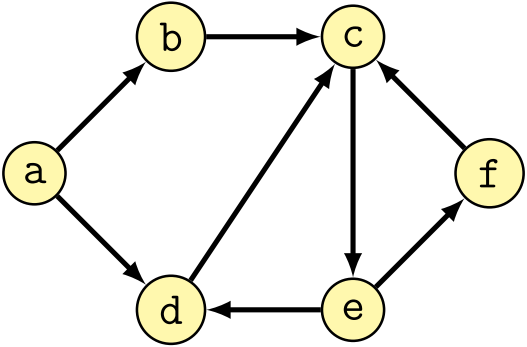
Repetita iuvant

- G' è un sottografo di G $(G' \subseteq G) \Leftrightarrow V' \subseteq V$ e $E' \subseteq E$
- G' è massimale $\Leftrightarrow \nexists$ un altro sottografo G'' di G tale che:
  - G'' è fortemente connesso
  - G'' è più grande di G' (i.e. $G' \subseteq G'' \subseteq G$)

Connessione forte

Domanda

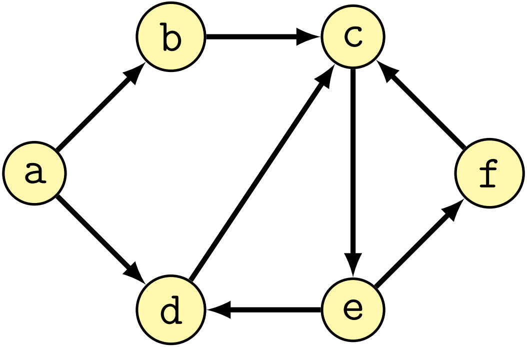
Questo grafo è fortemente connesso? No



Componenti fortemente connesse

Domanda

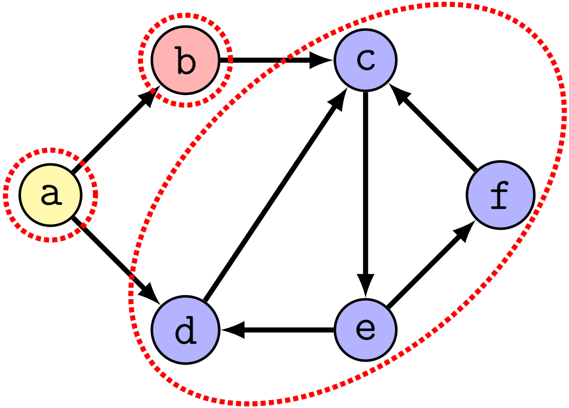
Quali sono le componenti fortemente connesse di questo grafo?



Componenti fortemente connesse

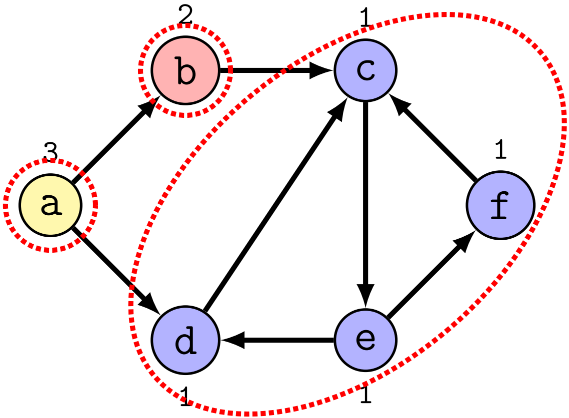
Domanda

Quali sono le componenti fortemente connesse di questo grafo?



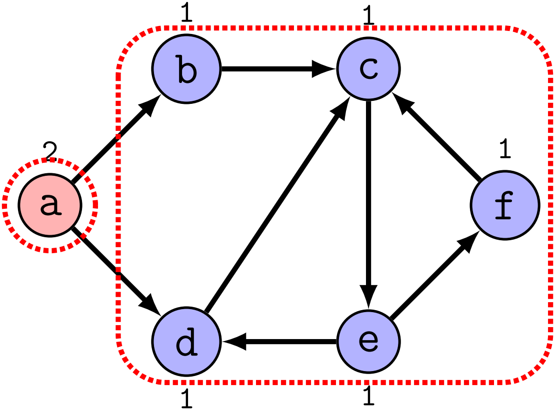
Soluzione "ingenua" (e non corretta)

- Si applica l'algoritmo cc() al grafo
- Purtroppo, il risultato dipende dal nodo di partenza



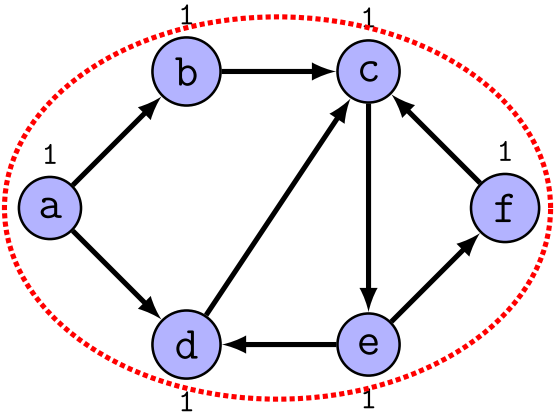
Soluzione "ingenua" (e non corretta)

- Si applica l'algoritmo cc() al grafo
- Purtroppo, il risultato dipende dal nodo di partenza



Soluzione "ingenua" (e non corretta)

- Si applica l'algoritmo cc() al grafo
- Purtroppo, il risultato dipende dal nodo di partenza



Algoritmo di Kosaraju

Kosaraju Algorithm (1978)

- Effettua una visita DFS del grafo G
- Calcola il grafo trasposto G_t
- Esegui una visita DFS sul grafo G_t utilizzando cc, esaminando i nodi nell'ordine inverso di tempo di fine della prima visita
- Le componenti connesse (e i relativi alberi DF) rappresentano le componenti fortemente connesse di G

int[] scc(Graph G)

```
Stack S = topSort(G)          % First visit
G^T = transpose(G)            % Graph transposal
return cc(G^T, S)             % Second visit
```

Ordinamento topologico su grafi generali

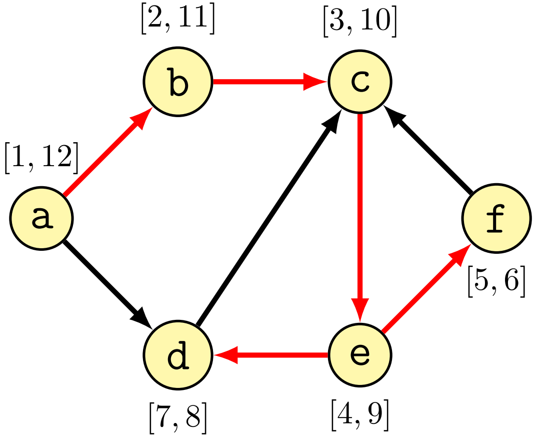
Idea generale

Applicando l'algoritmo di ordinamento topologico su un grafo generale, siamo sicuri che:

- se un arco (u, v) non appartiene ad un ciclo, allora u viene listato prima di v nella sequenza ordinata
- gli archi di un ciclo vengono listati in qualche ordine, ininfluente

Utilizziamo quindi `topsort()` per ottenere i nodi in ordine decrescente di tempo di fine

Esecuzione 1: Ordinamento topologico



Stack = { a, b, c, e, d, f }

Calcolo del grafo trasposto

Grafo trasposto (Transpose graph)

Dato un grafo orientato $G = (V, E)$, il grafo trasposto $G_t = (V, E_T)$ ha gli stessi nodi e gli archi orientati in senso opposto.:

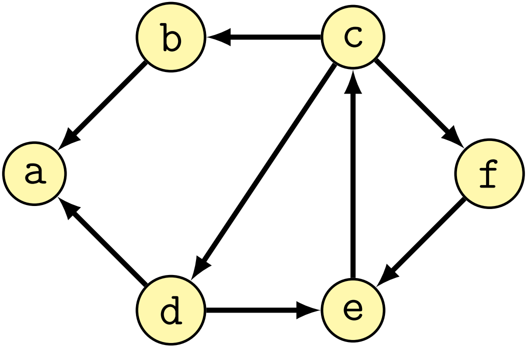
$$E_T = \{(u, v) \mid (v, u) \in E\}$$

```
Graph transpose(Graph G)
Graph G^T = Graph()
foreach u ∈ G.V() do
  G^T.insertNode(u)
foreach u ∈ G.V() do
  foreach v ∈ G.adj(u) do
    G^T.insertEdge(v, u)
return G^T
```

Costo computazionale: $O(m+n)$

- $O(n)$ nodi aggiunti
- $O(m)$ archi aggiunti
- Ogni operazione costa $O(1)$

Esecuzione 1: Grafo trasposto



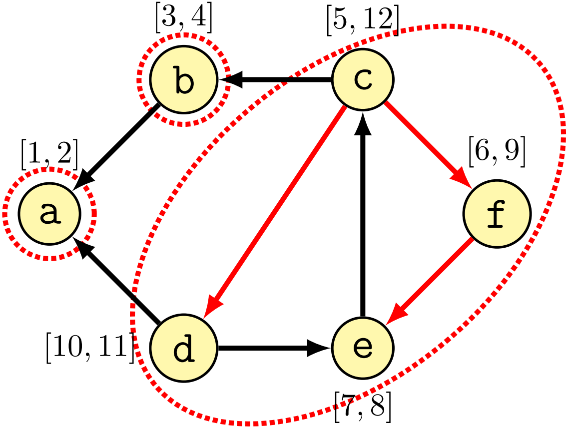
Calcolo delle componenti connesse

Invece di esaminare i nodi in ordine arbitrario, questa versione di `cc()` li esamina nell'ordine LIFO memorizzato nello stack.

```
cc(Graph G, Stack S)
---
int[] id = new int[G.size()]
foreach u ∈ G.V() do
    id[u] = 0
int counter = 0
while not S.isEmpty() do
    u = S.pop()
    if id[u] == 0 then
        counter = counter + 1
        ccdfs(G, counter, u, id)
return id
```

```
ccdfs(Graph G, int counter, Node u, int[] id)
---
id[u] = counter
foreach v ∈ G.adj(u) do
    if id[v] == 0 then
        ccdfs(G, counter, v, id)
```

Esecuzione 1: Componenti connesse



`Stack = { a, b, c, e, d, f }`

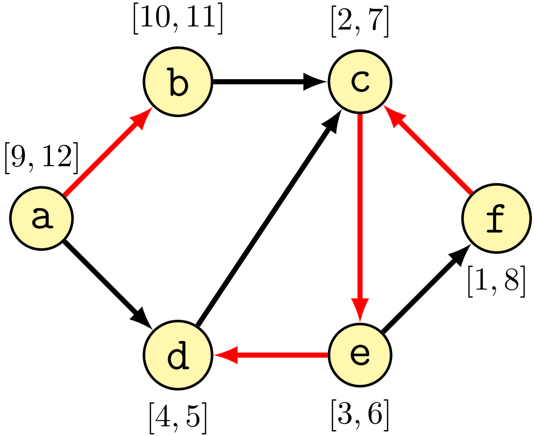
SCC: The algorithm

int[] scc(GRAPH G)	
STACK S = topSort(G)	% First visit
G^T = transpose(G)	% Graph transposal
return cc(G^T, S)	% Second visit

Costo computazionale: $O(m + n)$

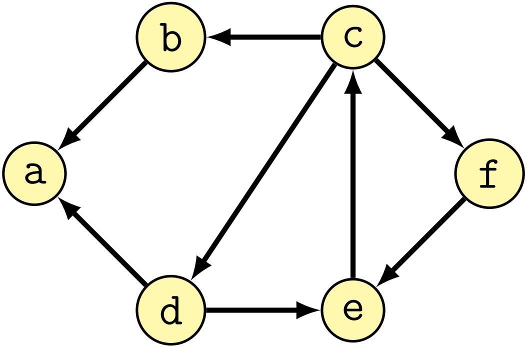
- Ogni fase richiede $O(m + n)$

Esecuzione 2: Ordinamento topologico

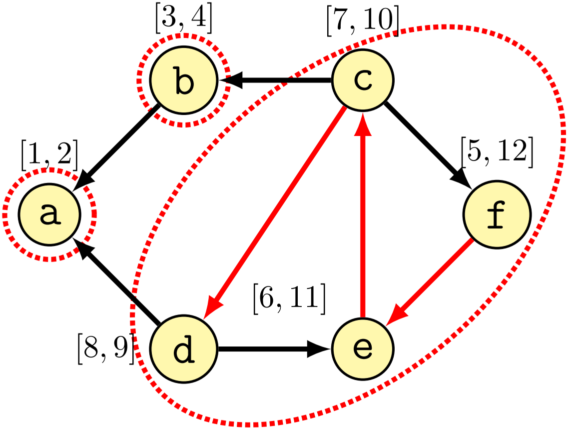


`Stack = { a, b, f, c, e, d }`

Esecuzione 2: Grafo trasposto



Esecuzione 2: Componenti connesse



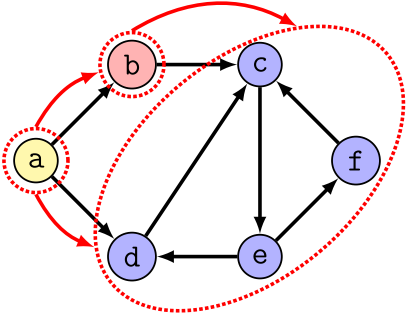
`Stack = { a, b, f, c, e, d }`

Dimostrazione di correttezza

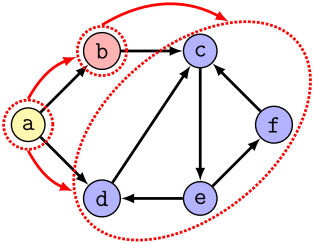
Grafo delle componenti

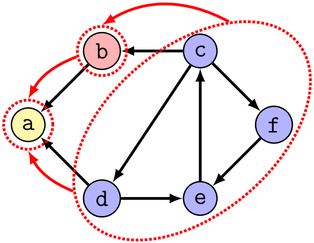
$C(G) = (V_c, E_c)$

- $V_c = \{C_1, C_2, \ldots, C_k\}$, dove C_i è la i-esima SCC of G
- $E_c = \{(C_i, C_j) | \exists (u_i, u_j) \in E : u_i \in C_i \wedge u_j \in C_j\}$



Dimostrazione di correttezza





Qual è la relazione fra il grafo delle componenti di G e il grafo delle componenti di G_T?

$C(G^T) = [C(G)]^T$

Il grafo delle componenti è aciclico?

SI

Dimostrazione di correttezza

Discovery time e finish time del grafo delle componenti

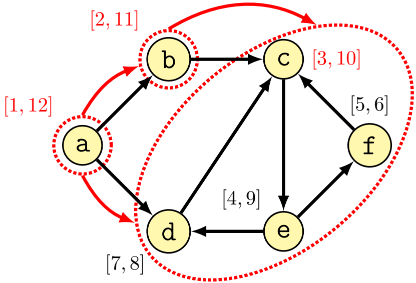
$$dt(C) = \min\{dt(u) | u \in C\}$$
$$ft(C) = \max\{ft(u) | u \in C\}$$

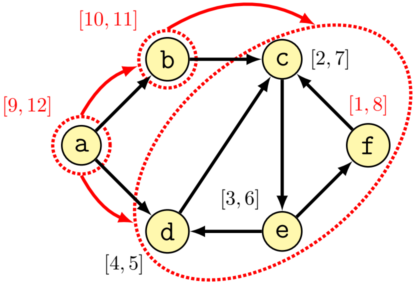
Questi discovery/finish time corrispondono a i discovery/finish time del primo nodo visitato in C

Dimostrazione di correttezza

Teorema

Siano C e C' due distinte SCCs nel grafo orientato $G = (V, E)$.
Se esiste un arco $(C, C') \in E_c$, allora $ft(C) > ft(C')$.



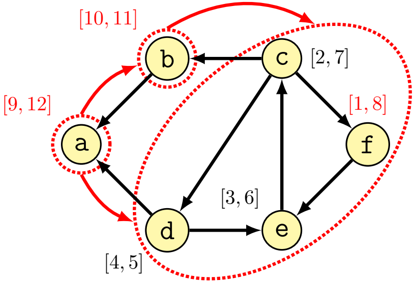


Dimostrazione di correttezza

Corollario

Siano C_x e C_y due SCC distinte nel grafo orientato $G = (V, E)$. Se esiste un arco $(x, y) \in E_t$ tale che $x \in C_x$ e $y \in C_y$, allora $ft(C_x) < ft(C_y)$.

$$(x, y) \in E_t \Rightarrow$$
$$(y, x) \in E \Rightarrow$$
$$(C_y, C_x) \in E_c \Rightarrow$$
$$ft(C_y) > ft(C_x) \Rightarrow$$
$$ft(C_x) < ft(C_y)$$

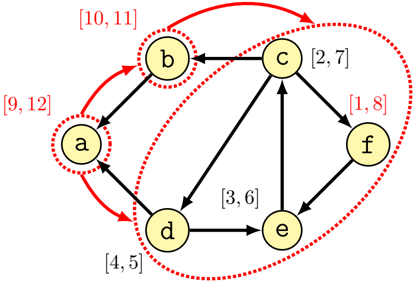


Dimostrazione di correttezza

Corollario

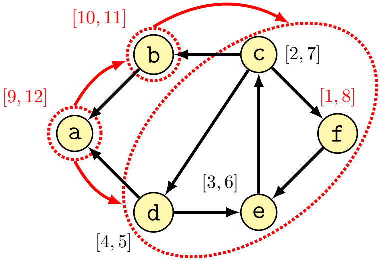
Siano C_x e C_y due SCC distinte nel grafo orientato $G = (V, E)$. Se esiste un arco $(x, y) \in E_t$ tale che $x \in C_x$ e $y \in C_y$, allora $ft(C_x) < ft(C_y)$.

$$\begin{aligned} (b, a) \in E_t &\Rightarrow \\ (a, b) \in E &\Rightarrow \\ (C_a, C_b) \in E_c &\Rightarrow \\ 12 = ft(C_a) > ft(C_b) = 11 &\Rightarrow \\ 11 = ft(C_b) < ft(C_a) = 12 & \end{aligned}$$



Dimostrazione di correttezza

- Se la componente C_x e la componente C_y sono connesse da un arco $(x, y) \in E_t$, allora:
  - Dal corollario, $ft(C_x) < ft(C_y)$
  - Dall'algoritmo, la visita di C_y inizierà prima della visita di C_x
- Non esistono cammini tra C_y e C_x in G_t (altrimenti il grafo sarebbe ciclico)
  - Dall'algoritmo, la visita di C_y non raggiungerà C_x,



In altre parole, `cc()` assegnerà correttamente gli identificatori delle componenti ai nodi.

Reality check

Algoritmo di Tarjan (1972)

- Tarjan, R. E. "Depth-first search and linear graph algorithms", SIAM Journal on Computing 1(2): 146–160 (1972)
- Algoritmo con costo $O(m+n)$ come Kosaraju
- È preferito a Kosaraju in quanto necessita di una sola visita e non richiede il grafo trasposto

Applicazioni

Gli algoritmi per SCC possono essere utilizzati per risolvere il problema 2-satisfiability (2-SAT), un problema di soddisfacibilità booleana con clausole composte da coppie di letterali.

Conclusioni

113 Pages in category "Graph algorithms"

A

- A* search algorithm
- Algorithmic version for Szemerédi regularity partition
- Alpha–beta pruning
- Aperiodic graph

B

- B*
- Barabási–Albert model
- Belief propagation
- Bellman–Ford algorithm
- Bianconi–Barabási model
- Bidirectional search
- Borůvka's algorithm
- Bottleneck traveling salesman problem
- Breadth-first search
- Bron–Kerbosch algorithm
- Bully algorithm

C

- Centrality
- Chaitin's algorithm
- Christofides algorithm
- Clique percolation method
- Closure problem
- Color-coding
- Contraction hierarchies
- Courcelle's theorem
- Cuthill–McKee algorithm

D

- D*
- Degeneracy (graph theory)
- Depth-first search
- Dijkstra–Scholten algorithm
- Dijkstra's algorithm
- Dinic's algorithm
- Disparity filter algorithm of weighted network
- Double pushout graph rewriting
- Dulmage–Mendelsohn decomposition
- Dynamic connectivity
- Dynamic link matching

E

- Edmonds–Karp algorithm
- Edmonds' algorithm
- Blossom algorithm
- Euler tour technique

F

- FKT algorithm
- Flooding algorithm
- Floyd–Warshall algorithm
- Force-directed graph drawing
- Ford–Fulkerson algorithm
- Fringe search

G

- Girvan–Newman algorithm
- Goal node (computer science)
- Gomory–Hu tree
- Graph bandwidth
- Graph edit distance
- Graph embedding
- Graph isomorphism
- Graph isomorphism problem
- Graph kernel
- Graph reduction
- Graph traversal

H

- Havel–Hakimi algorithm
- Hierarchical closeness
- Hierarchical clustering of networks
- Hopcroft–Karp algorithm

I

- Iterative deepening A*
- Initial attractiveness
- Iterative compression
- Iterative deepening depth-first search

J

- Johnson's algorithm
- Journal of Graph Algorithms and Applications
- Jump point search
- Junction tree algorithm

K

- K shortest path routing
- Karger's algorithm
- Kleitman–Wang algorithms
- Knight's tour
- Knuth's Simpath algorithm
- Kosaraju's algorithm
- Kruskal's algorithm

L

- Lexicographic breadth-first search
- Longest path problem

M

- MaxCliqueDyn maximum clique algorithm
- Minimax
- Minimum bottleneck spanning tree
- Misra & Gries edge coloring algorithm

N

- Nearest neighbour algorithm
- Network flow problem
- Network simplex algorithm
- Nonblocking minimal spanning switch

P

- PageRank
- Parallel all-pairs shortest path algorithm
- Path-based strong component algorithm
- Pre-topological order
- Prim's algorithm
- Proof-number search
- Push–relabel maximum flow algorithm

R

- Reverse-delete algorithm
- Rocha–Thatte cycle detection algorithm

S

- Sethi–Ullman algorithm
- Shortest Path Faster Algorithm
- SMA*
- Spectral layout
- Spreading activation
- Stoer–Wagner algorithm
- Subgraph isomorphism problem
- Suurballe's algorithm

T

- Tarjan's off-line lowest common ancestors algorithm
- Tarjan's strongly connected components algorithm
- Theta*
- Topological sorting
- Transitive closure
- Transitive reduction
- Travelling salesman problem
- Tree traversal

W

- Widest path problem
- Wiener connector

Y

- Yen's algorithm