

Grafi: visite

Corso di **Algoritmi e strutture dati**
Corso di Laurea in **Informatica**
Docenti: Ugo de'Liguoro, András Horváth

Indice

1. Visita in generale
2. Visita in ampiezza
3. Visita in profondità

Sommario

Obiettivo:

- ▶ visitare tutti nodi e archi in modo sistematico
- ▶ problema di base in molte applicazioni
- ▶ la visita fornisce informazione sul grafo visitato
- ▶ vari tipi di visite:
 - ▶ in ampiezza, breadth first search
 - ▶ in profondità, depth first search

1. Idea di base

- ▶ insieme di vertici diviso in tre sottoinsiemi:
 - ▶ **bianco**: nodi non ancora scoperti (non visitati)
 - ▶ **grigio**: vertici scoperti di cui adiacenti non sono ancora tutti scoperti (nodi da cui bisogna andare ancora avanti; la frangia)
 - ▶ **nero**: nodi scoperti di cui adiacenti sono già stati scoperti (nodi da cui non bisogna andare avanti più)

1. Versione “astratta” dell’algoritmo

- ▶ INIZIALIZZA(G)
 for $\forall u \in V$ **do**
 $u.color \leftarrow \text{bianco}$
- ▶ VISITA(G, s)
 $s.color \leftarrow \text{grigio}$
 while \exists nodo *grigio* **do**
 $u \leftarrow$ nodo *grigio*
 if $\exists v \text{ bianco} \in \text{adj}[u]$ **then**
 $v.color \leftarrow \text{grigio}$
 else
 $u.color \leftarrow \text{black}$

1. Proprietà, invarianti

- ▶ **proprietà I**: colore di un nodo può solo passare da bianco a grigio a nero
- ▶ **invariante I**: se $(u, v) \in E$ e u è nero, allora v è grigio o nero
- ▶ **invariante II**: tutti i vertici grigi o neri sono raggiungibili da s
- ▶ **invariante III**: qualunque cammino da s ad un nodo bianco deve contenere almeno un vertice grigio

1. Proprietà, invarianti

- ▶ invariante I e II sono evidenti
- ▶ dimostrazione di invariante III:
 - ▶ se s è ancora grigio: vero
 - ▶ se s è nero: se non ci fosse nessun vertice grigio, allora ci sarebbe un nodo bianco adiacente ad uno nero, che è impossibile per l'invariante I

1. Al termine del algoritmo

- ▶ teorema: al termine dell'algoritmo
 un vertice v è nero $\iff v$ è raggiungibile da s
- ▶ \Rightarrow : dall'invariante II
- ▶ \Leftarrow : invariante III e la condizione di uscita del ciclo implicano che non ci può essere nessun vertice bianco raggiungibile da s

1. Costruzione del sottografo di predecessori

- ▶ per ogni vertice che viene scoperto, vogliamo ricordare quale vertice grigio ha permesso di scoprirlo
- ▶ vuole dire anche ricordare l'arco che ci porta alla scoperta del nodo
- ▶ associamo ad ogni nodo un attributo (variabile) che memorizza il nodo dal quale era scoperto
- ▶ inizializzazione dell'algoritmo:
INIZIALIZZA(G)
 for $\forall u \in V$ **do**
 $u.color \leftarrow bianco$
 $u.\pi \leftarrow nil$

1. Costruzione del sottografo di predecessori

- ▶ VISITA(G, s)
 $s.color \leftarrow grigio$
 while \exists nodo *grigio* **do**
 $u \leftarrow$ nodo *grigio*
 if $\exists v$ *bianco* $\in adj[u]$ **then**
 $v.color \leftarrow grigio$
 $v.\pi \leftarrow u$
 else
 $u.color \leftarrow black$
- ▶ proprietà II: al termine di VISITA(G, s), l'unico vertice nero con predecessore nil è s

1. Sottografo di predecessori

- ▶ sottografo: $G_\pi = (V_\pi, E_\pi)$
- ▶ $V_\pi = \{v \in V : v.\pi \neq nil\} \cup \{s\}$
- ▶ $E_\pi = \{(v.\pi, v) \in E : v \in V_\pi - \{s\}\}$
- ▶ all'termine di VISITA(G, s), V_π è l'insieme di tutti i vertici neri (tutti i nodi raggiungibili da s)
- ▶ teorema: il sottografo di predecessori è un albero
- ▶ dimostrazione: segue dal seguente invariante del while: $G_\pi = (V_\pi, E_\pi)$ è connesso e $|E_\pi| = |V_\pi| - 1$
- ▶ il sottografo $G_\pi = (V_\pi, E_\pi)$ è l'albero di scoperta

1. Visitare grafi non connessi

- ▶ l'algoritmo precedente visita solo il componente di cui il nodo iniziale (s) fa parte
- ▶ visita tutto il grafo solo se il grafo è connesso
- ▶ visita intera di un grafo non connesso:
 VISITA-TUTTI-VERTICI(G)
 INIZIALIZZA(G)
 for $\forall u \in V$ **do**
 if $u.color = bianco$ **then**
 VISITA(G, u)
- ▶ il sottografo di predecessori diventa una foresta (un grafo composto da più alberi) se il grafo contiene più componenti
- ▶ questa foresta si chiama **foresta di scoperta**

1. Cammino di scoperta

- ▶ algoritmo che stampa il cammino dal sorgente della visita ad un nodo u

PRINT-PATH(G, s, u)

if $u = s$ **then**

 stampa u

else

if $u.\pi = \text{nil}$ **then**

 stampa “non esiste cammino da s ad u ”

else

 PRINT-PATH($G, s, u.\pi$)

 stampa u

1. Versione “concreta” dell'algoritmo

- ▶ una struttura dati D per gestire l'insieme di vertici grigi
- ▶ operazioni necessari:
 - ▶ MAKE-EMPTY: crea una struttura nuova
 - ▶ FIRST(D): restituisce il primo elemento (senza modificare D)
 - ▶ ADD(D, x): aggiunge l'elemento x a D
 - ▶ REMOVE-FIRST(D): toglie da D il primo elemento
 - ▶ NOT-EMPTY(D): restituisce true se D non è vuoto, false altrimenti
- ▶ ruolo di ADD(D, x):
 - ▶ aggiunge x come primo elemento → pila (stack)
 - ▶ aggiunge x come ultimo elemento → coda (queue)

1. Versione “concreta” dell'algoritmo

VISITA(G, s)

$D \leftarrow \text{MAKE-EMPTY}$

$s.\text{color} \leftarrow \text{grigio}$

 ADD(D, s)

while NON-EMPTY(D) **do**

$u \leftarrow \text{FIRST}(D)$

if $\exists v \text{ bianco} \in \text{adj}[u]$ **then**

$v.\text{color} \leftarrow \text{grigio}$

$v.\pi \leftarrow u$

 ADD(D, v)

else

$u.\text{color} \leftarrow \text{black}$

 REMOVE-FIRST(D)

1. Complessità della visita

- ▶ bisogna sapere come viene implementato il test:
 - if** $\exists v \text{ bianco} \in \text{adj}[u]$
- ▶ assumiamo di aver realizzato il test con ciclo che percorre dall'inizio la lista di adiacenza
- ▶ ciclo finisce quando si trova il primo nodo bianco
- ▶ problema: la lista di adiacenza di un nodo u può essere percorso più volte

1. Complessità della visita

- ▶ dalla proprietà I sappiamo che un vertice grigio o nero non può ridiventare bianco
- ▶ non è necessario percorrere la lista di adiacenza dal inizio
- ▶ il ciclo può partire da dove è arrivato l'ultima volta
- ▶ associamo ad ogni vertice il valore corrente del puntatore alla lista di adiacenti
- ▶ la lista di adiacenza è percorsa una volta sola → implementazione più efficiente

1. Complessità della visita

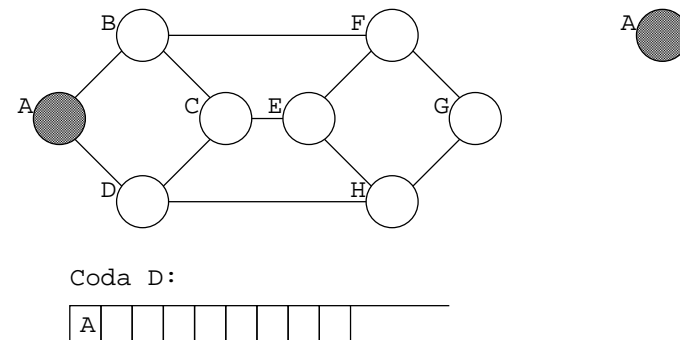
- ▶ inizializzazione richiede tempo $O(|V|)$
- ▶ ogni vertice viene inserito (eliminato) una volta in (da) D
- ▶ assumiamo che le operazioni di inserimento e eliminazione richiedano un tempo costante
- ▶ tempo totale dedicato alle operazioni su D è $O(|V|)$
- ▶ le liste di adiacenza vengono percorse una volta
- ▶ tempo totale dedicato alla ricerca di nodi bianchi è $O(|E|)$
- ▶ **tempo totale:** $O(|V| + |E|)$

2. Visita in ampiezza

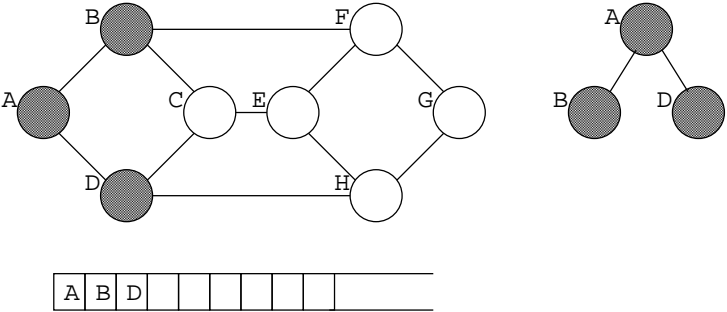
- ▶ VISITA(G, s)
 $D \leftarrow \text{MAKE-EMPTY}$
 $s.\text{color} \leftarrow \text{grigio}$
 ADD(D, s)
 while NON-EMPTY(D) **do**
 $u \leftarrow \text{FIRST}(D)$
 if $\exists v \text{ bianco} \in \text{adj}[u]$ **then**
 $v.\text{color} \leftarrow \text{grigio}$
 $v.\pi \leftarrow u$
 ADD(D, v)
 else
 $u.\text{color} \leftarrow \text{black}$
 REMOVE-FIRST(D)

- ▶ D è una **coda** → **visita in ampiezza** (breadth first search, BFS)

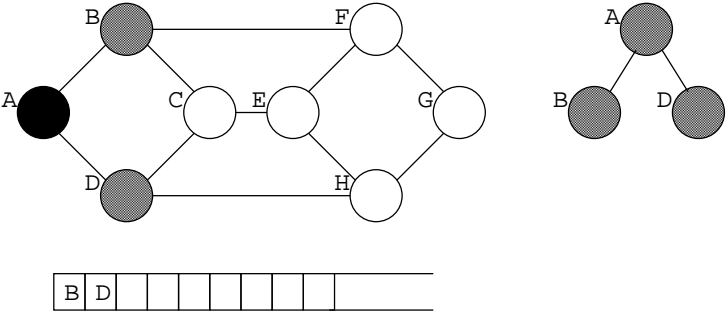
2. Esempio di BFS



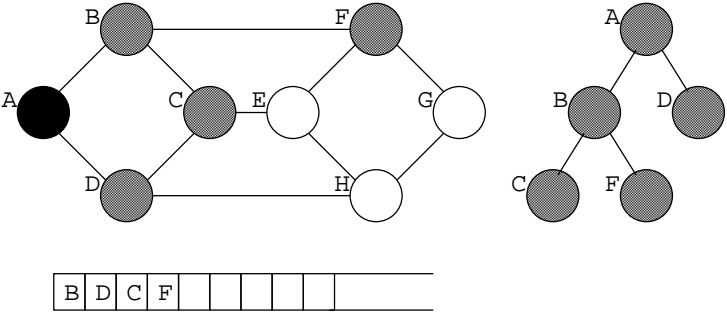
2. Esempio di BFS



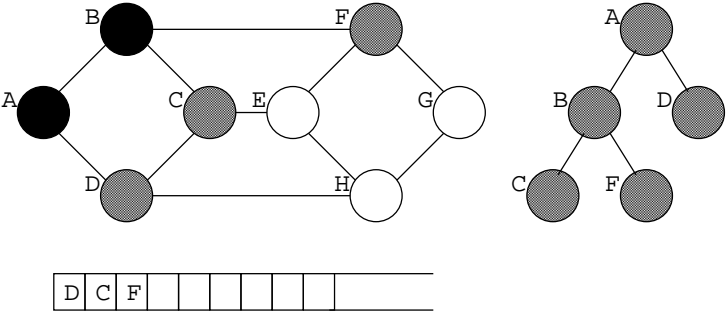
2. Esempio di BFS



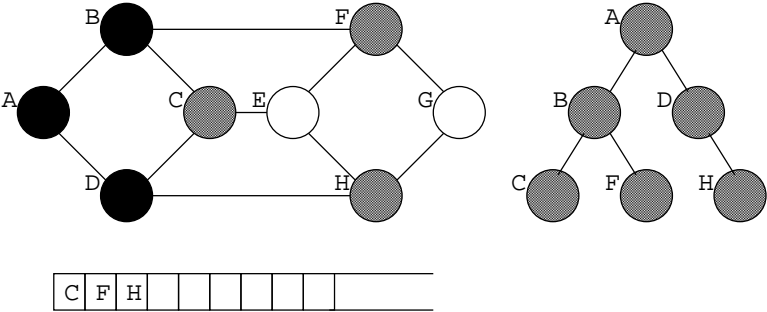
2. Esempio di BFS



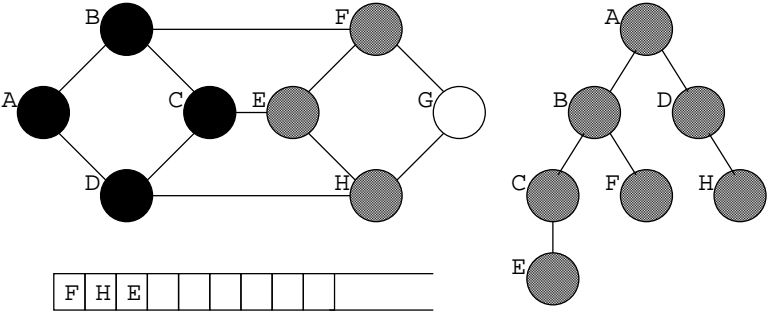
2. Esempio di BFS



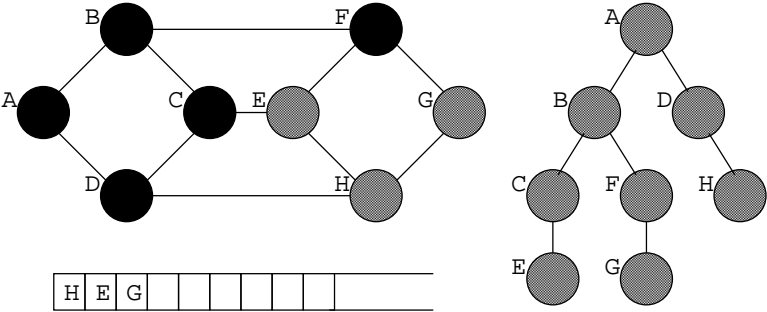
2. Esempio di BFS



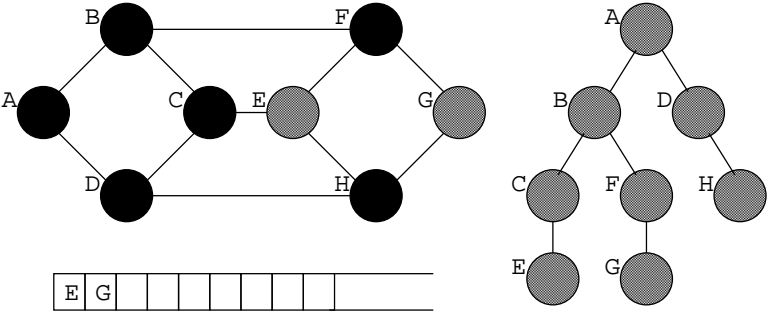
2. Esempio di BFS



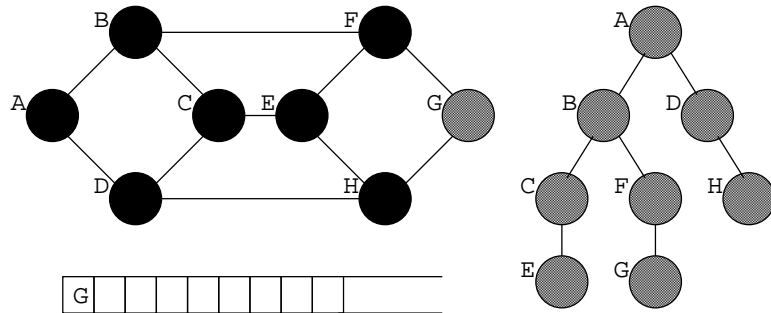
2. Esempio di BFS



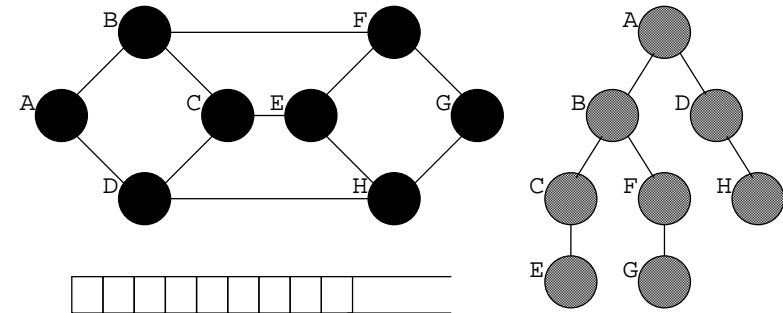
2. Esempio di BFS



2. Esempio di BFS



2. Esempio di BFS



2. Versione modificata dell'algoritmo

- ▶ il primo elemento della coda non cambia finchè ci sono adiacenti bianchi
- ▶ **VISITA(G, s)**
 - $D \leftarrow \text{MAKE-EMPTY}$
 - $s.\text{color} \leftarrow \text{grigio}$
 - $\text{ADD}(D, s)$
 - while** $\text{NON-EMPTY}(D)$ **do**
 - $u \leftarrow \text{FIRST}(D)$
 - for** $\forall v : v \text{ è bianco ed } v \in \text{adj}[u]$ **do**
 - $v.\text{color} \leftarrow \text{grigio}$
 - $v.\pi \leftarrow u$
 - $\text{ADD}(D, v)$
 - $u.\text{color} \leftarrow \text{black}$
 - $\text{REMOVE-FIRST}(D)$

2. Terza versione (ancora più “concreta”)

Ogni elemento nella lista di adiacenti ha due campi:

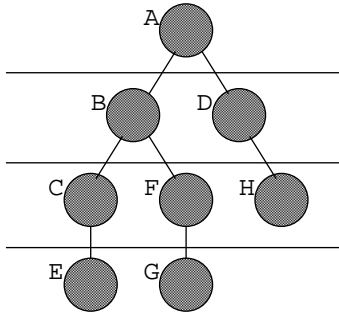
- ▶ vtx è la vertice
- ▶ $next$ è il prossimo elemento nella lista

```

VISITA(G, s)
D ← MAKE-EMPTY
s.color ← grigio
ADD(D, s)
while NON-EMPTY(D) do
    u ← FIRST(D)
    ptr ← adj[u]
    while ptr ≠ nil do
        v ← ptr.vtx
        if v.color = bianco then
            v.color ← grigio
            v.π ← u
            ADD(D, v)
        ptr ← ptr.next
    u.color ← black
    REMOVE-FIRST(D)
    
```


2. Albero di BFS

- ▶ albero di BFS viene costruito a livelli
- ▶ la costruzione del livello $n + 1$ non comincia prima di concludere la costruzione del livello n



- ▶ associamo ad ogni nodo un attributo (d) che ricorda il livello del nodo

2. Albero di BFS

- ▶ algoritmo di visita BFS col calcolo del livello :
- ▶ INIZIALIZZA(G)
for $\forall u \in V$ **do**
 $u.color \leftarrow bianco$
 $u.\pi \leftarrow nil$
 $u.d \leftarrow \infty$

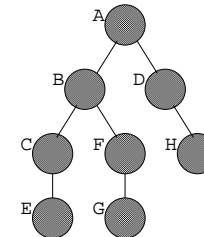
2. Albero di BFS

```

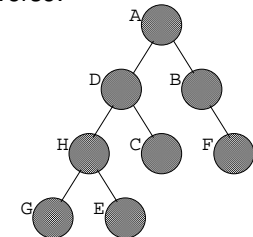
VISITA( $G, s$ )
 $D \leftarrow \text{MAKE-EMPTY}$ 
 $s.color \leftarrow grigio$ 
 $s.d \leftarrow 0$ 
ADD( $D, s$ )
while NON-EMPTY( $D$ ) do
   $u \leftarrow \text{FIRST}(D)$ 
  for  $\forall v : v \text{ è bianco ed } v \in \text{adj}[u]$  do
     $v.color \leftarrow grigio$ 
     $v.\pi \leftarrow u$ 
     $v.d \leftarrow u.d + 1$ 
    ADD( $D, v$ )
   $u.color \leftarrow black$ 
  REMOVE-FIRST( $D$ )
    
```

2. Albero di BFS

- ▶ albero dipende dal ordine in cui i nodi sono elencati nelle liste di adiecenza:
- lista di adiecenza in ordine alfabetico:

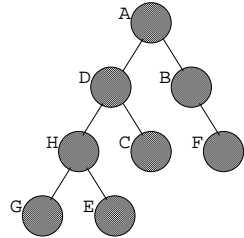
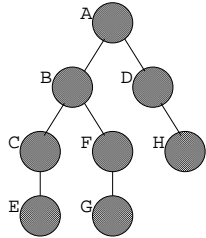


- lista di adiecenza in ordine inverso:



2. Proprietà della visita BFS

Ma ogni nodo rimane allo stesso livello:



► teorema: al termine della visita BFS

$$\forall v \in V, v.d = \delta(s, v)$$

dove $\delta(s, v)$ indica la distanza di v dal sorgente s della visita (lunghezza di cammino minimo)

2. Proprietà della visita BFS

- per ogni vertice v raggiungibile da s , il cammino da s a v nell'albero BFS è un cammino minimo
- il livello di un nodo nell'albero ottenuto con la visita BFS è indipendente dal ordine in cui sono memorizzati i vertici nelle liste di adiacenza

3. Visita in profondità

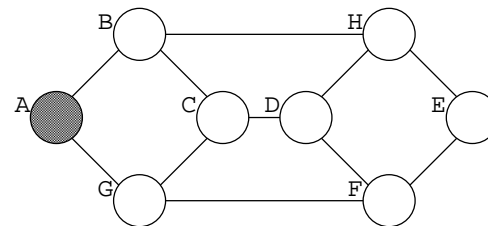
```

► VISITA( $G, s$ )
   $D \leftarrow \text{MAKE-EMPTY}$ 
   $s.color \leftarrow \text{grigio}$ 
   $\text{ADD}(D, s)$ 
  while  $\text{NON-EMPTY}(D)$  do
     $u \leftarrow \text{FIRST}(D)$ 
    if  $\exists v \text{ bianco} \in \text{adj}[u]$  then
       $v.color \leftarrow \text{grigio}$ 
       $v.\pi \leftarrow u$ 
       $\text{ADD}(D, v)$ 
    else
       $u.color \leftarrow \text{black}$ 
       $\text{REMOVE-FIRST}(D)$ 

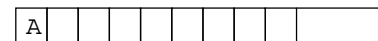
```

► D è una **pila** → **visita in profondità** (depth first search, DFS)

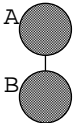
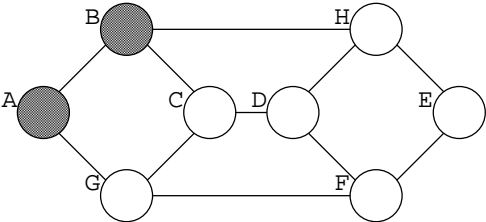
3. Esempio di DFS



Pila:



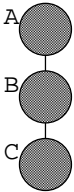
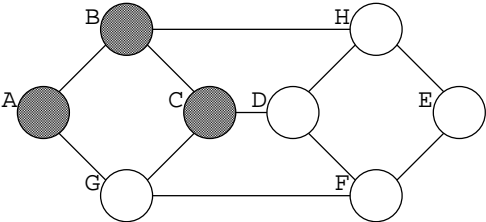
3. Esempio di DFS



Pila D:

A	B							
---	---	--	--	--	--	--	--	--

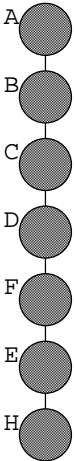
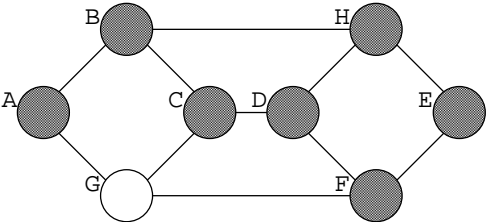
3. Esempio di DFS



Pila D:

A	B	C						
---	---	---	--	--	--	--	--	--

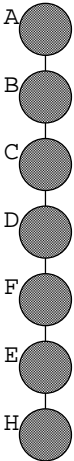
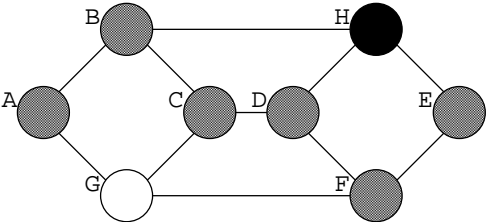
3. Esempio di DFS



Pila D:

A	B	C	D	F	E	H		
---	---	---	---	---	---	---	--	--

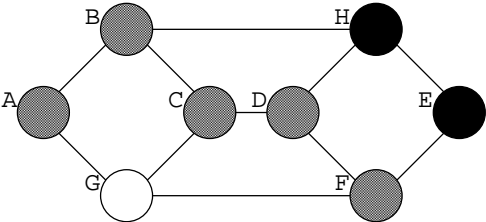
3. Esempio di DFS



Pila D:

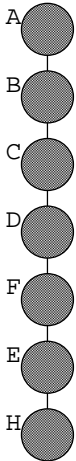
A	B	C	D	F	E			
---	---	---	---	---	---	--	--	--

3. Esempio di DFS

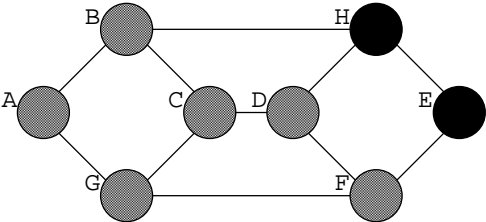


Pila D:

A	B	C	D	F				
---	---	---	---	---	--	--	--	--

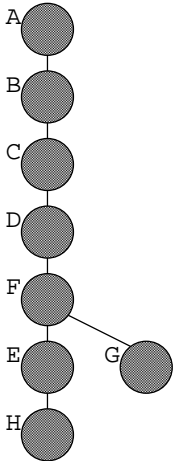


3. Esempio di DFS

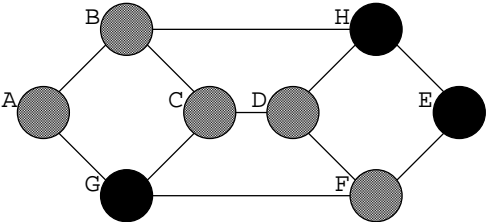


Pila D:

A	B	C	D	F	G			
---	---	---	---	---	---	--	--	--

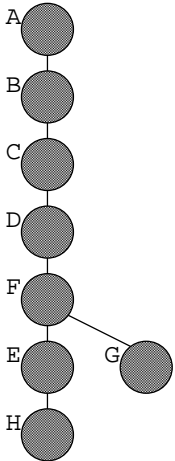


3. Esempio di DFS

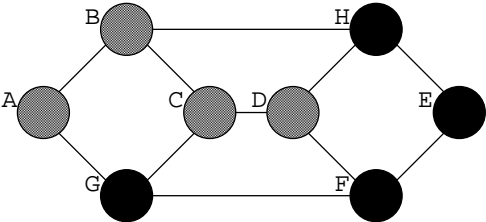


Pila D:

A	B	C	D	F				
---	---	---	---	---	--	--	--	--

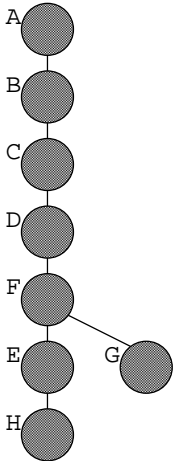


3. Esempio di DFS

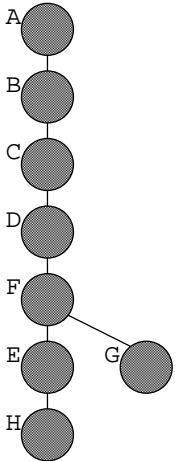
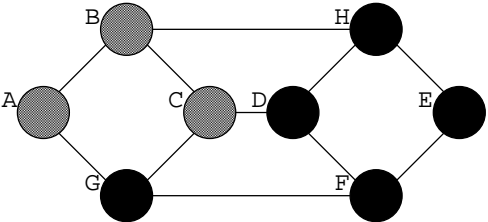


Pila D:

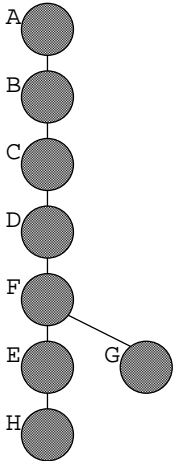
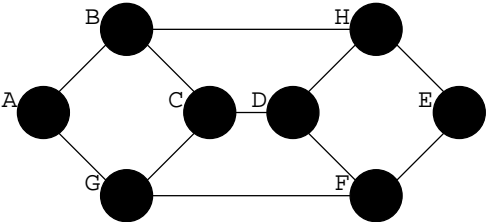
A	B	C	D					
---	---	---	---	--	--	--	--	--



3. Esempio di DFS

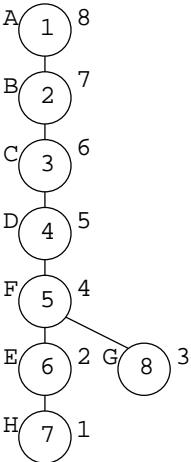
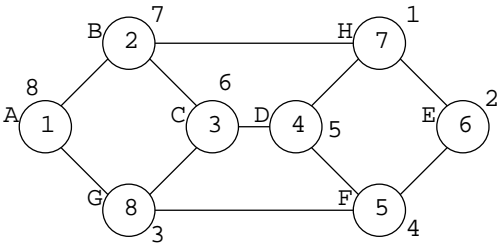


3. Esempio di DFS



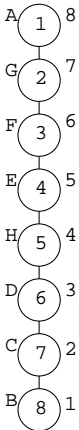
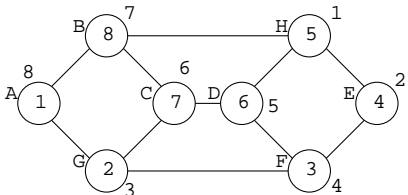
3. Ordine di scoperta di nodi

- ▶ due ordini tra i nodi:
 - ▶ l'ordine in cui i nodi diventano grigi (scritto nel nodo)
 - ▶ l'ordine in cui i nodi diventano neri (scritto accanto al nodo)
- ▶ liste di adiacenza in ordine alfabetico



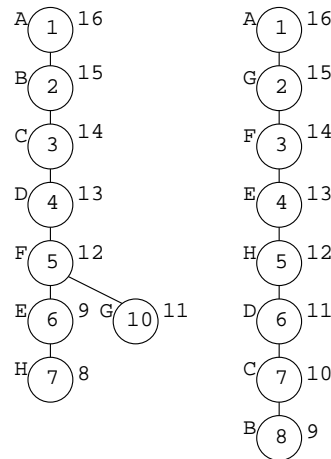
3. Ordine di scoperta di nodi

- ▶ liste di adiacenza in ordine contrario



3. Ordine di scoperta di nodi

- ▶ un unico contatore che viene incrementato quando un nodo cambia colore
- ▶ ogni nodo viene marcato due volte con questo numero (bianco \rightarrow grigio, grigio \rightarrow nero)



3. Versione sbagliata (non funziona)

- ```

▶ VISITA(G, s)
 D \leftarrow MAKE-EMPTY
 s.color \leftarrow grigio
 ADD(D, s)
 while NON-EMPTY(D) do
 u \leftarrow FIRST(D)
 for $\forall v : v$ è bianco ed è $\in \text{adj}[u]$ do
 v.color \leftarrow grigio
 v. π \leftarrow u
 ADD(D, v)
 u.color \leftarrow black
 REMOVE-FIRST(D)
▶ perchè non funziona?

```

### 3. Perchè non funziona?

- ▶ se  $v$  è bianco, diventa grigio e finisce sul top di  $D$
- ▶ la lista di adiacenti da considerare dovrebbe essere quella del nuovo nodo sul top di  $D$
- ▶ invece nella condizione del **for** rimane il vertice precedente

### 3. Una versione corretta

```

VISITA(G, s)
 D \leftarrow MAKE-EMPTY
 s.color \leftarrow grigio
 ADD(D, s)
 while NON-EMPTY(D) do
 while $\exists v : v$ è bianco ed è $\in \text{adj}[\text{FIRST}(D)]$ do
 v.color \leftarrow grigio
 v. π \leftarrow FIRST(D)
 ADD(D, v)
 u.color \leftarrow black
 REMOVE-FIRST(D)

```

### 3. E una versione corretta e più concreta

```

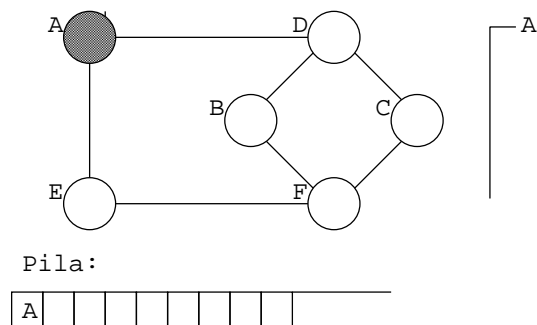
VISITA(G, s)
 D ← MAKE-EMPTY
 s.color ← grigio
 ADD(D, s)
 while NON-EMPTY(D) do
 while FIRST(D).ptr ≠ nil do
 v ← FIRST(D).ptr.vtx
 FIRST(D).ptr ← FIRST(D).ptr.next
 if v.color = bianco then
 v.color ← grigio
 v.π ← FIRST(D)
 ADD(D, v)
 FIRST(D).color ← black
 REMOVE-FIRST(D)

```

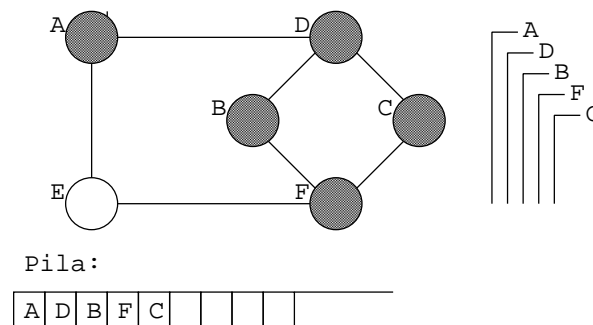
### 3. Inizializzazione

- ▶ naturalmente l'attributo *ptr* va inizializzato
- ▶ INIZIALIZZA(*G*)
  - for  $\forall u \in V$  do
    - $u.color \leftarrow bianco$
    - $u.\pi \leftarrow nil$
    - $u.ptr \leftarrow adj[u]$

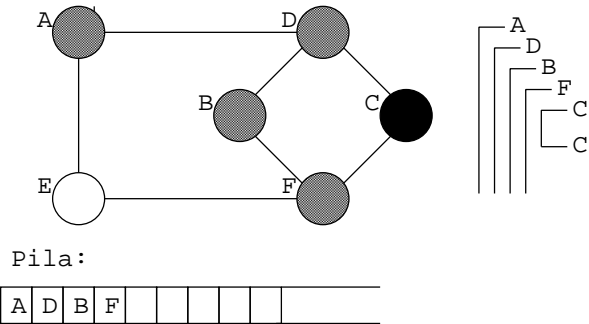
### 3. Un altro esempio di DFS



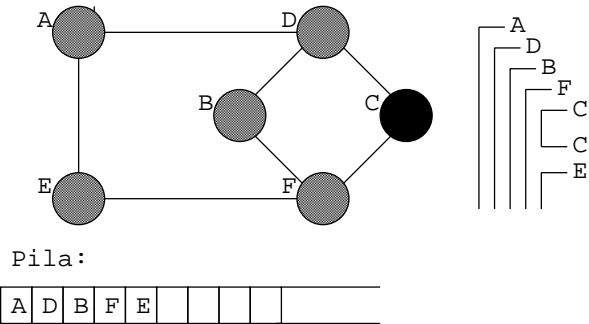
### 3. Un altro esempio di DFS



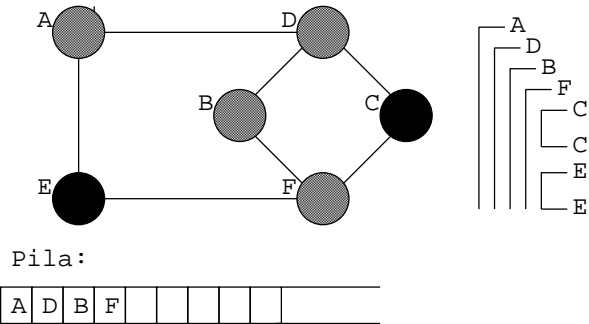
3. Un altro esempio di DFS



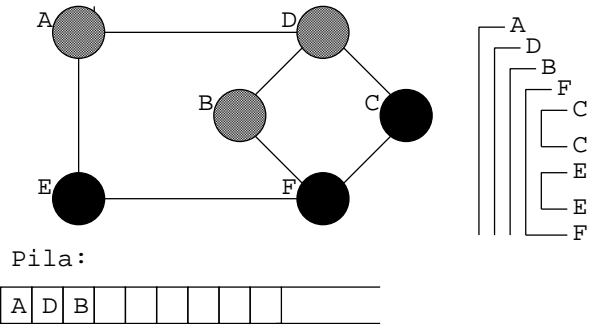
3. Un altro esempio di DFS



3. Un altro esempio di DFS

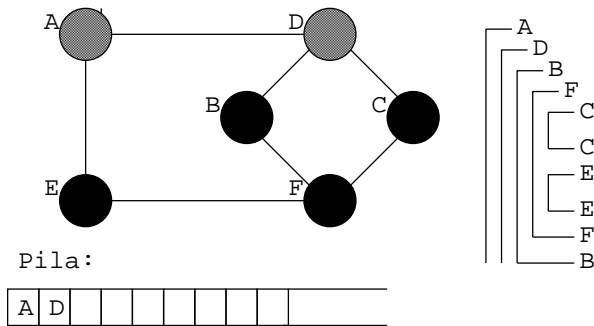


3. Un altro esempio di DFS

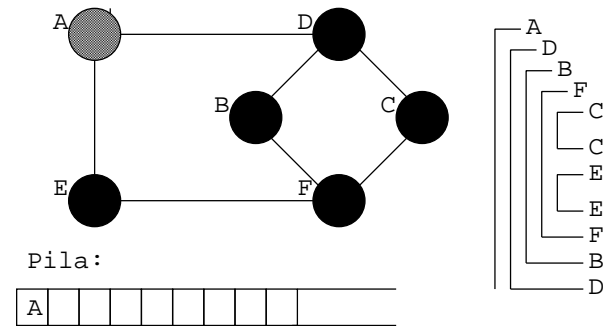




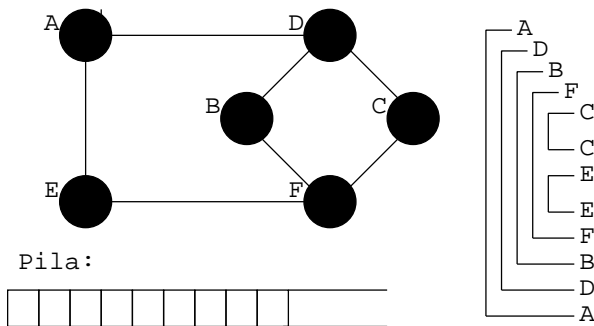
### 3. Un altro esempio di DFS



### 3. Un altro esempio di DFS

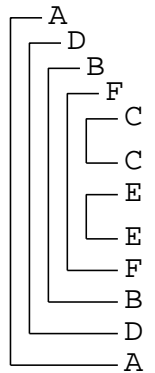


### 3. Un altro esempio di DFS



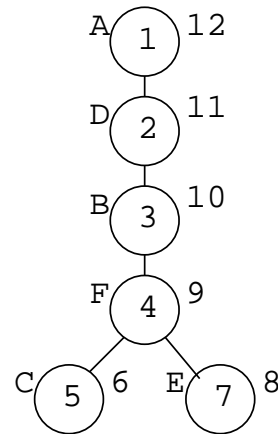
### 3. Struttura di “attivazione”

- ▶ gli intervalli di “attivazione” di una qualunque coppia di nodi sono:
  - ▶ disgiunti oppure
  - ▶ uno interamente contenuto nell'altro



### 3. Struttura di “attivazione”

- ▶ un vertice non viene “disattivato” finchè non sono stati “attivati” e “disattivati” tutti i suoi discendenti
- ▶ è l'ordine in cui si percorre l'albero delle chiamate ricorsive di una procedura ricorsiva
- ▶ progettiamo una versione ricorsiva dell'algoritmo di visita in profondità



### 3. Versione ricorsiva della visita in profondità

```

VISITA(G, s)
 s.color ← grigio
 while ∃ v : v è bianco ed è ∈ adj[s] do
 v.π ← s
 VISITA(G, v)
 s.color ← nero

```

### 3. Corrispondenza fra la versione ricorsiva e iterativa

- ▶ c'è corrispondenza fra lo stack della versione iterativa e lo stack delle attivazioni della procedura ricorsiva
- ▶ supponiamo che le due versioni visitino gli adiacenti nello stesso ordine
- ▶ se il contenuto dello stack della versione iterativa è  $\{v_1, \dots, v_r\}$  ( $v_1 = s$  e  $v_r$  è al top dello stack)
- ▶ allora la sequenza di attivazioni per la procedura ricorsiva è  $\{VISITA(G, v_1), \dots, VISITA(G, v_r)\}$
- ▶ può essere dimostrata per induzione sul numero di elementi nello stack

### 3. Versione estesa per raccogliere altre informazioni

- ▶ introduciamo un contatore “time” per ricordare l'ordine delle attivazioni e disattivazioni
- ▶ 

```

VISITA(G, s)
 s.color ← grigio
 s.d ← time
 time ← time + 1
 while ∃ v : v è bianco ed è ∈ adj[s] do
 v.π ← s
 VISITA(G, v)
 s.f ← time
 time ← time + 1
 s.color ← nero

```

### 3. Versione estesa per raccogliere altre informazioni

- ▶ bisogna inizializzare le nuove variabili:
- ▶ INIZIALIZZA( $G$ )
  - for**  $\forall u \in V$  **do**
    - $u.color \leftarrow bianco$
    - $u.\pi \leftarrow nil$
    - $u.d \leftarrow \infty$
    - $u.f \leftarrow \infty$
  - $time \leftarrow 1$
- ▶ tempi di un nodo non visitato rimangono infiniti

### 3. Visita in profondità di tutti i nodi

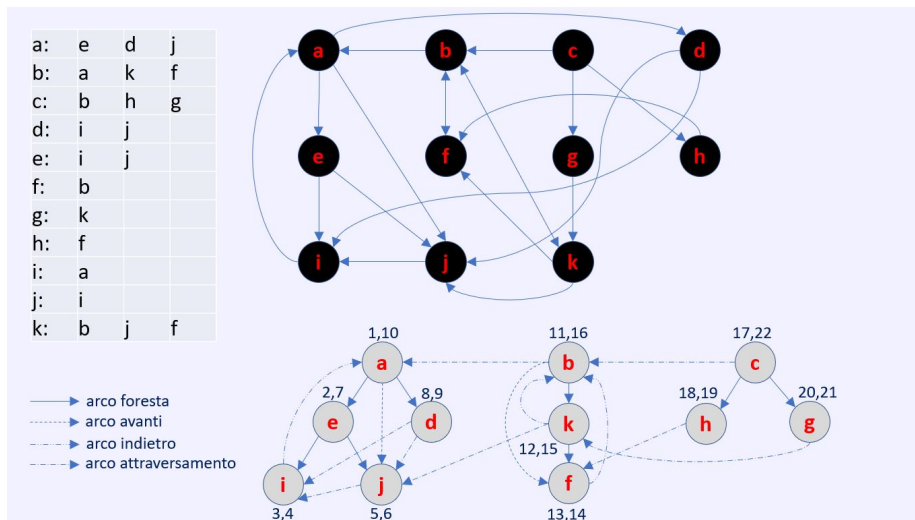
- ▶ l'algoritmo precedente visita solo il componente di cui il nodo iniziale ( $s$ ) fa parte
- ▶ visita tutto il grafo solo se il grafo è connesso
- ▶ visita intera di un grafo non connesso:
  - VISITA-TUTTI-VERTICI( $G$ )
  - INIZIALIZZA( $G$ )
  - for**  $\forall u \in V$  **do**
    - if**  $u.color = bianco$  **then**
      - VISITA( $G, u$ )

### 3. Proprietà della visita in profondità di tutti i nodi

- ▶ Teorema delle parentesi: in ogni visita DFS in un grafo (orientato o non orientato), per ogni coppia di nodi  $u, v$ , una e una sola delle seguenti condizioni è soddisfatta:
  - ▶  $u.d < v.d < v.f < u.f$  e  $u$  è un antenato di  $v$  in un albero della foresta DFS
  - ▶  $v.d < u.d < u.f < v.f$  e  $u$  è un discendente di  $v$  in un albero della foresta DFS
  - ▶  $u.d < u.f < v.d < v.f$  o  $v.d < v.f < u.d < u.f$  e non esiste relazione antenato-discendente tra  $u$  e  $v$  nella foresta DFS
- ▶ ogni caso implica caratteristiche del grafo:
  - ▶  $u.d < v.d < v.f < u.f$ : nel grafo esiste un cammino da  $u$  a  $v$
  - ▶  $v.d < u.d < u.f < v.f$ : nel grafo esiste un cammino da  $v$  a  $u$
  - ▶  $u.d < u.f < v.d < v.f$  e  $u$  e  $v$  fanno parte di due alberi distinti: nel grafo non esiste cammino da  $u$  a  $v$
  - ▶  $v.d < v.f < u.d < u.f$  e  $u$  e  $v$  fanno parte di due alberi distinti: nel grafo non esiste cammino da  $v$  a  $u$

### 3. Proprietà della visita in profondità di tutti i nodi

- ▶ classificazione degli archi del grafo durante DFS
  - ▶ arco dell'albero: arco inserito nella foresta DFS
  - ▶ arco all'indietro: arco che collega un nodo ad un suo antenato
  - ▶ arco in avanti: arco che collega un nodo ad un suo discendente
  - ▶ arco di attraversamento: arco che collega due vertici che non sono in relazione antenato-discendente



### 3. Proprietà della visita in profondità di tutti i nodi

- ▶ un arco  $(u, v)$  viene classificato quando si esamina  $v$  nella lista di adiacenti  $\text{adj}[u]$
- ▶ in quel momento  $v.\text{color}$  può essere:
  - ▶ *bianco*:  $(u, v)$  è un arco della foresta DFS
  - ▶ *grigio*:  $u$  è un discendente di  $v$  in un albero della foresta DFS,  $(u, v)$  è un arco all'indietro
  - ▶ *nero*: la visita di  $v$  è già terminata (e quindi  $v.f < u.f$ ),  $(u, v)$  è un arco
    - ▶ in avanti se  $v$  è un discendente di  $u$ , in tal caso  $u.d < v.d \Rightarrow u.d < v.d < v.f < u.f$
    - ▶ di attraversamento altrimenti, in tal caso  $v.d < u.d \Rightarrow v.d < v.f < u.d < u.f$
- ▶ i precedenti casi forniscono un criterio per la classificazione

### 3. Proprietà della visita in profondità di tutti i nodi

- ▶ teorema: in una visita DFS di un grafo non orientato, ogni arco è un arco dell'albero o un arco all'indietro
- ▶ teorema: un grafo, orientato o non orientato, è aciclico se e solo se una visita DFS (qualunque) non produce archi all'indietro