Algoritmi di Visita di Grafi

Damiano Macedonio
mace@unive.it

Attraversamento grafi

- Definizione del problema
  - Dato un grafo *G=(V, E)* ed un nodo *s* di V (detto *sorgente*), visitare ogni nodo nel grafo raggiungibile da *s*
  - Ogni nodo deve essere visitato una sola volta
- Visita in ampiezza (breadth-first search)
  - Visita i nodi "espandendo" la frontiera fra nodi scoperti / da scoprire
  - Es: *Cammini più brevi da singola sorgente*
- Visita in profondità (depth-first search)
  - Visita i nodi andando il "più lontano possibile" nel grafo
  - Es: *Componenti fortemente connesse, ordinamento topologico*

Visita: attenzione alle soluzioni “facili”

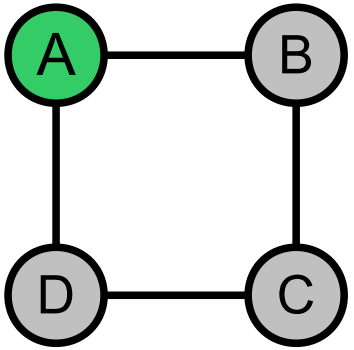
- Prendere ispirazione dalla visita degli alberi
  - utilizziamo una visita BFS basata su coda
  - trattiamo i “vertici adiacenti” come se fossero i “figli”

```
algoritmo non-visita(grafo G, nodo s)
  coda := {s}
  while (coda ≠ ∅) do
    u = coda.dequeue()
    “visita u”
    for each “nodo v adiacente a u” do
      coda.enqueue(v)
    endfor
  endwhile
```

Sbagliato

Perché non funziona?

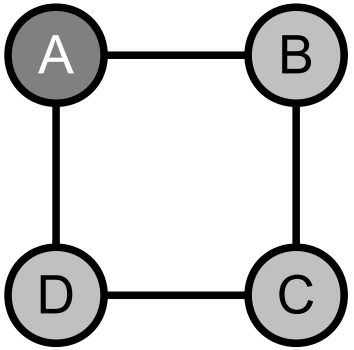
```
algoritmo non-visita(grafo G, nodo s)
  coda := {s}
  while (coda ≠ ∅) do
    u = coda.dequeue()
    "visita u"
    for each "nodo v adiacente a u" do
      coda.enqueue(v)
    endfor
  endwhile
```



coda = { A }

Perché non funziona?

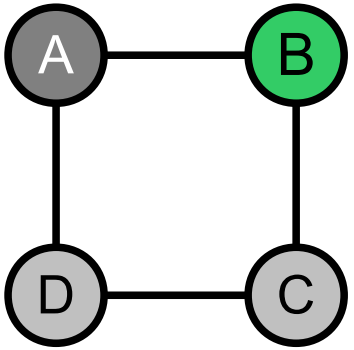
```
algoritmo non-visita(grafo G, nodo s)
  coda := {s}
  while (coda ≠ ∅) do
    u = coda.dequeue()
    "visita u"
    for each "nodo v adiacente a u" do
      coda.enqueue(v)
    endfor
  endwhile
```



coda = { A }
coda = { B, D }

Perché non funziona?

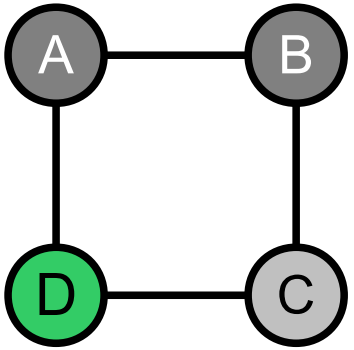
```
algoritmo non-visita(grafo G, nodo s)
  coda := {s}
  while (coda ≠ ∅) do
    u = coda.dequeue()
    "visita u"
    for each "nodo v adiacente a u" do
      coda.enqueue(v)
    endfor
  endwhile
```



coda = { A }
coda = { B, D }
coda = { D, A, C }

Perché non funziona?

```
algoritmo non-visita(grafo G, nodo s)
  coda := {s}
  while (coda ≠ ∅) do
    u = coda.dequeue()
    "visita u"
    for each "nodo v adiacente a u" do
      coda.enqueue(v)
    endfor
  endwhile
```



coda = { A }
coda = { B, D }
coda = { D, A, C }
coda = { A, C, A, C }

Perché non funziona?

```
algoritmo non-visita(grafo G, nodo s)
  coda := {s}
  while (coda ≠ ∅) do
    u = coda.dequeue()
    "visita u"
    for each "nodo v adiacente a u" do
      coda.enqueue(v)
    endfor
  endwhile
```

- Problema: questo algoritmo non termina se applicato a grafi con cicli

Schema di algoritmo per la visita

```
algoritmo visita(G, s)→albero
  rendi "non marcati" tutti i nodi
  T := s
  F := { s }
  "marca" il nodo s
  while (F ≠ ∅)  do
    u := F.extract()
    "visita il nodo u"
    for each v adiacente a u do
      if (v non è marcato) then
        marca il nodo v
        F.insert(v)
        v.parent := u
      endif
    endfor
  endwhile
  return T
```

- *F* è l'insieme *frontiera* (o *frangia*)
- Il funzionamento di *extract()* e *insert()* non è specificato
- *T* è l'albero che viene costruito dalla visita
- *v.parent* è il padre di *v* nell'albero *T*

Algoritmo generico per la visita

- Alcune cose da notare:
  - I nodi vengono visitati al più una volta (marcatura)
  - Tutti i nodi raggiungibili da *s* vengono visitati
    - Ne segue che *T* è un albero che contiene esattamente tutti i nodi raggiungibili da *s*
  - Ciascun arco viene “percorso” al piú due volte nel caso dei grafi non orientati ({u,v}, {v,u}).
  - La visita avviene in base all'ordine di estrazione
- Costo computazionale
  - $O(n+m)$ usando liste di adiacenza
  - $O(n^2)$ usando matrice di adiacenza
  - n è il numero di nodi, m è il numero di archi

Visita in ampiezza (breadth first search, BFS)

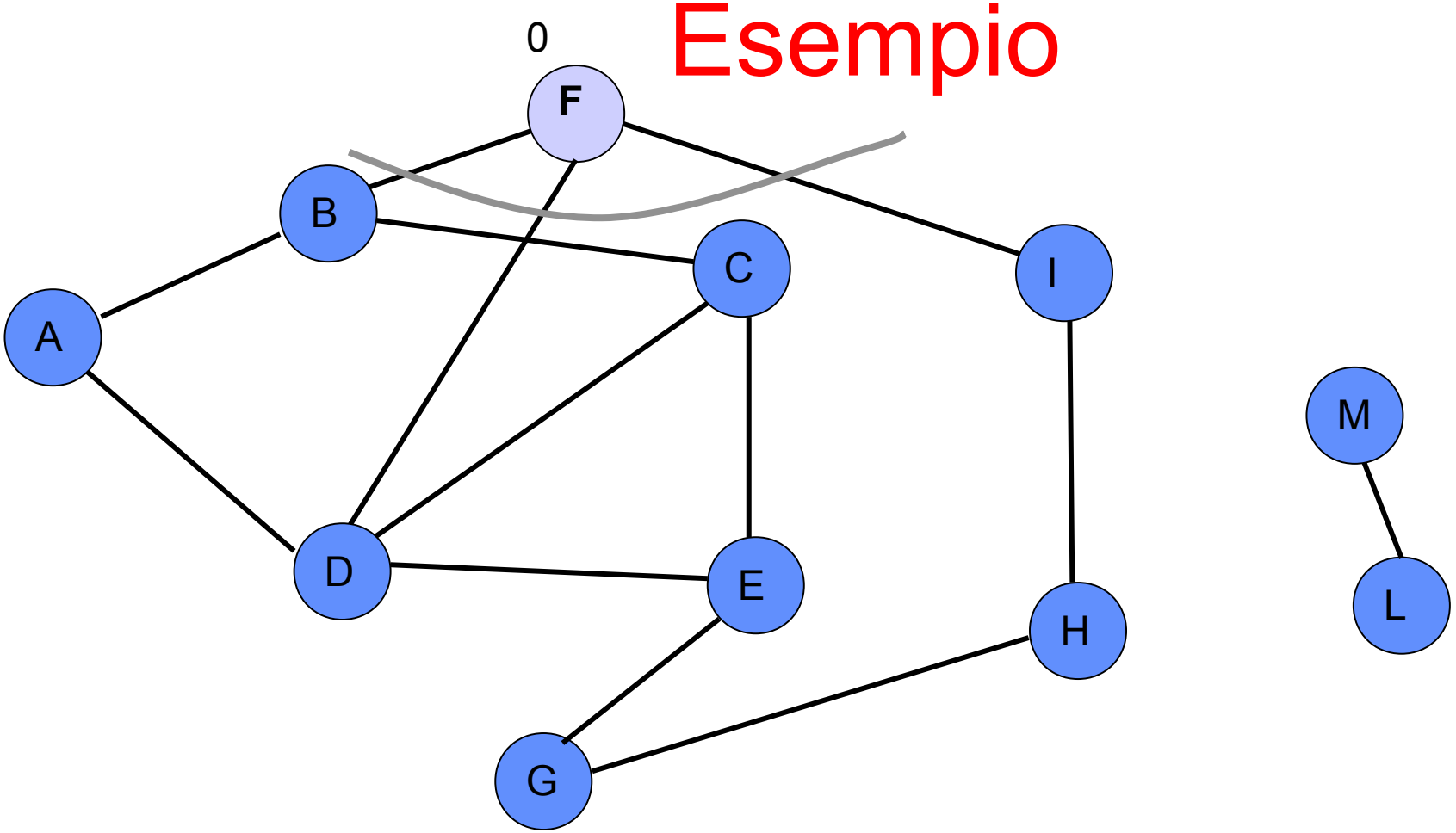
- Visita i nodi a distanze crescenti dalla sorgente
  - visita i nodi a distanza *k* prima di quelli a distanza *k*+1
- Genera un albero BF (breadth-first)
  - albero contenente tutti i vertici raggiungibili da s, e tale che il cammino da s ad un nodo nell'albero corrisponda al cammino più breve nel grafo
- Calcola la distanza minima da s a tutti i nodi da esso raggiungibili
  - distanza = numero di archi attraversati per andare da s ad un nodo da esso raggiungibile

Visita in ampiezza (breadth first search, BFS)

```
algoritmo BFS(Grafo G, vertice s)→albero
  for each v in V do v.mark := false
  T := s
  F := new Queue()
  F.enqueue(s)
  s.mark := true
  s.dist := 0
  while (F ≠ ∅) do
    u := F.dequeue()
    "visita il vertice u"
    for each v adiacente a u do
      if (not v.mark) then
        v.mark := true
        v.dist := u.dist+1
        F.enqueue(v)
        v.parent := u
      endif
    endfor
  endwhile
  return T
```

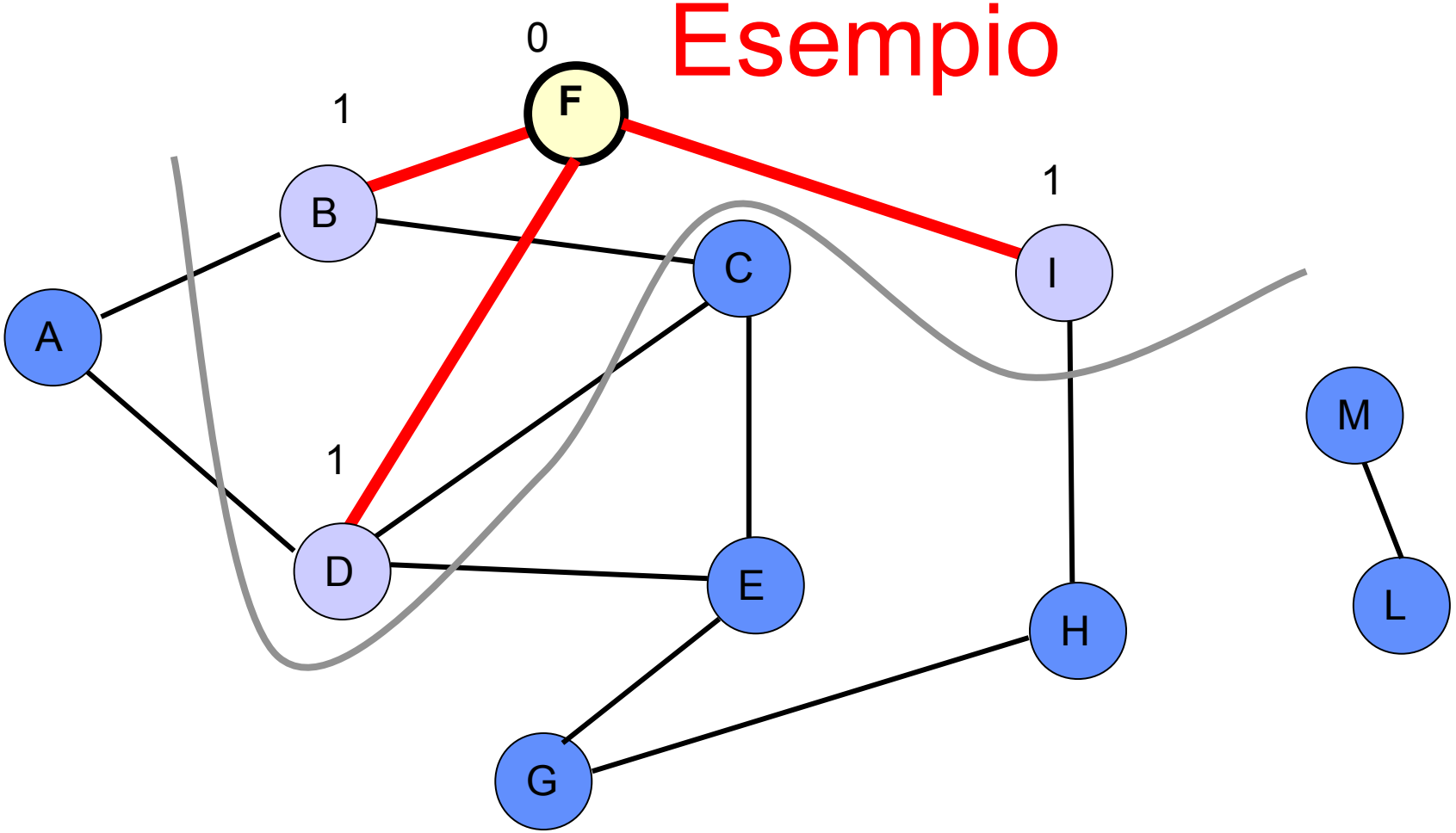
- Insieme *F* gestito tramite una coda
- *v.mark* è la marcatura del nodo *v*
- *v.dist* è la distanza del nodo *v* dal vertice *s*

Esempio



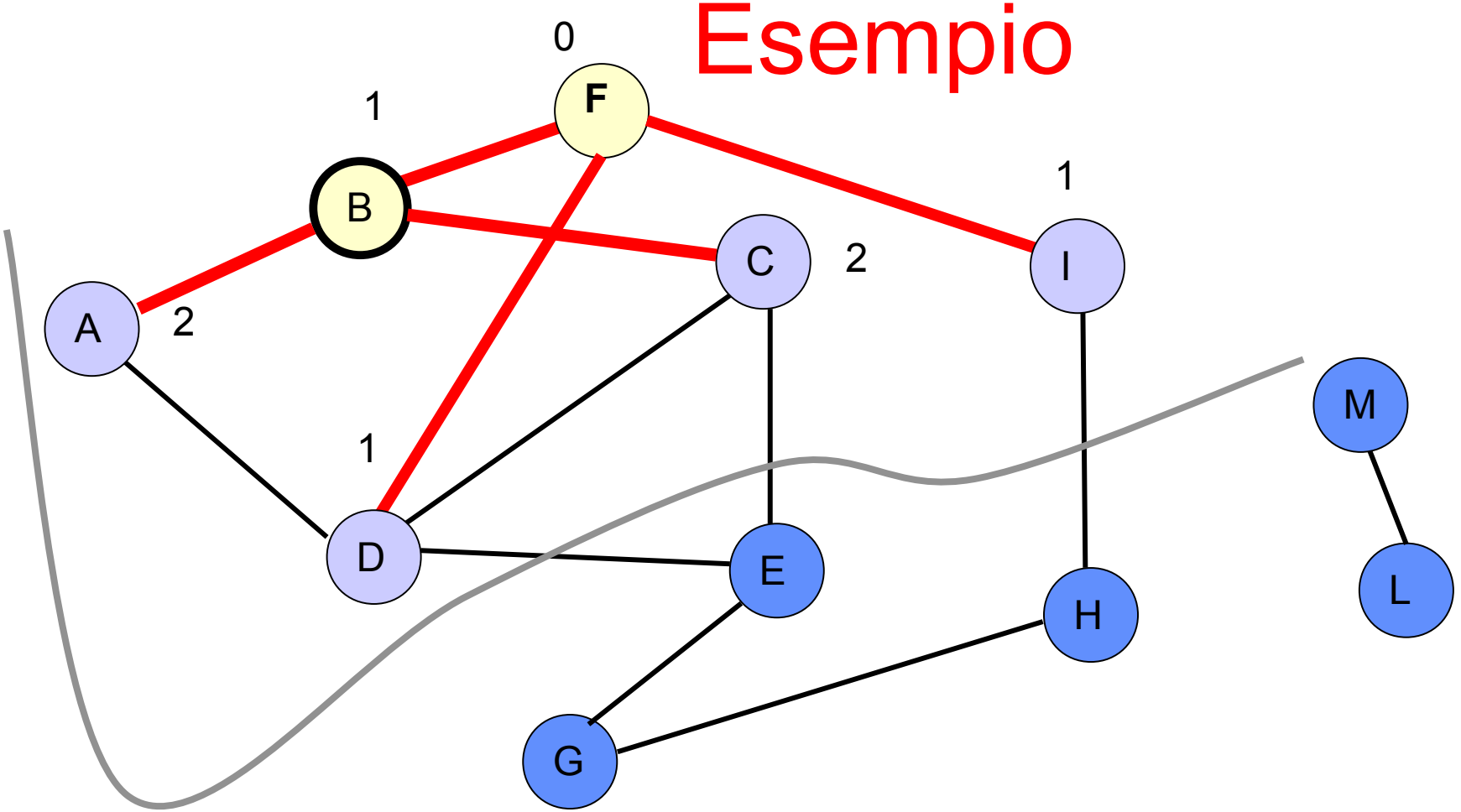
Coda:{F}

Esempio



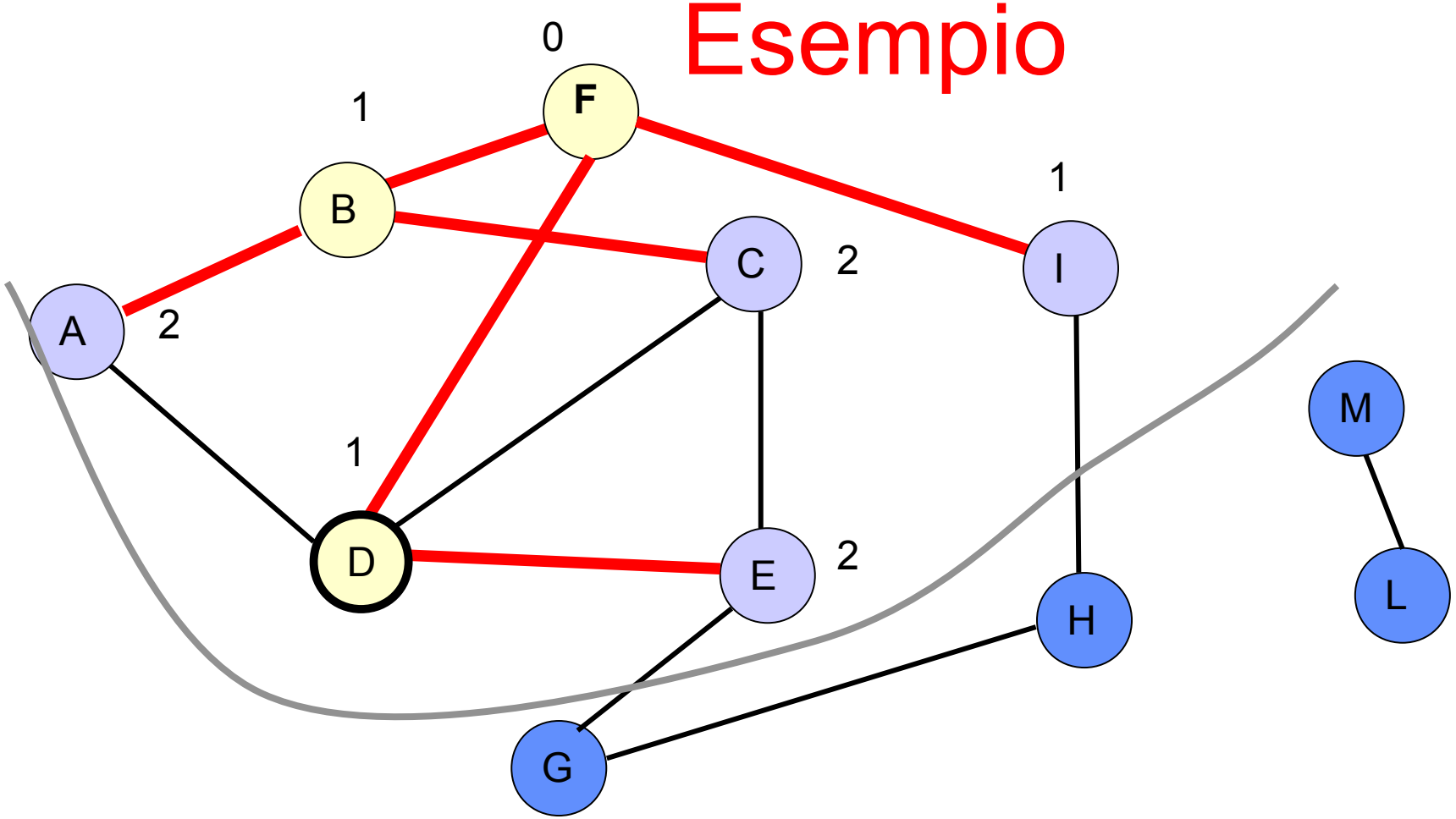
Coda:{B,D,I}

Esempio



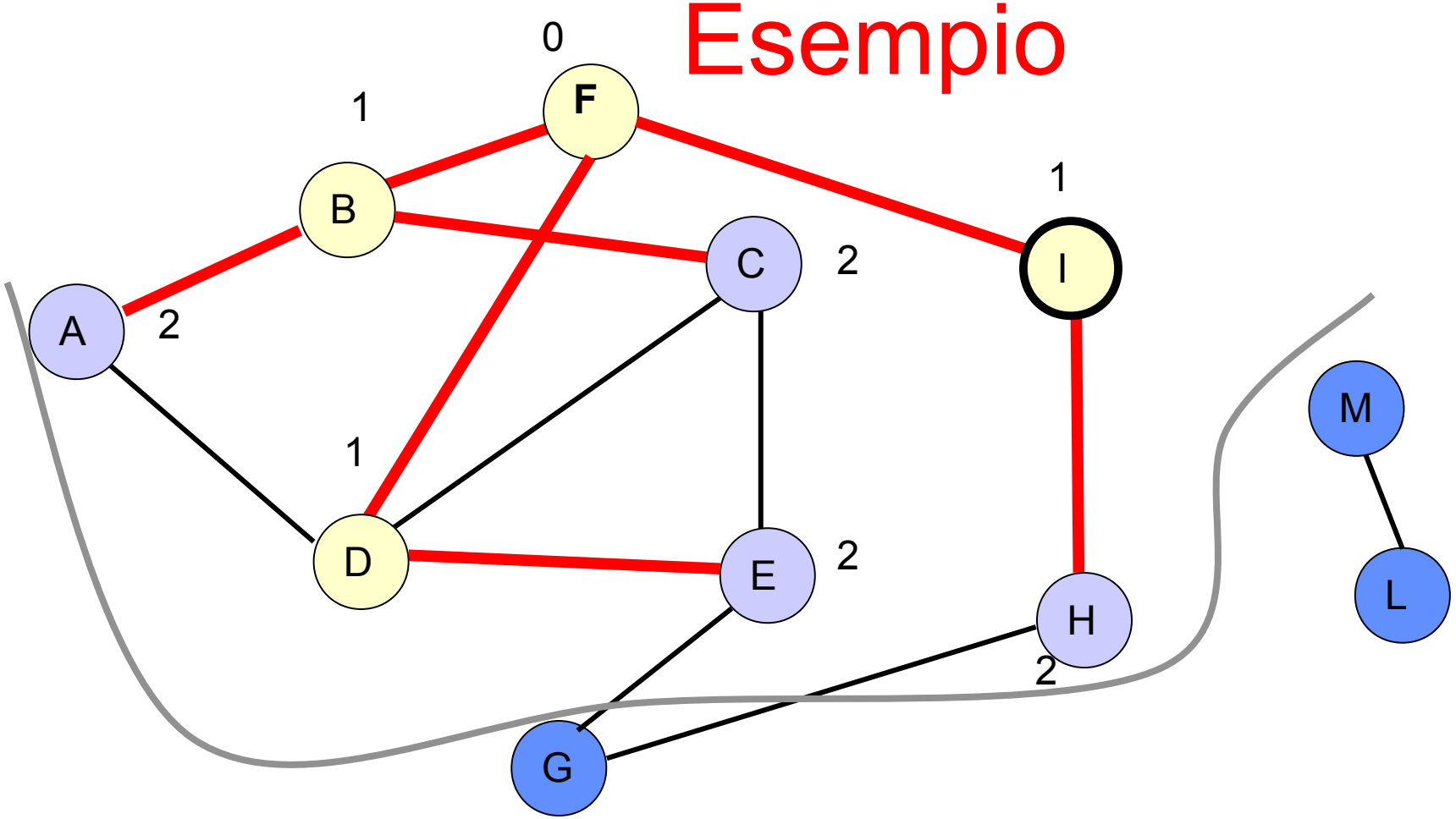
Coda:{D,I,C,A}

Esempio



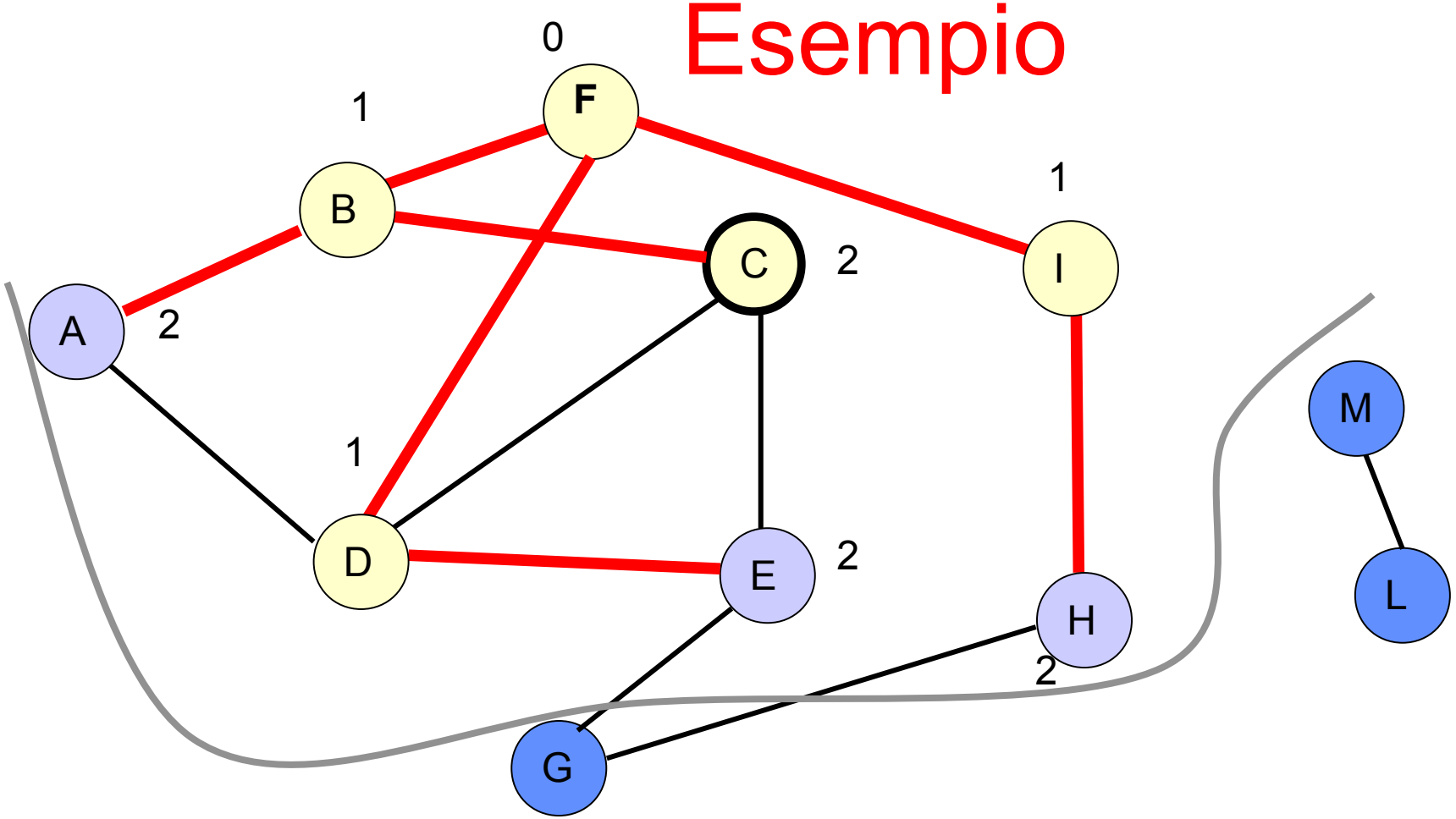
Coda:{I,C,A,E}

Esempio



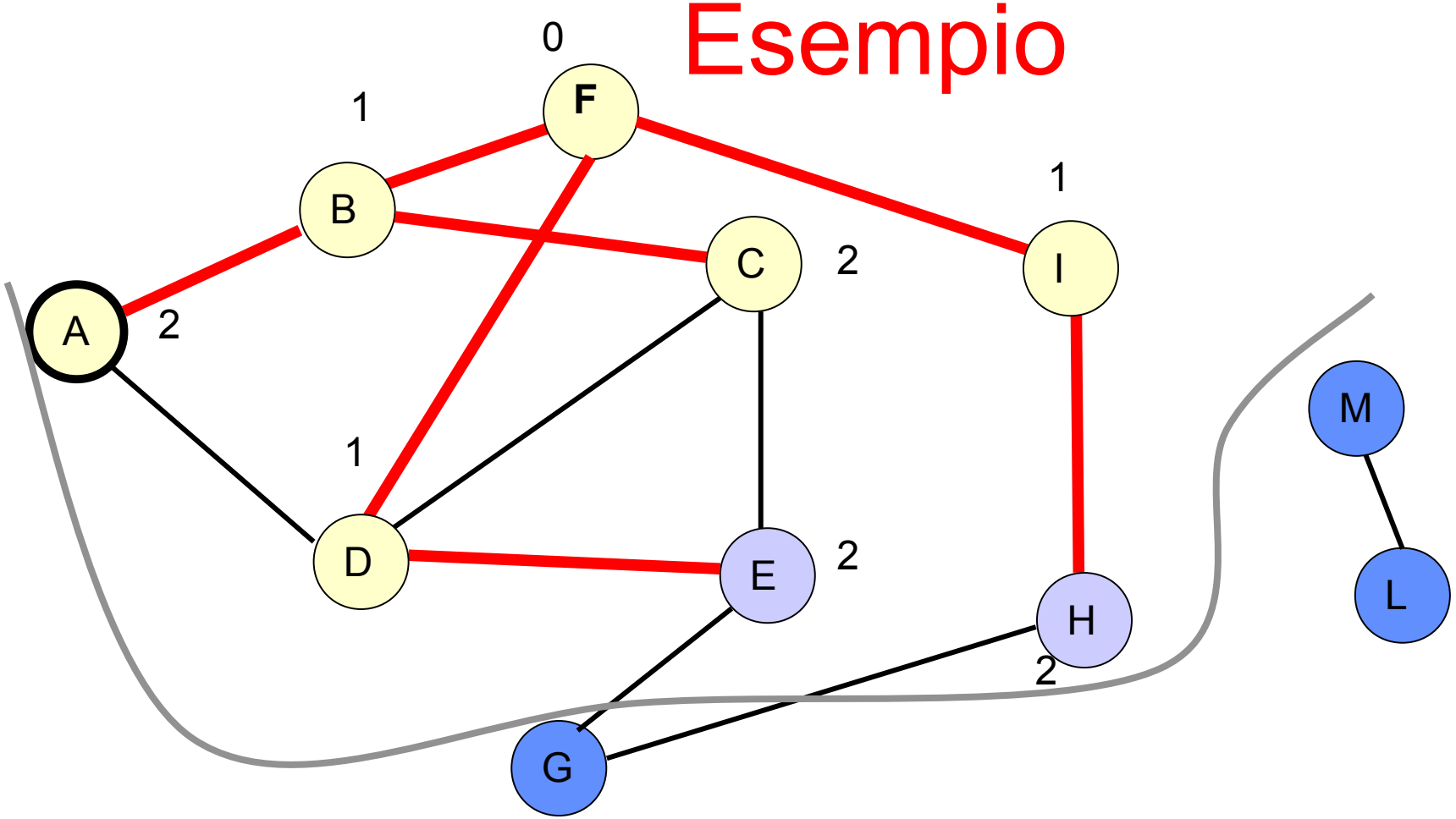
Coda:{C,A,E,H}

Esempio



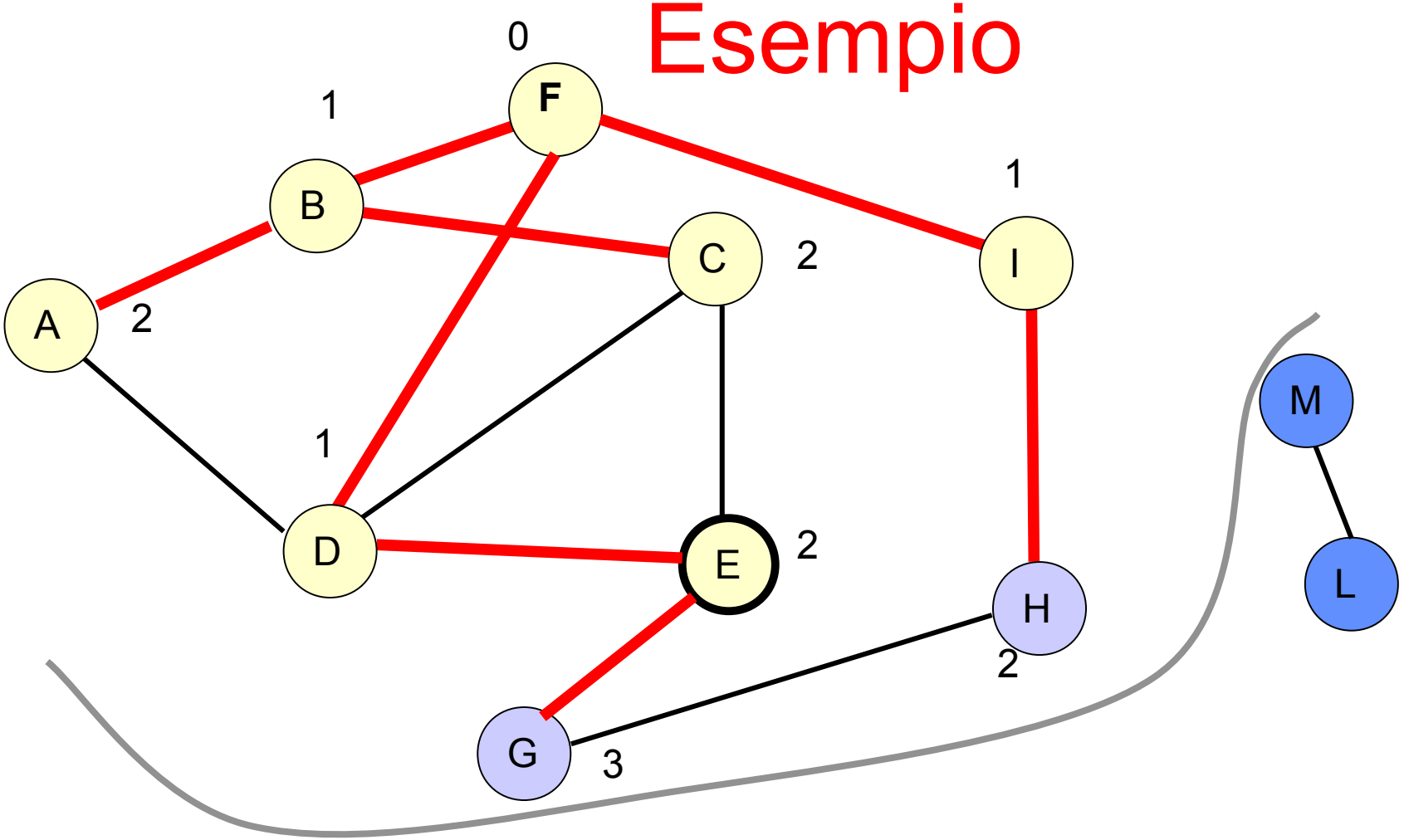
Coda:{A,E,H}

Esempio



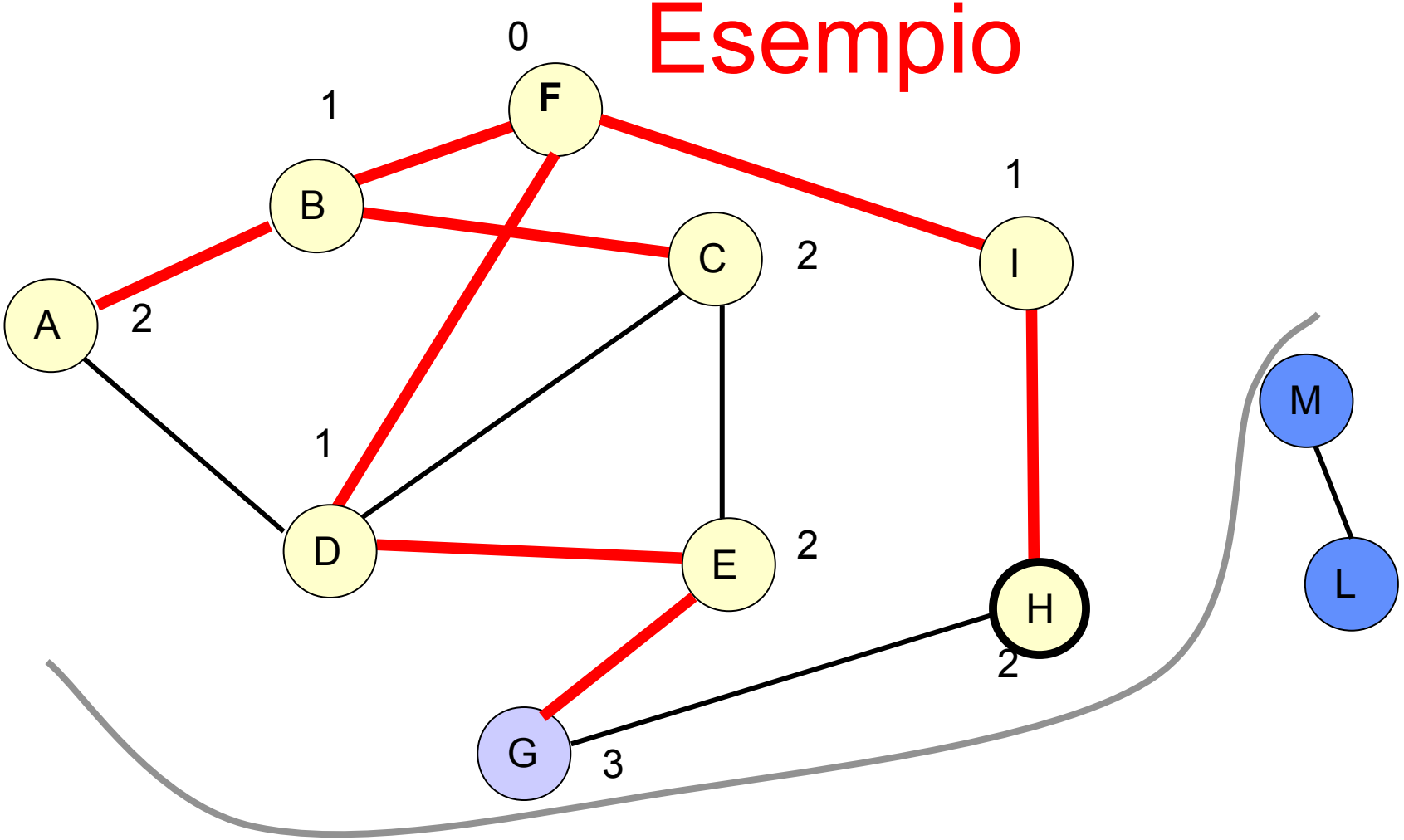
Coda:{E,H}

Esempio



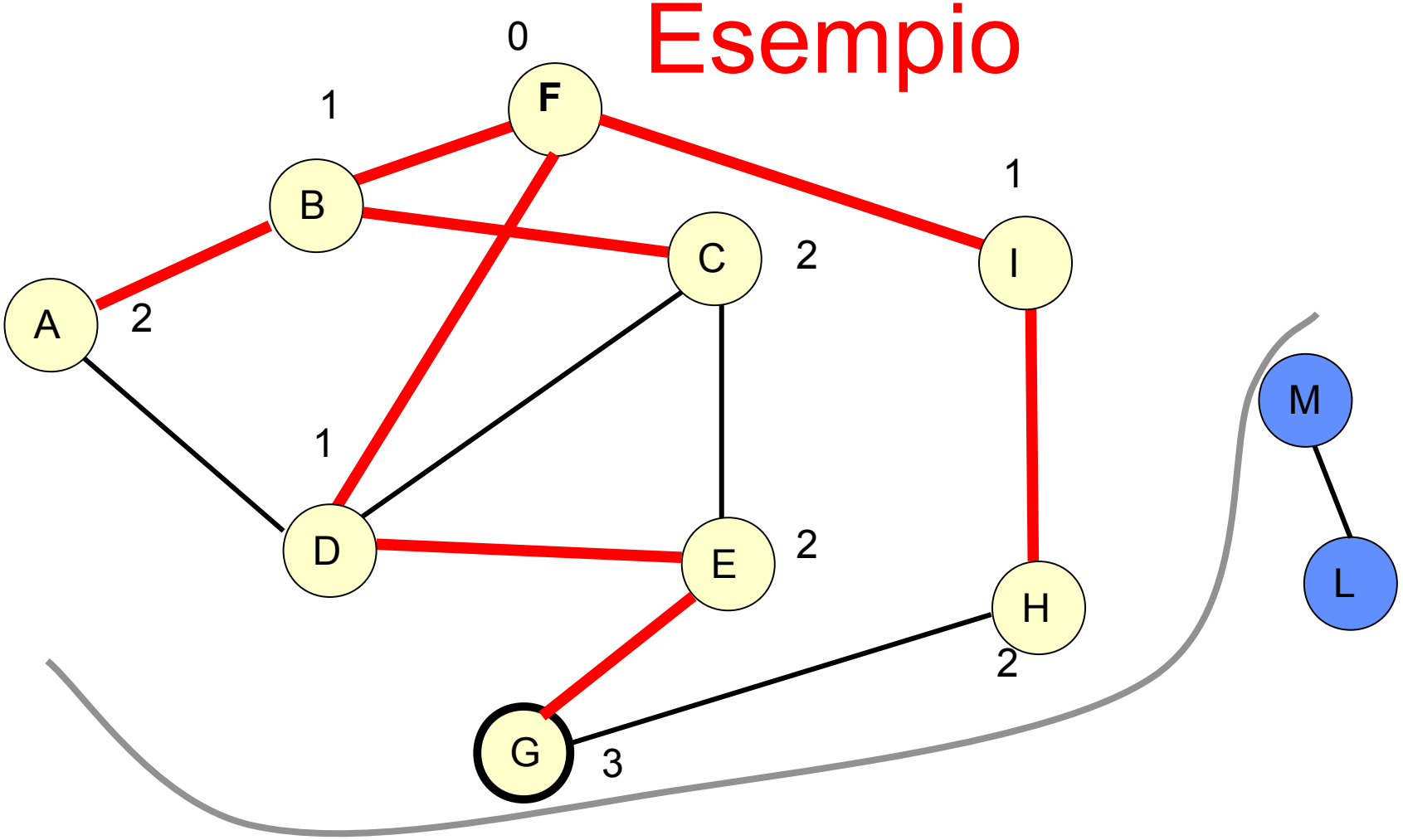
Coda:{H,G}

Esempio



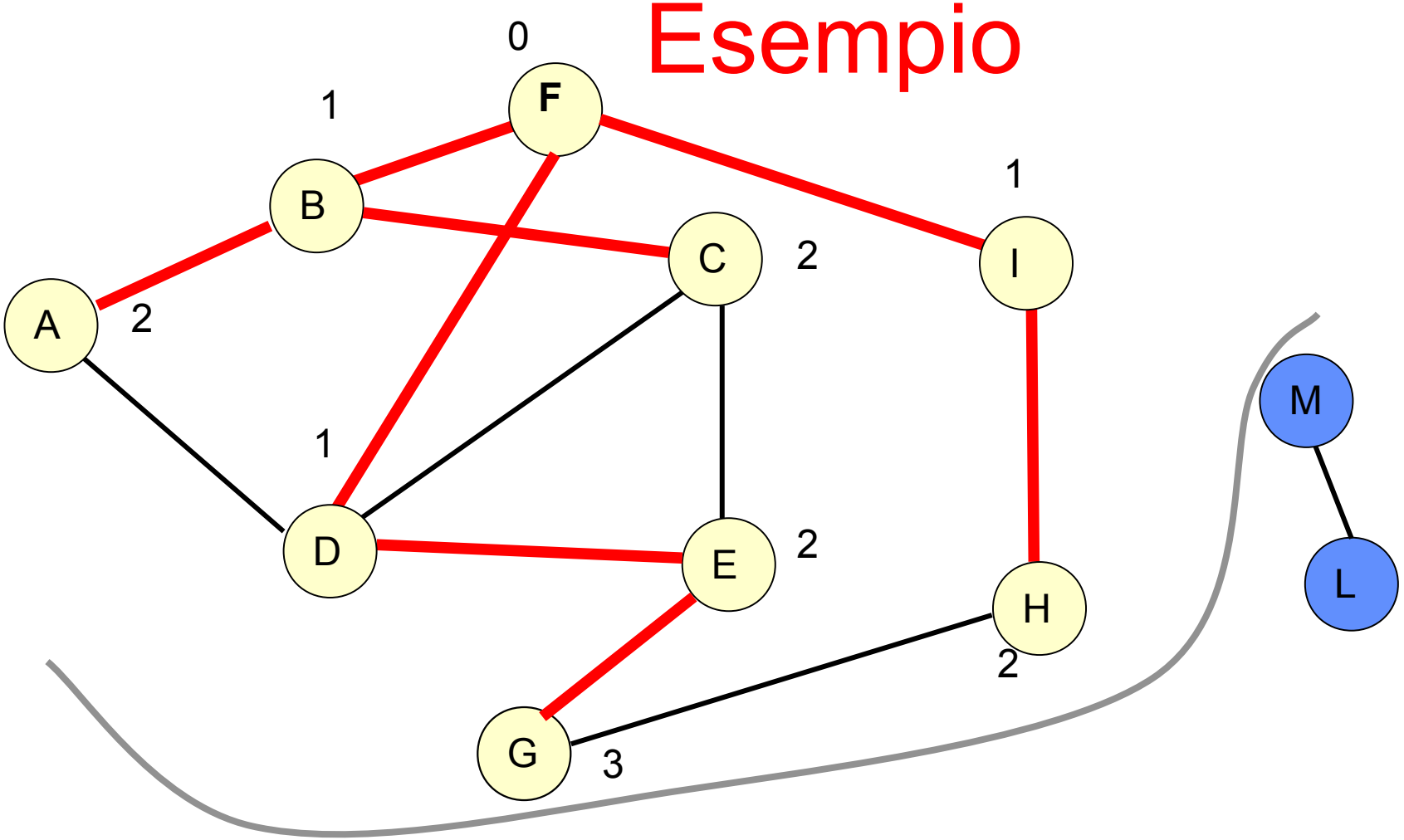
Coda: {G}

Esempio



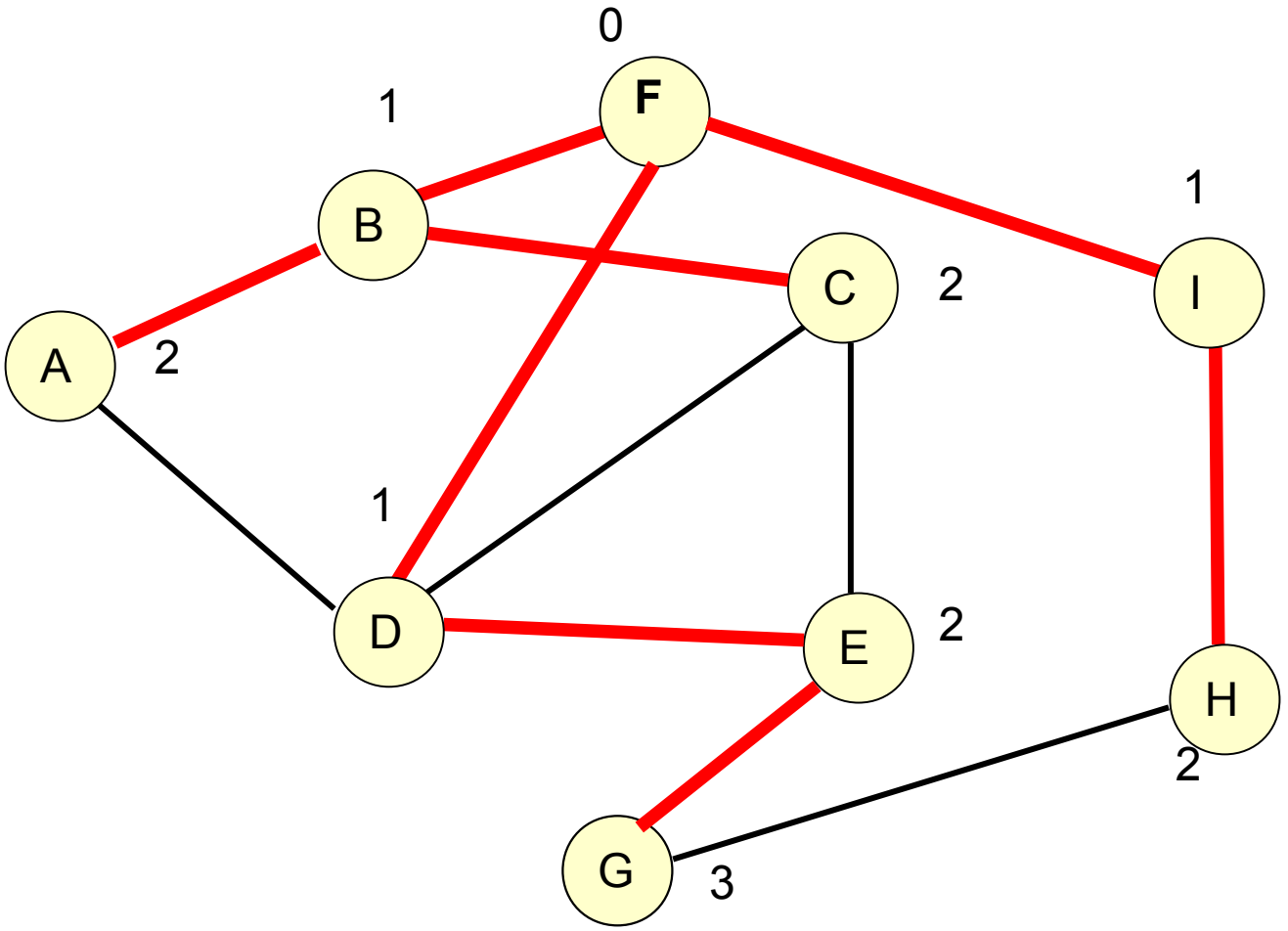
Coda: { }

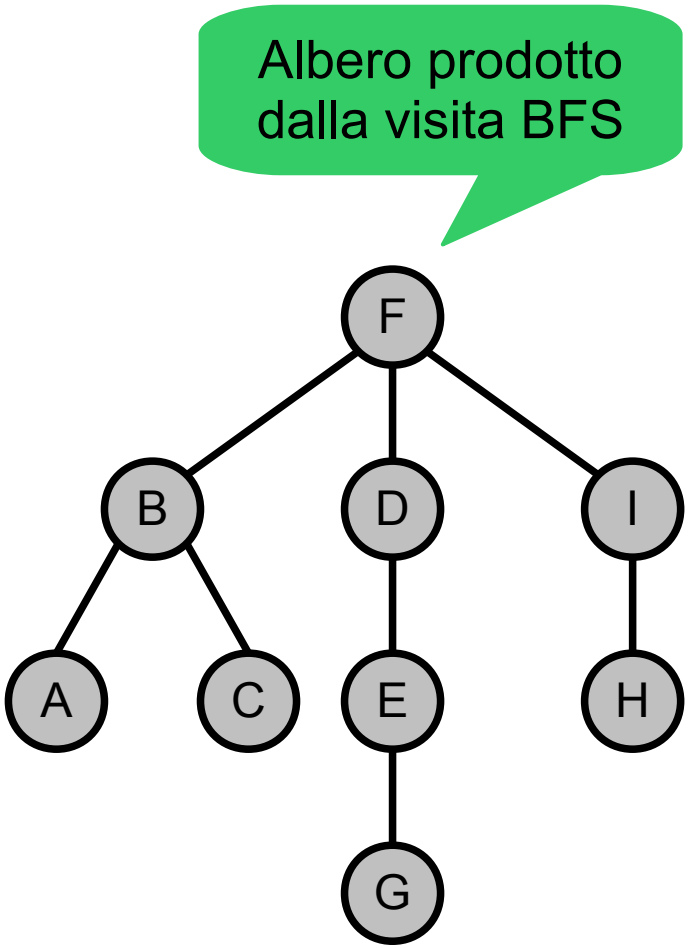
Esempio



Coda: { }

Esempio


0
F
1
B
1
I
C
2
A
2
1
D
E
2
H
2
G
3

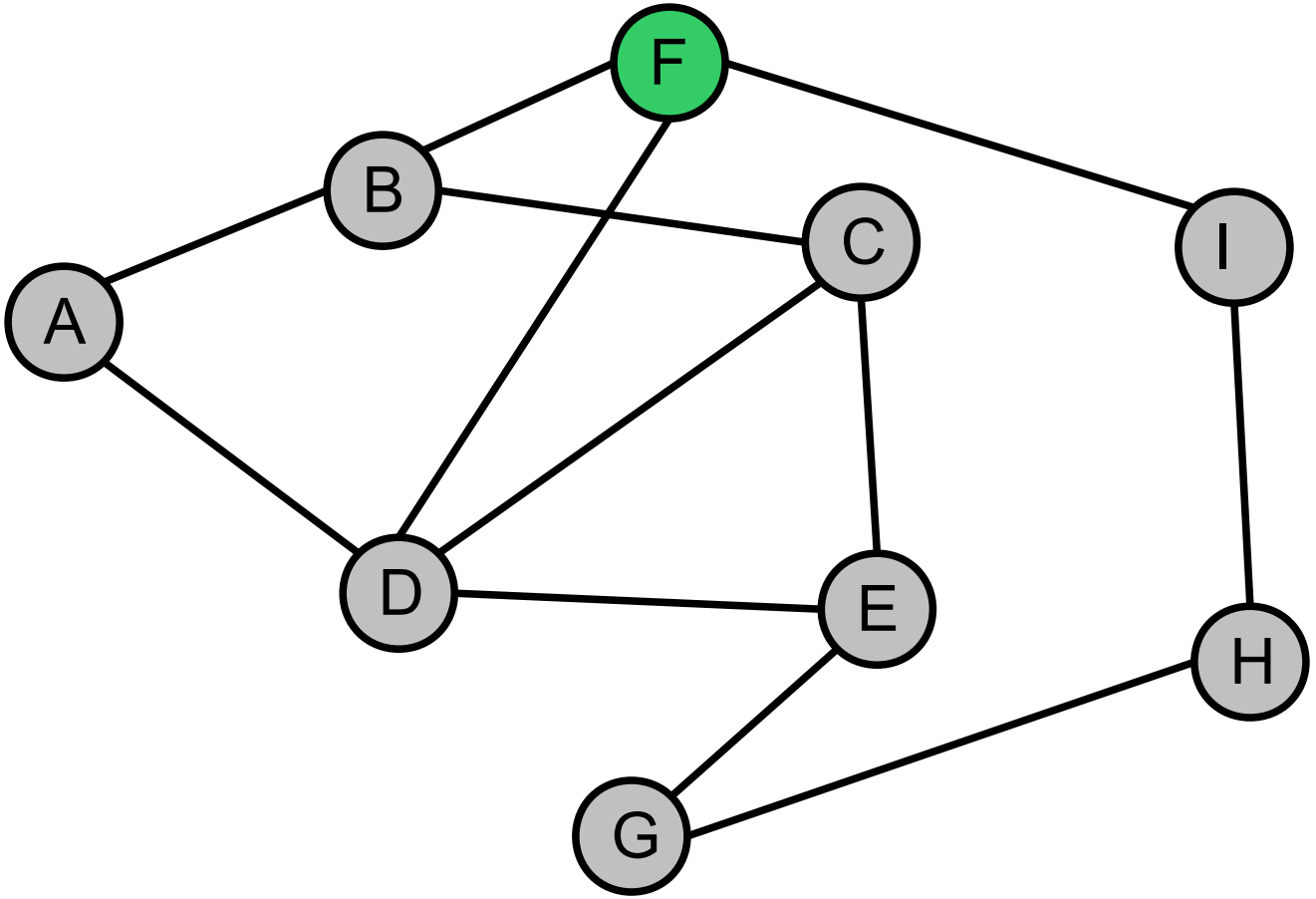

Albero prodotto dalla visita BFS
F
B
D
I
A
C
E
H
G

Applicazioni

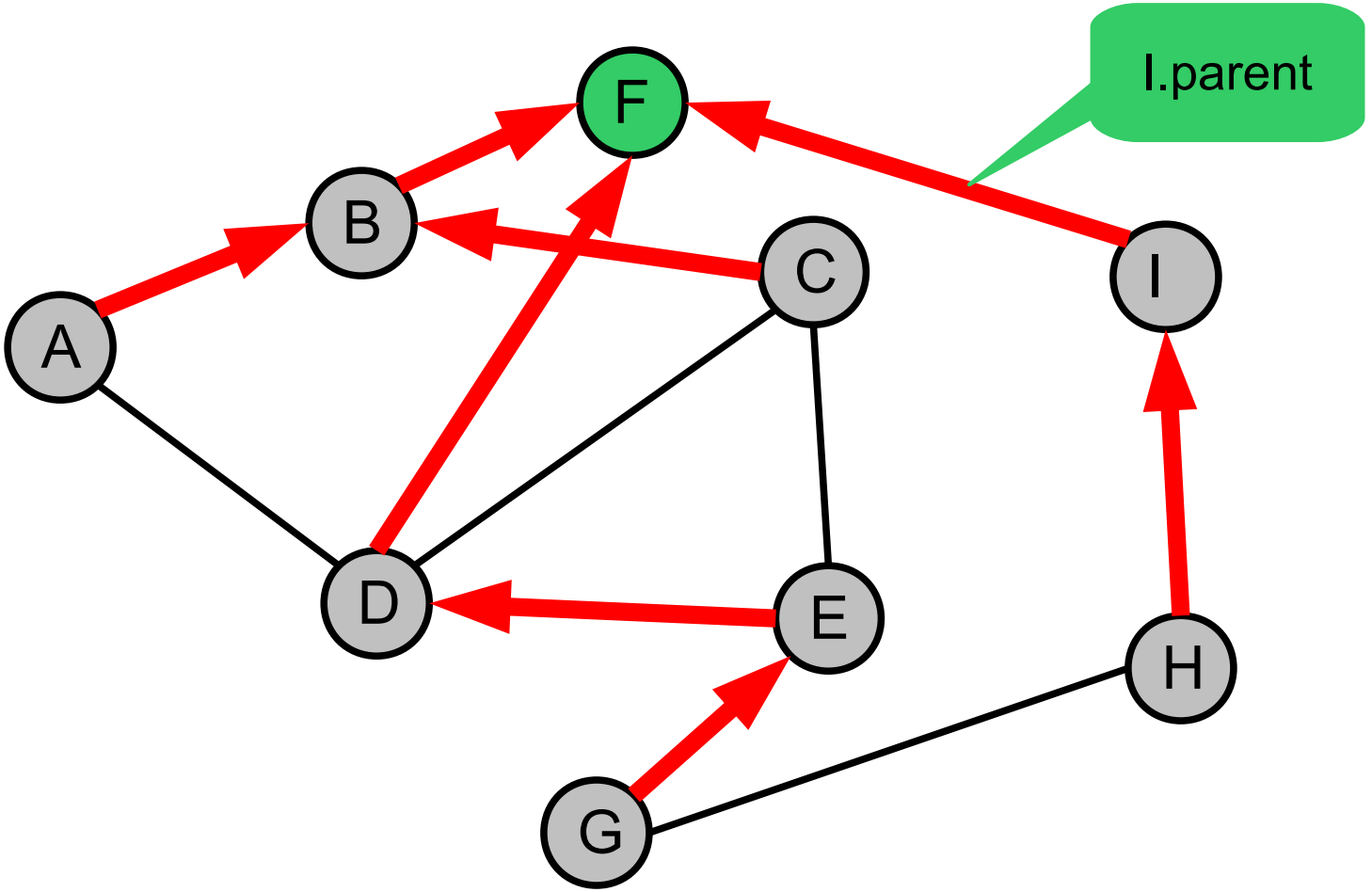
- La visita BFS può essere utilizzata per ottenere il cammino più breve (minor numero di archi attraversati) fra due vertici *s* e *v*
- Esempio: se il grafo *G* è stato precedentemente visitato con l'algoritmo BFS a partire da *s* e l'albero della visita *T* è stato creato, si può applicare l'algoritmo seguente:

```
algoritmo print-path(G, s, v)
  if (v = s) then
    print s
  else if (v.parent = nil) then
    print "no path from s to v"
  else
    print-path(G, s, v.parent)
    print v
  endif
```

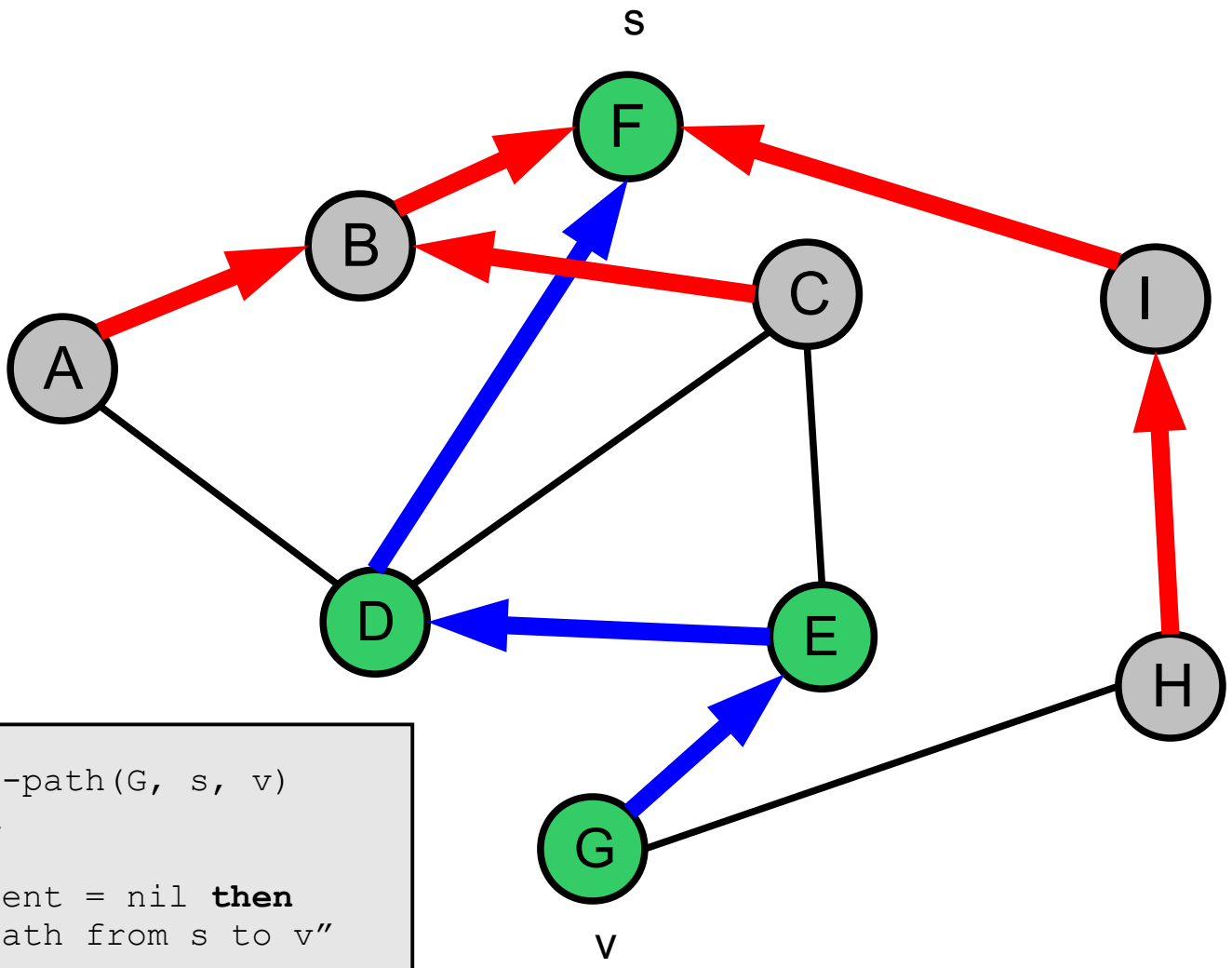
Esempio
cammino piú breve da F a G



Esempio
cammino piú breve da F a G



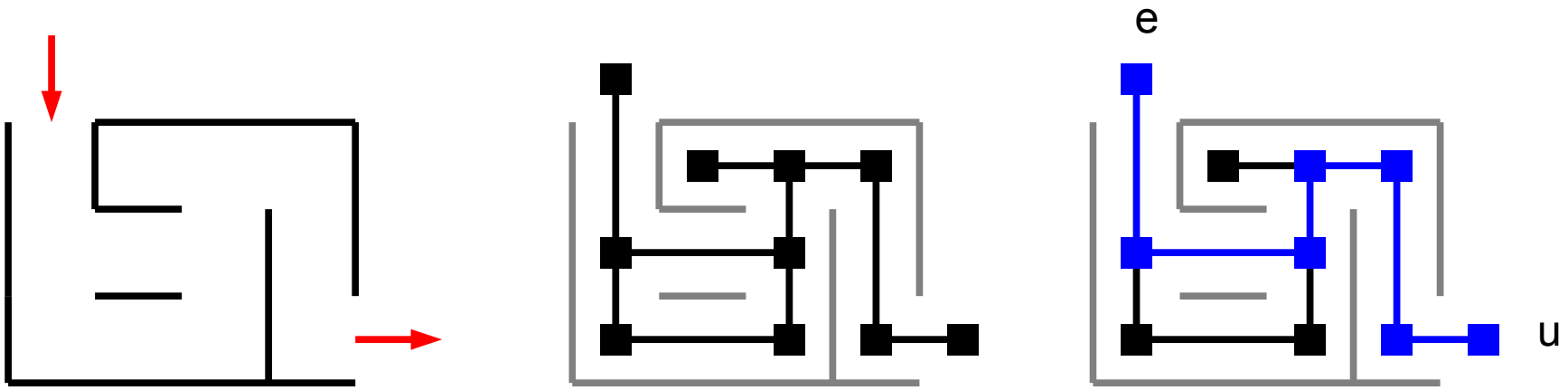
Esempio
print-path(G, s, v)




```
algoritmo print-path(G, s, v)
  if v = s then
    print s
  else if v.parent = nil then
    print "no path from s to v"
  else
    print-path(G,s,v.parent)
    print v
  endif
```

Esempio

Percorso piú breve per uscire dal labirinto?



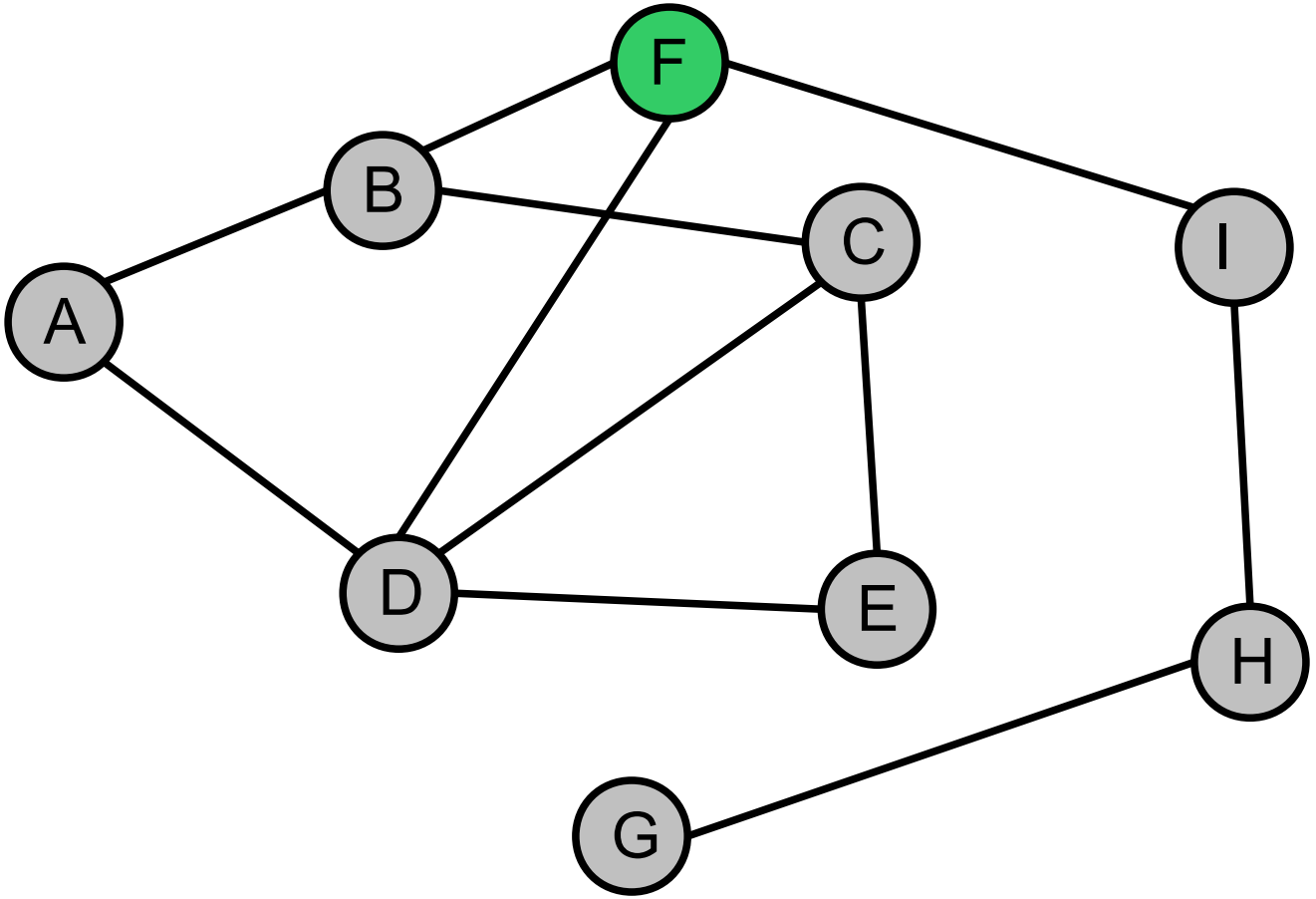
Visita in profondità (depth first search, DFS)

- Segue ciascun percorso “il più a lungo possibile”

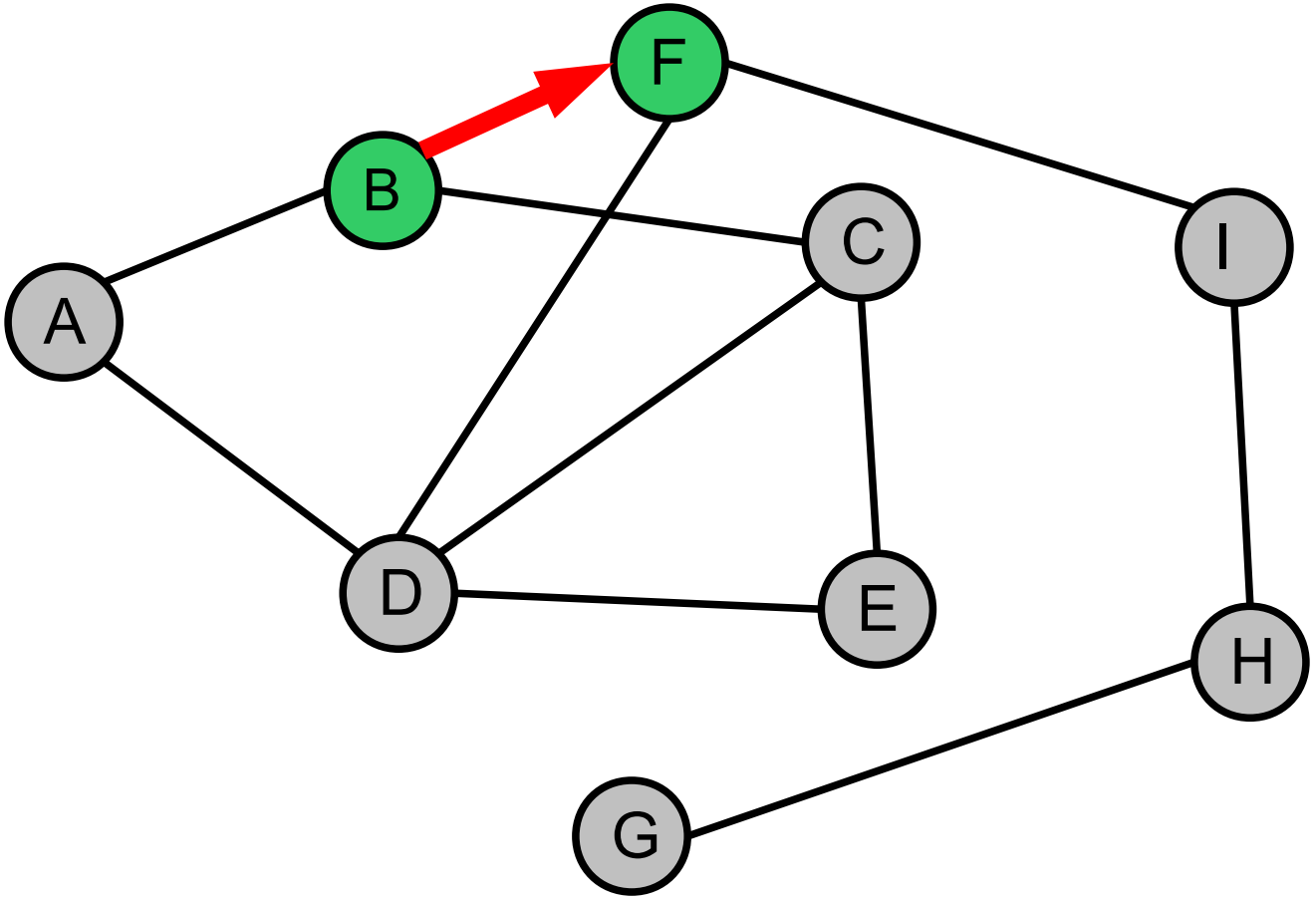
```
algoritmo DFS(Grafo G, nodo s)
  for each v in V do
    v.mark := false;
    v.parent := NULL;
  endfor
  DFS-visit(G, s)

algoritmo DFS-visit(Grafo G, nodo u)
  u.mark := true;
  “visita il vertice u”
  for each v adiacente a u do
    if (v.mark == false) then
      v.parent := u;
      DFS-visit(v);
    endif
  endfor
```

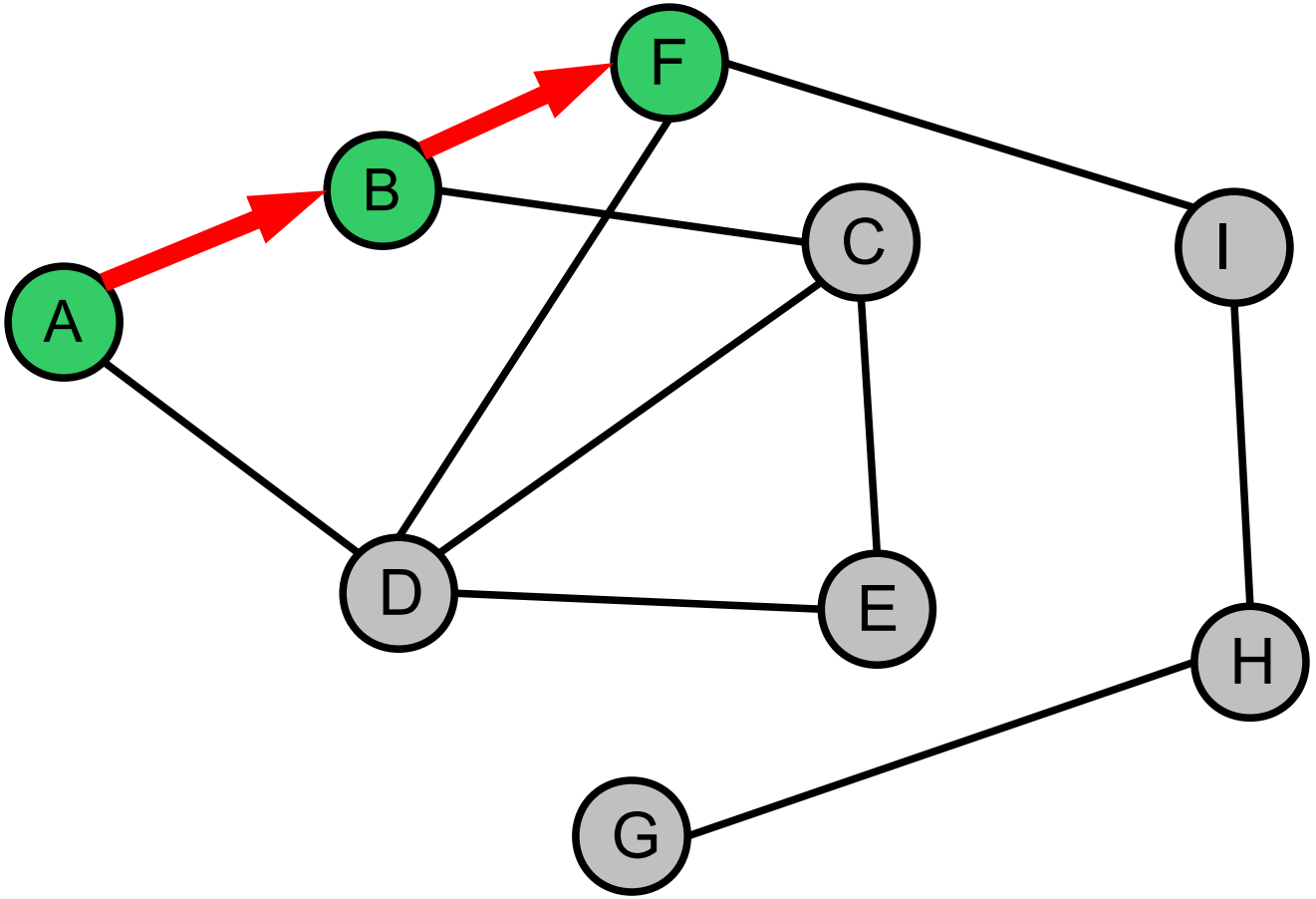
Esempio



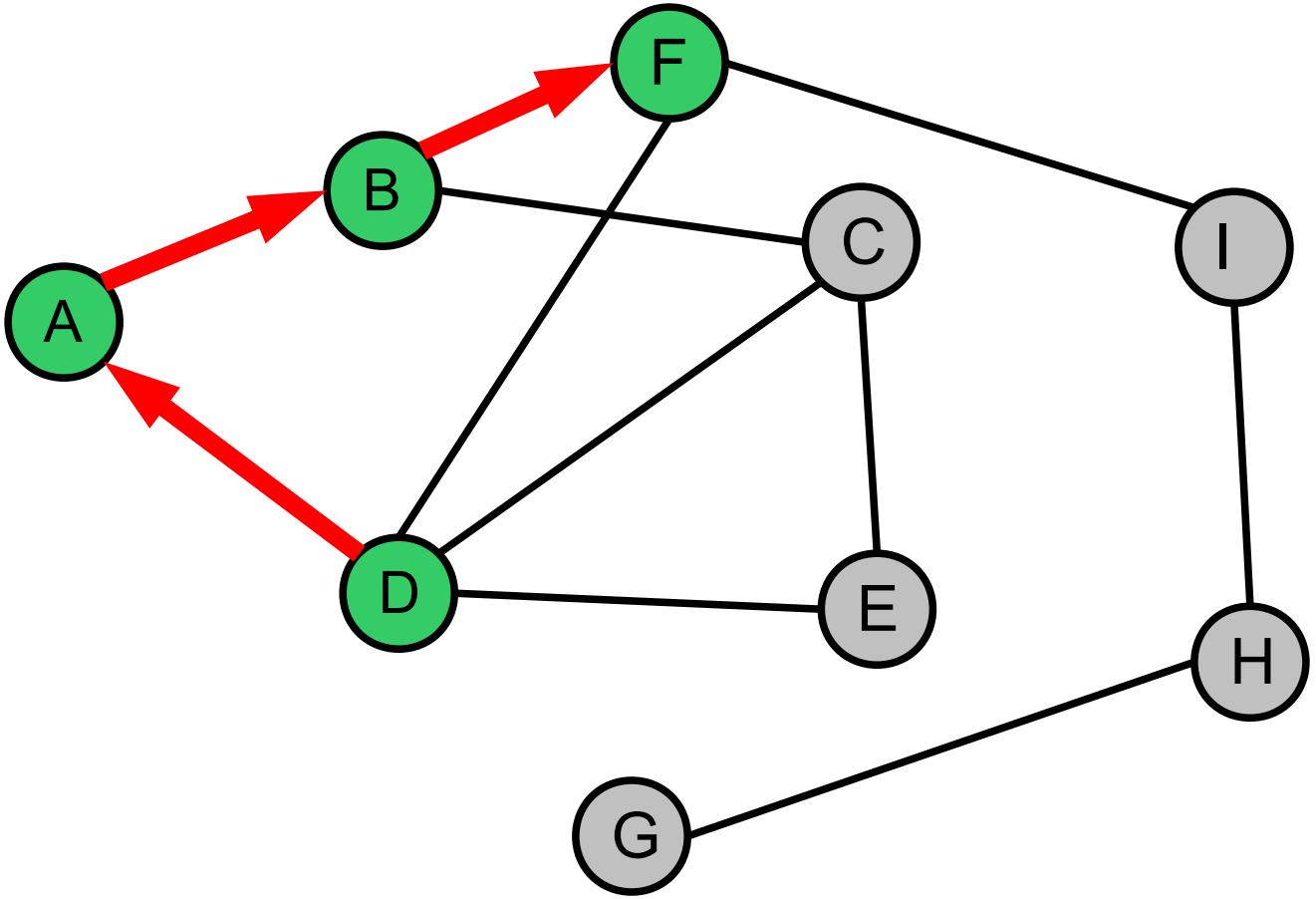
Esempio



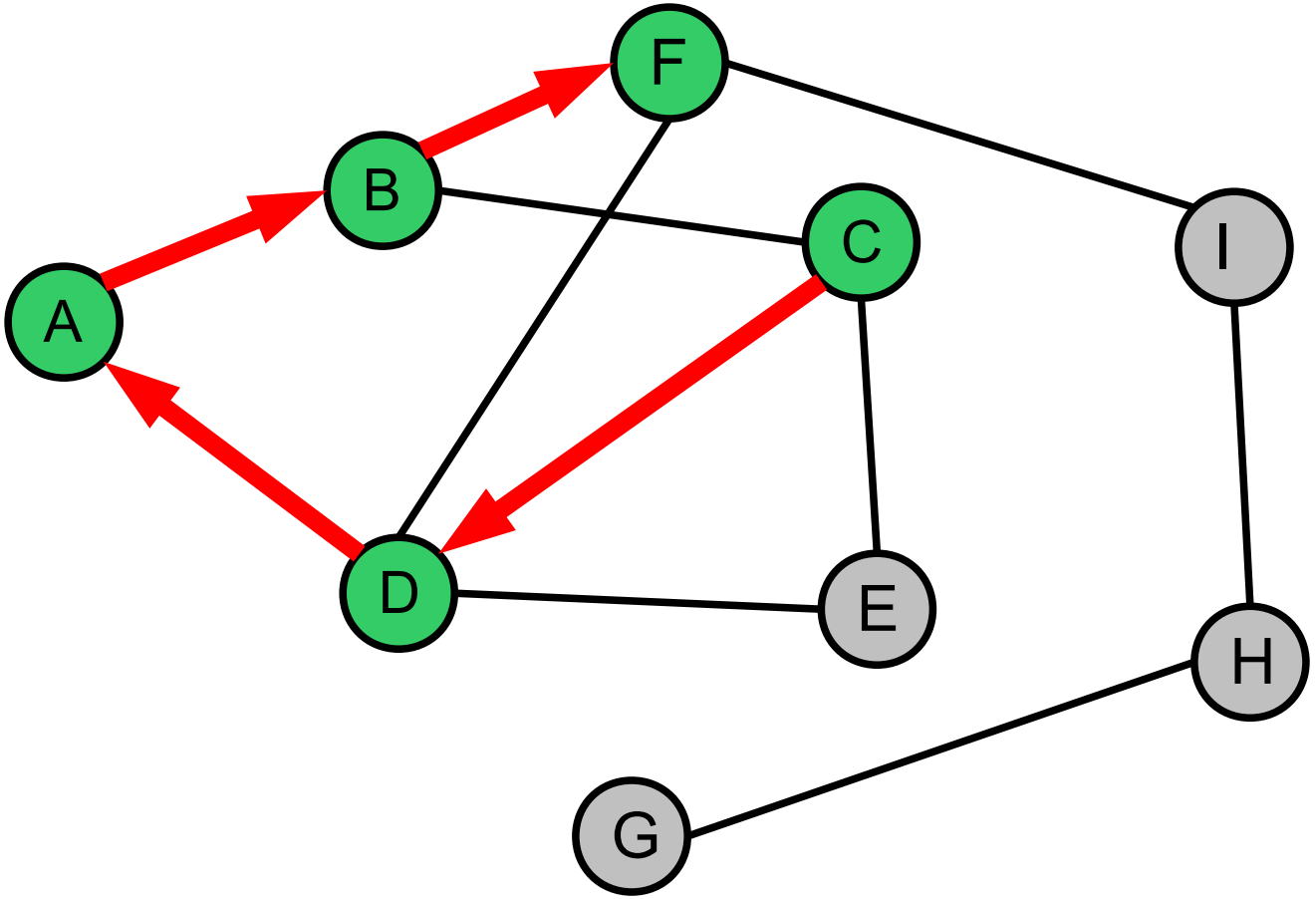
Esempio



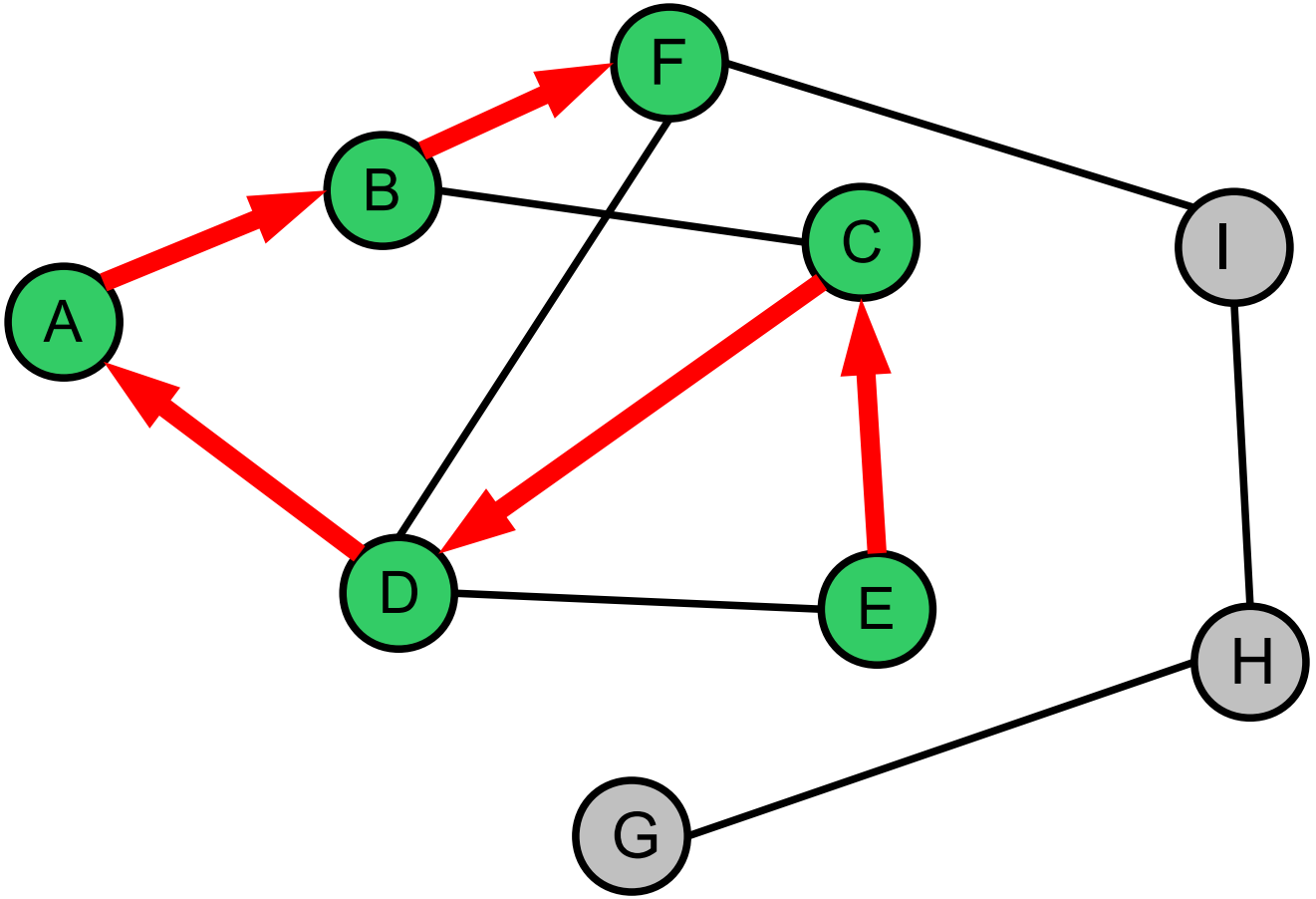
Esempio



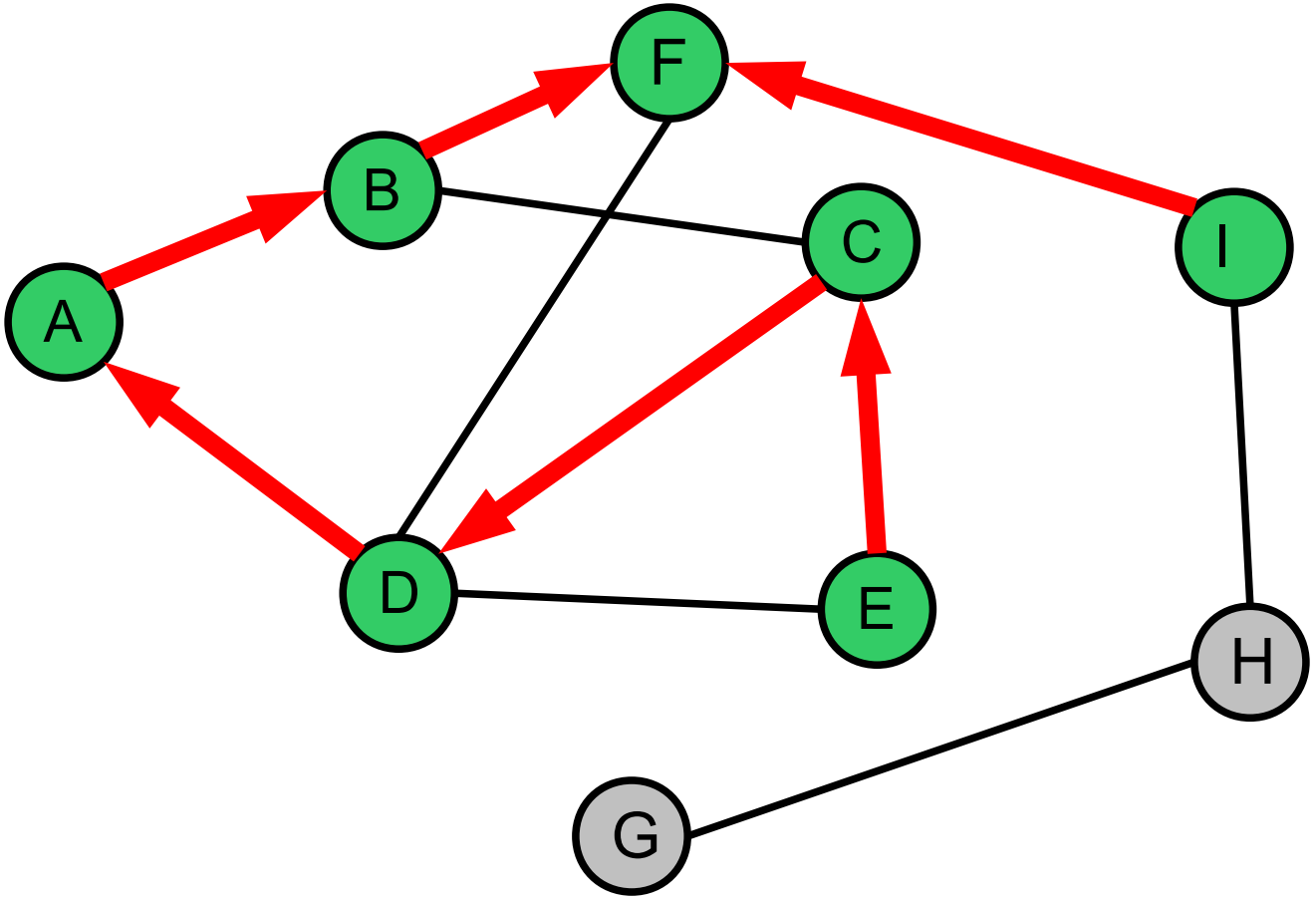
Esempio



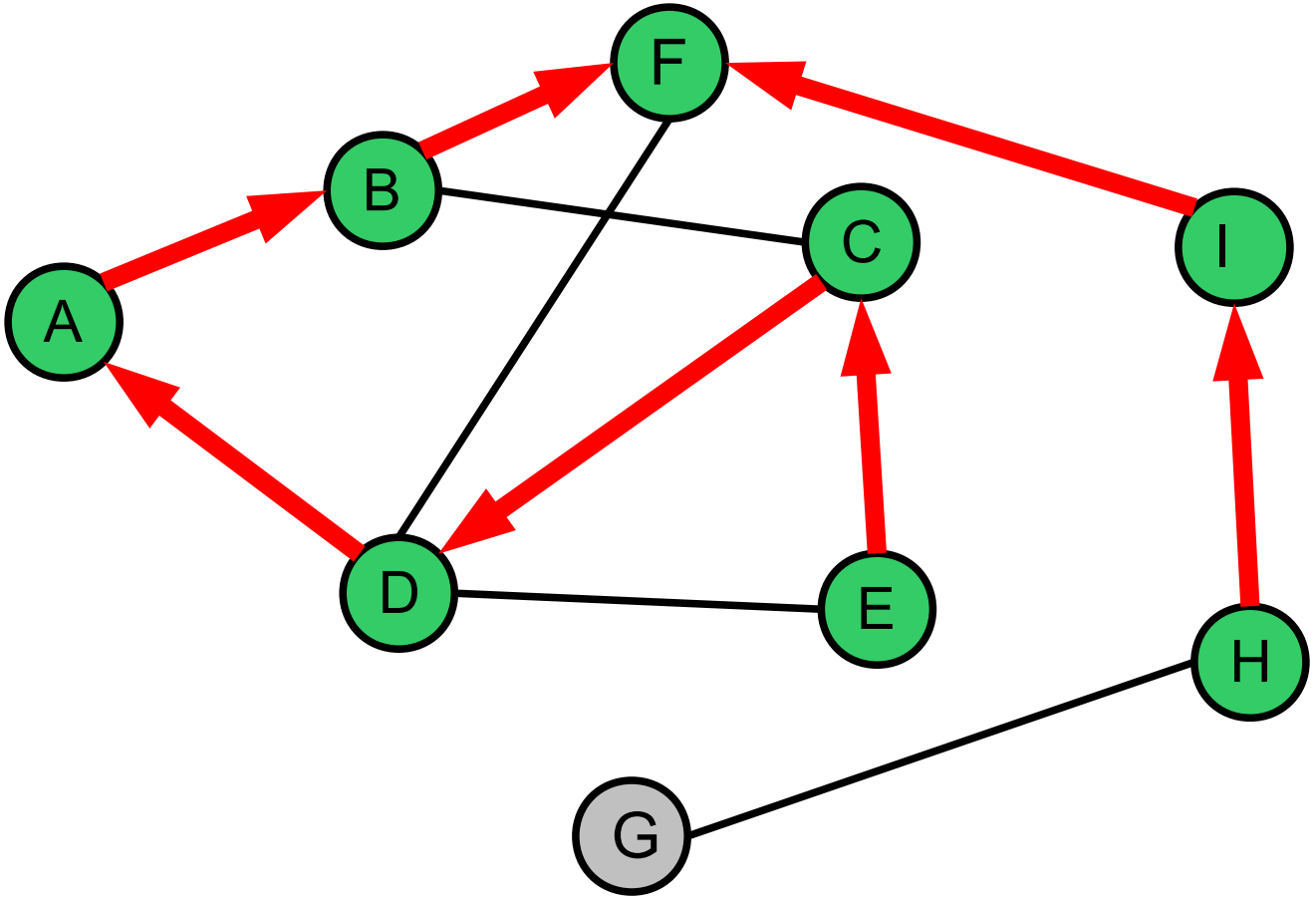
Esempio



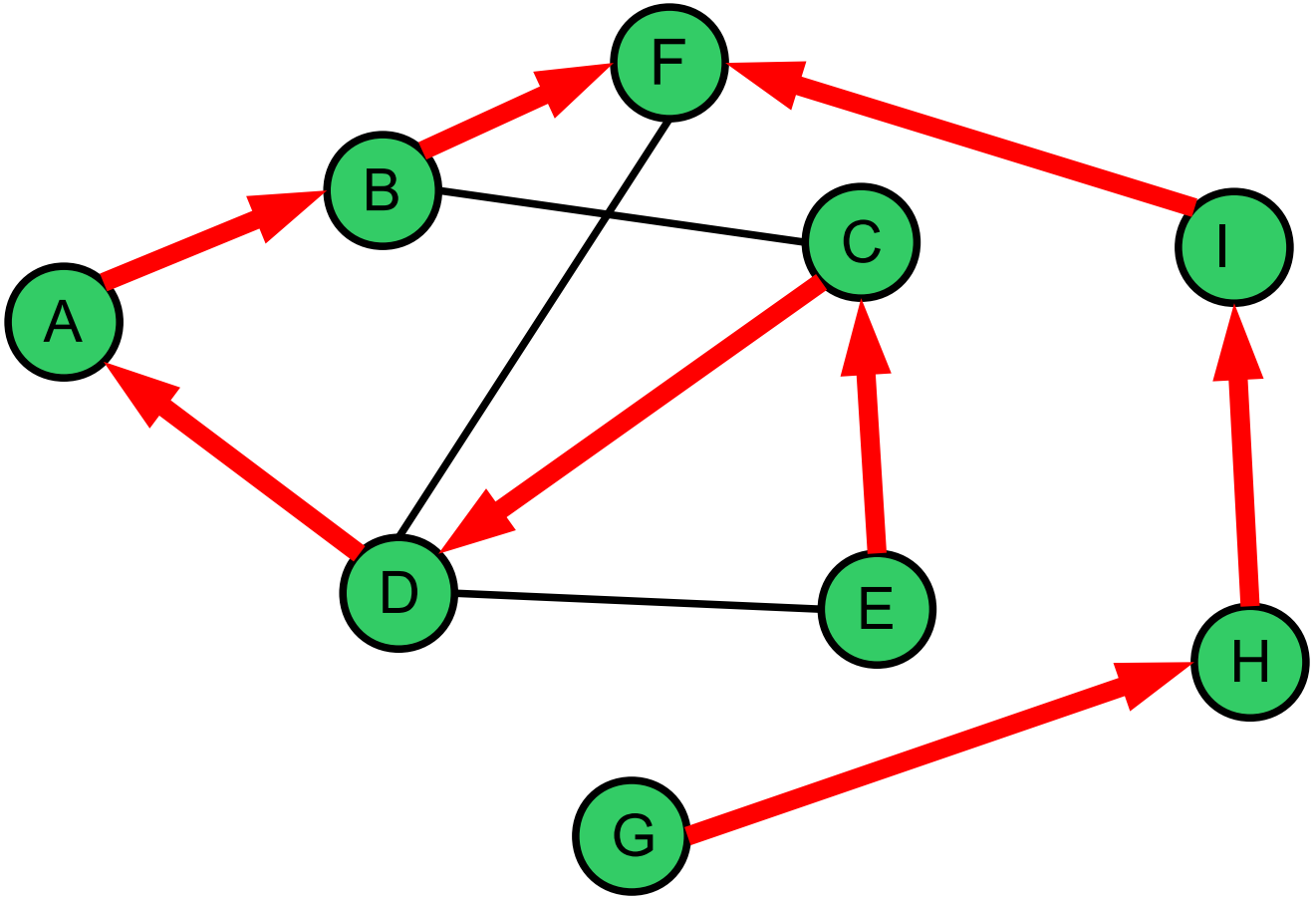
Esempio



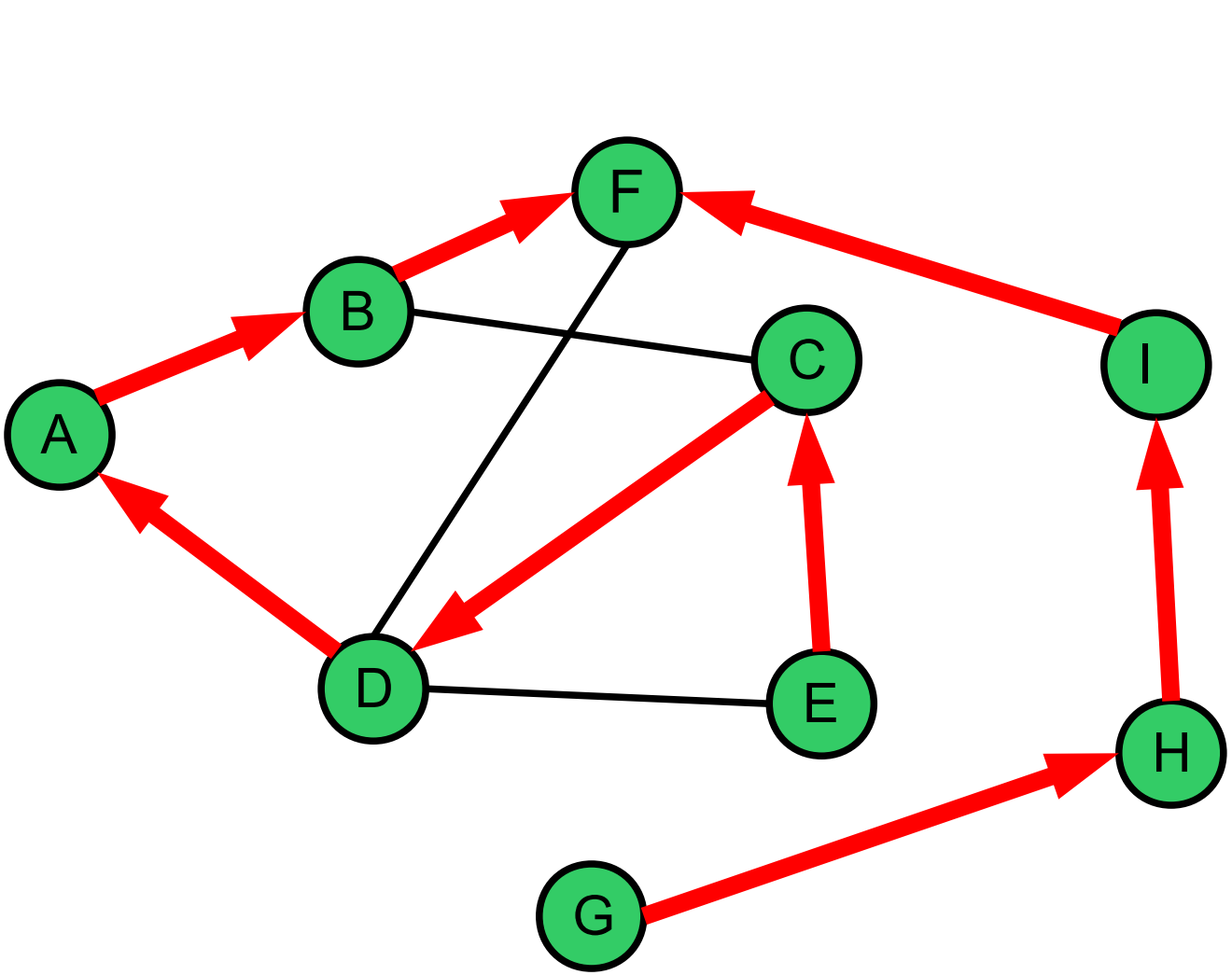
Esempio

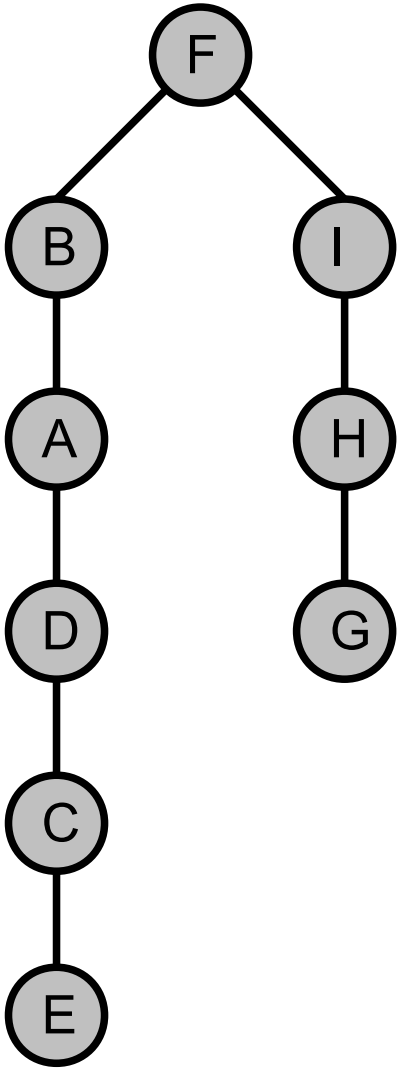


Esempio

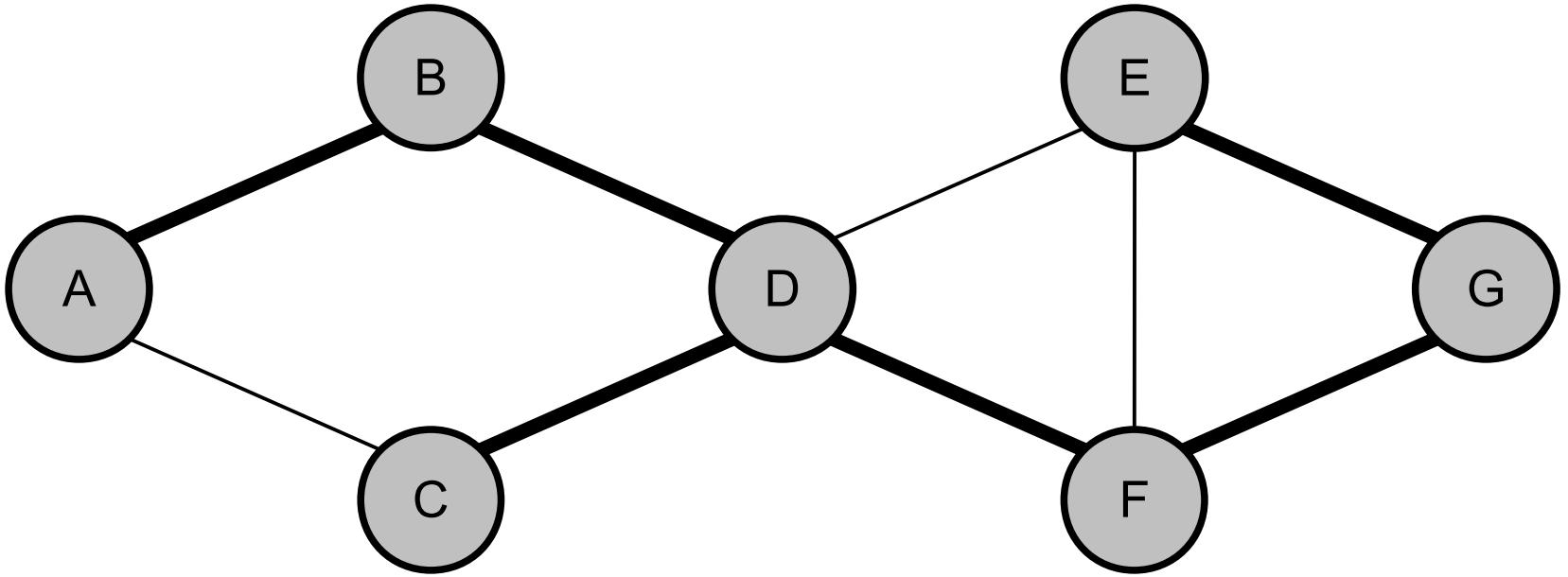


Esempio





Esercizio



Applicazioni degli algoritmi di visita

- Individuare
  - le componenti connesse di un grafo non orientato
  - le componenti fortemente connesse di un grafo orientato

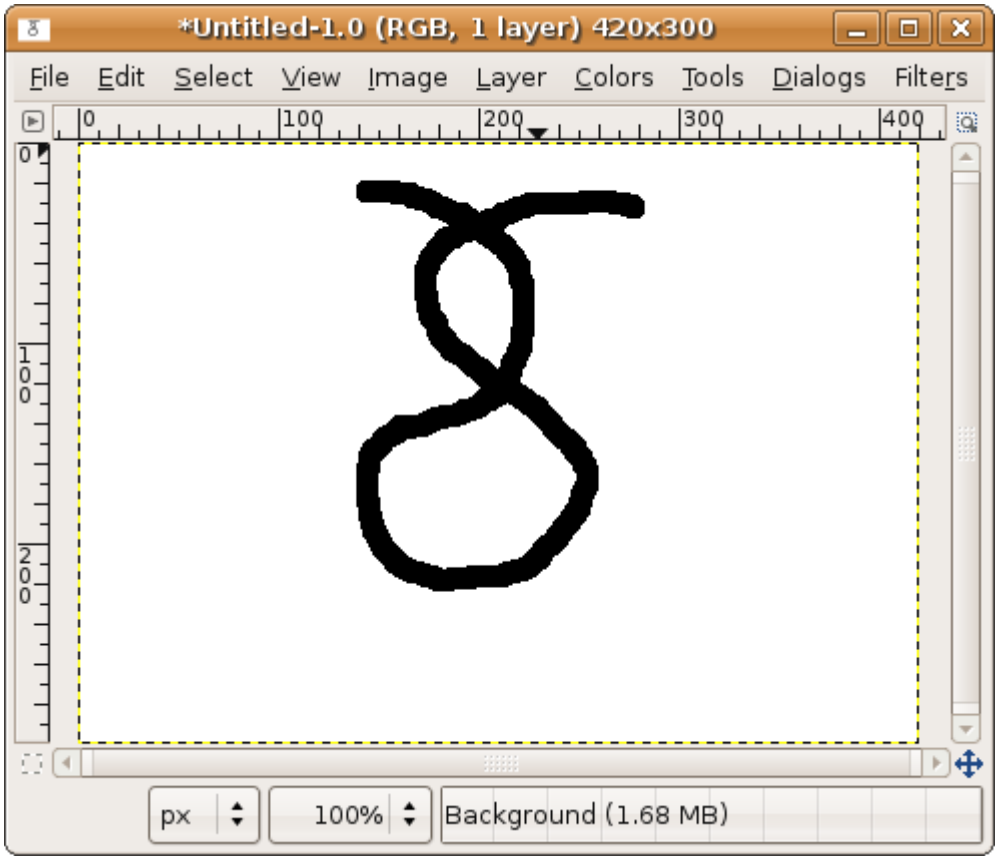
Componenti Connesse

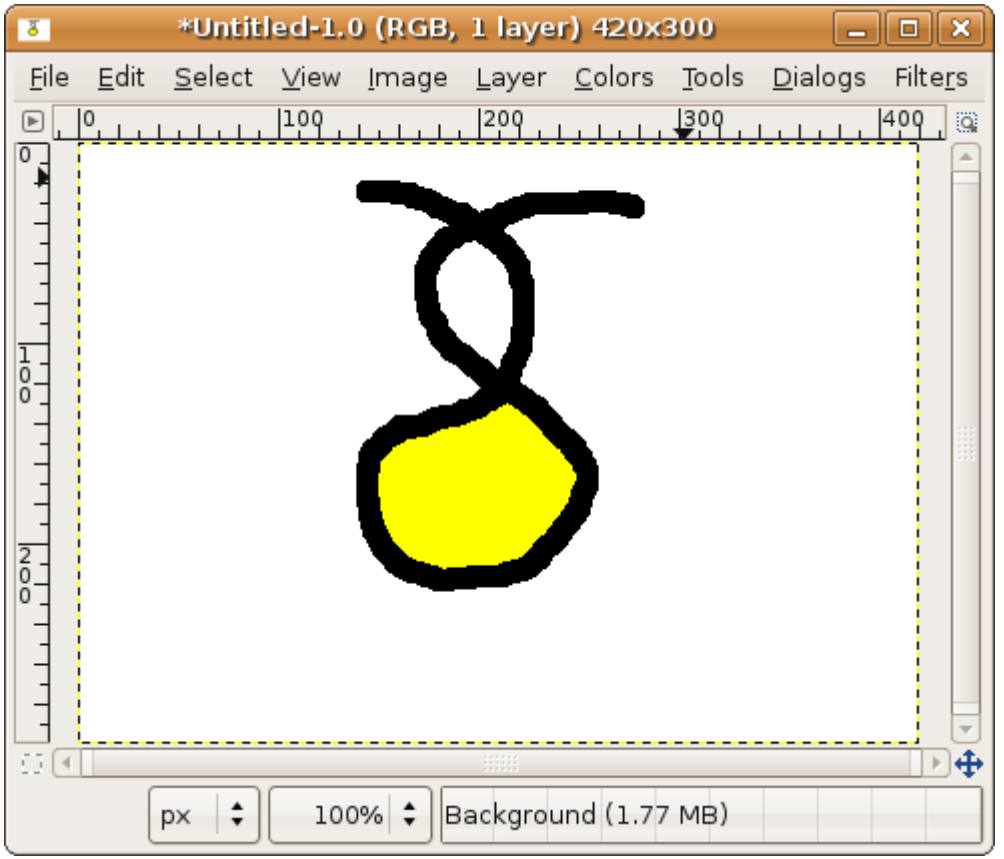
```
algoritmo CC( Grafo G=(V,E) )
  for each v in V do
    v.cc := -1;
    v.parent := NULL;
  endfor
  k := 0;
  for each v in V do
    if (v.cc < 0) then
      CC-visit(G, v,k);
      k := k+1;
    endif
  endfor

algoritmo CC-visit(Grafo G, nodo v, int k)
  v.cc := k;
  for each u adiacente a v do
    if (u.cc < 0) then
      u.parent := v;
      CC-visit(u, k);
    endif
  endfor
```

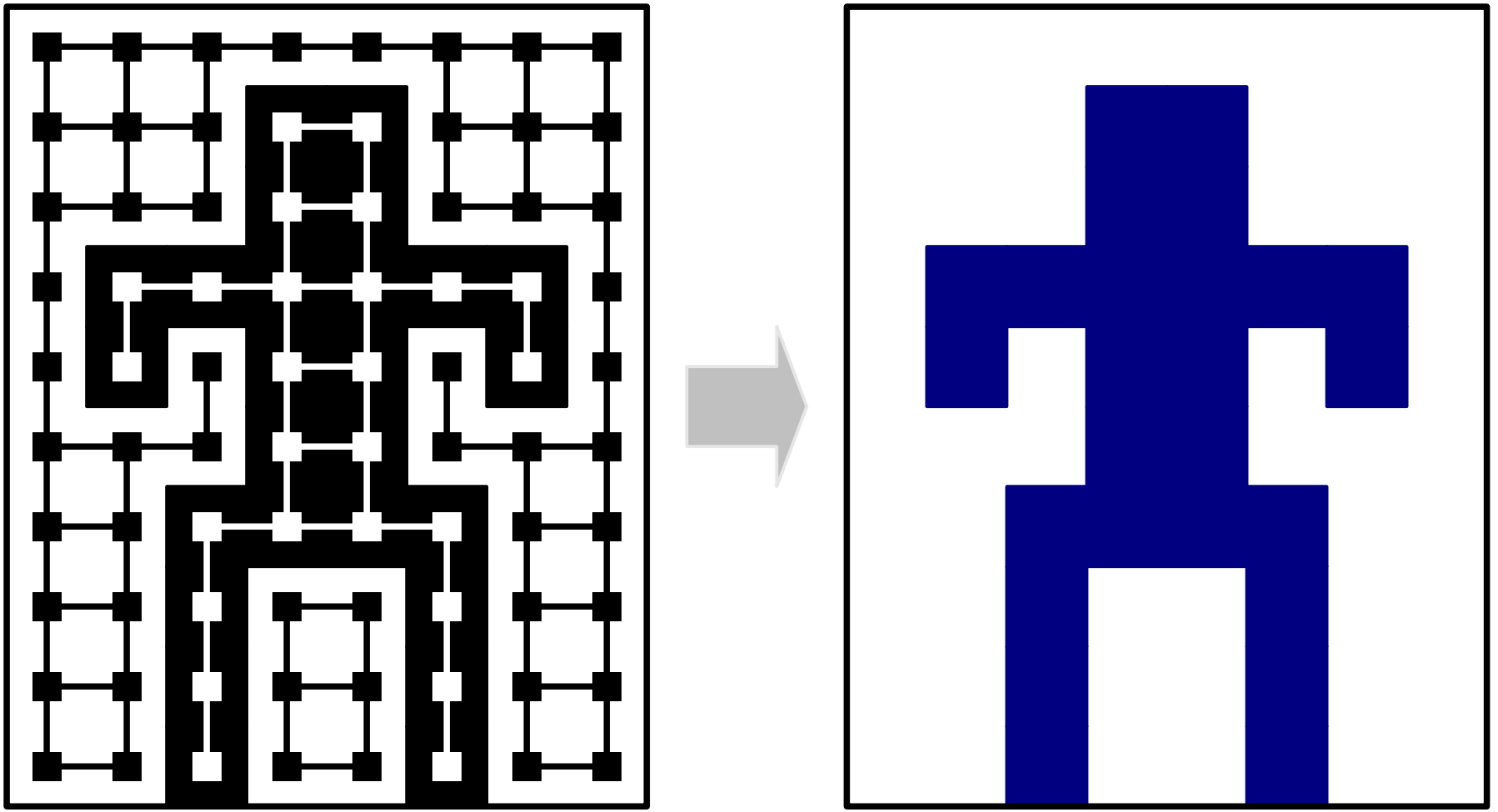
Etichetta con il valore *k* tutti i nodi della stessa componente connessa cui appartiene *v*

Applicazione: floodfill





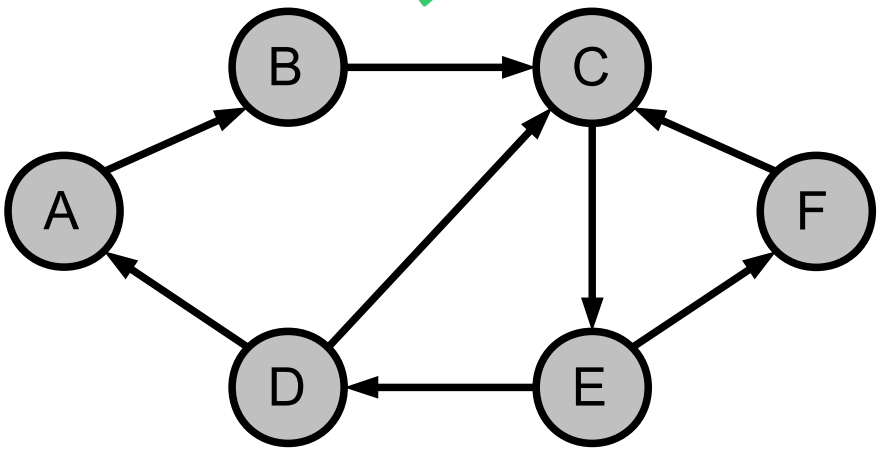
Applicazione: floodfill



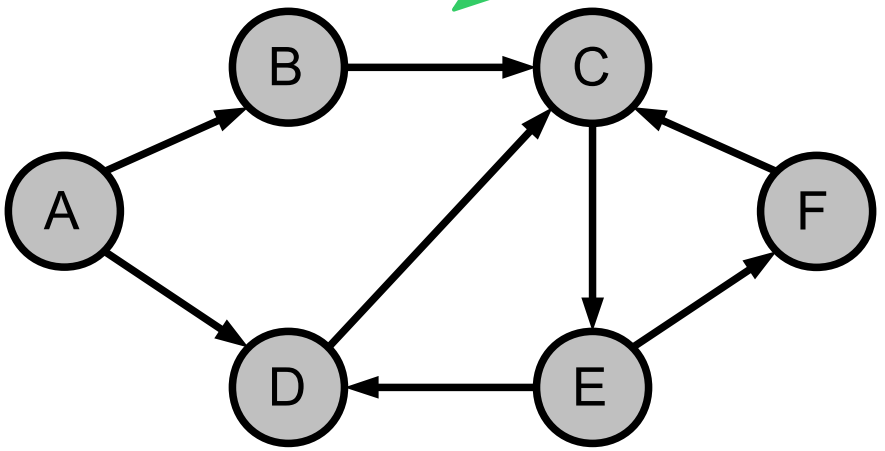
Componenti fortemente connesse
(Strongly Connected Components)

- Un grafo orientato *G* è fortemente connesso se ogni coppia di vertici è connessa da un cammino

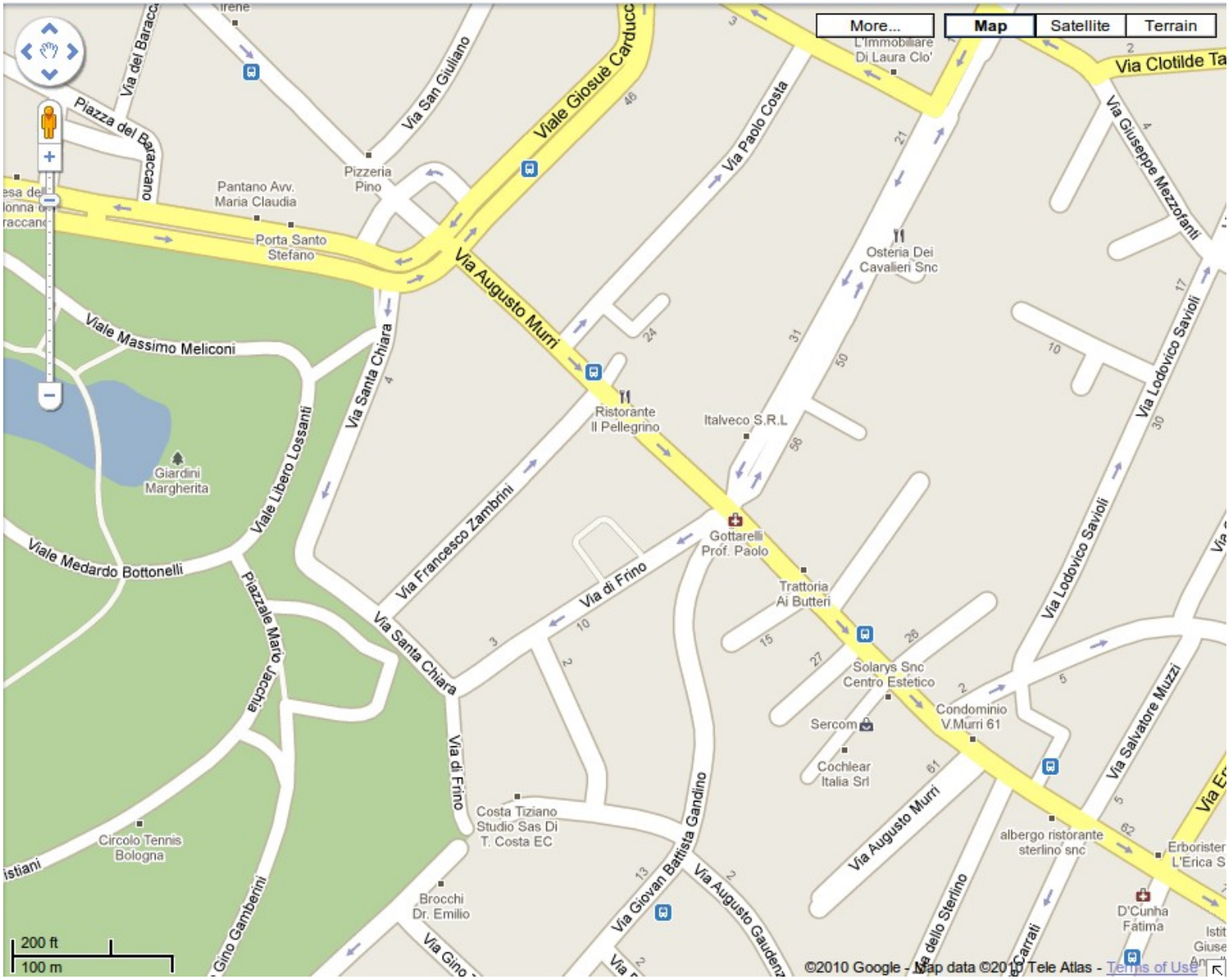
Questo grafo orientato ***è fortemente connesso.***



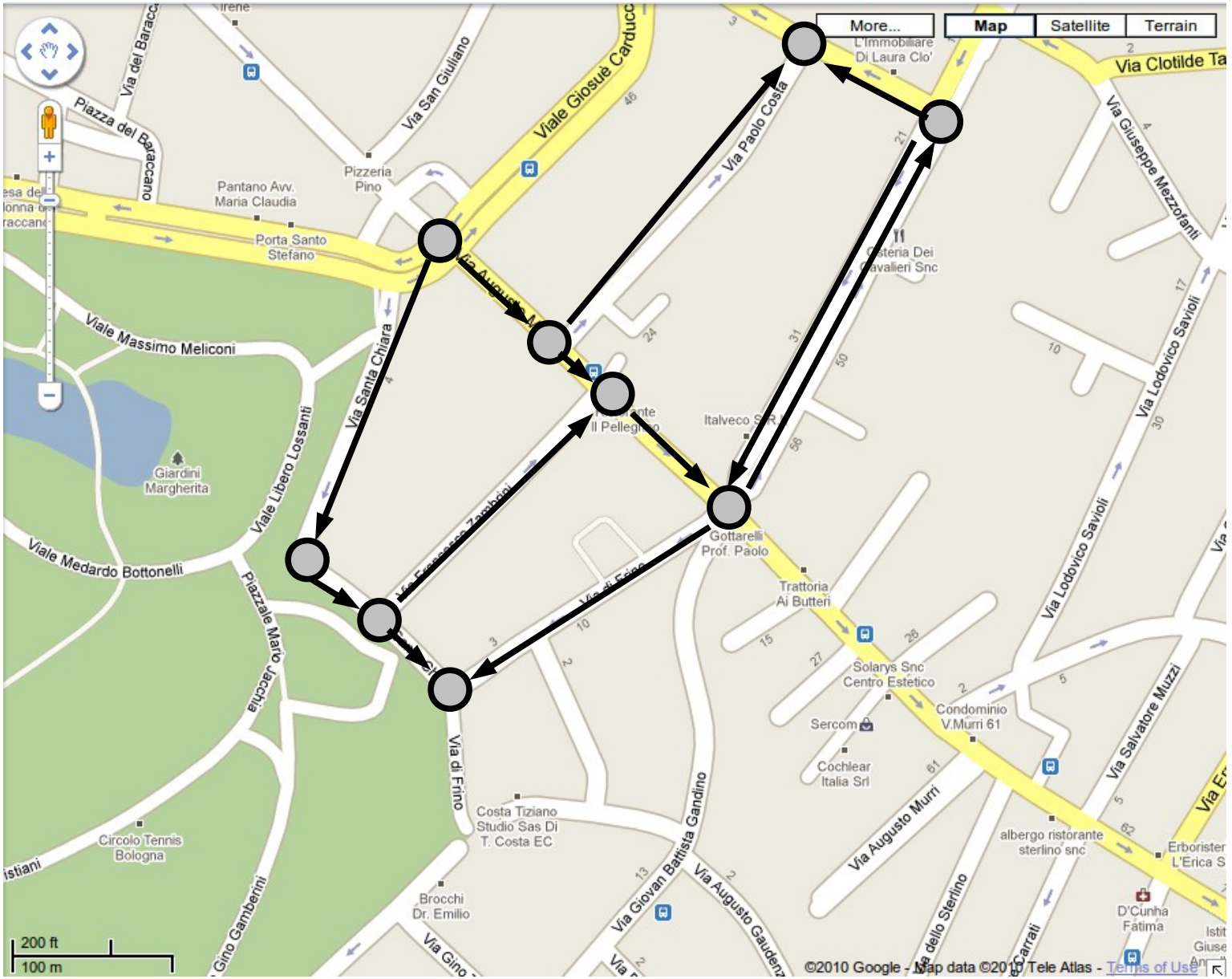
Questo grafo orientato **non è *fortemente connesso*;** ad es., *non esiste cammino da* D *a* A.



Nel mondo reale

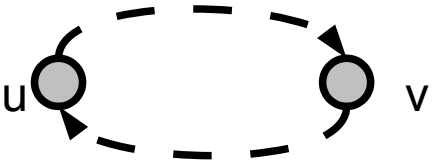


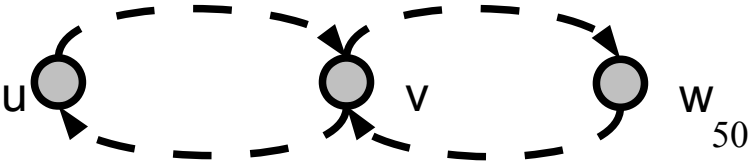
Nel mondo reale



Componenti fortemente connesse (grafo orientato)

- u e v appartengono alla stessa componente fortemente connessa se e solo se esiste un cammino (orientato) che connette *u* con *v* e viceversa
- La relazione di connettività forte è di equivalenza
  - Riflessiva
    - *u* è raggiungibile da se stesso per definizione
  - Simmetrica
    - Se *u* è fortemente connesso a *v*, allora esiste un cammino (orientato) che connette *u* e *v* e viceversa. Quindi anche *v* è fortemente connesso a *u*.
  - Transitiva
    - Se *u* è fortemente connesso a *v*, e *v* è fortemente connesso a *w*, allora *u* è fortemente connesso a *w*.





Idea

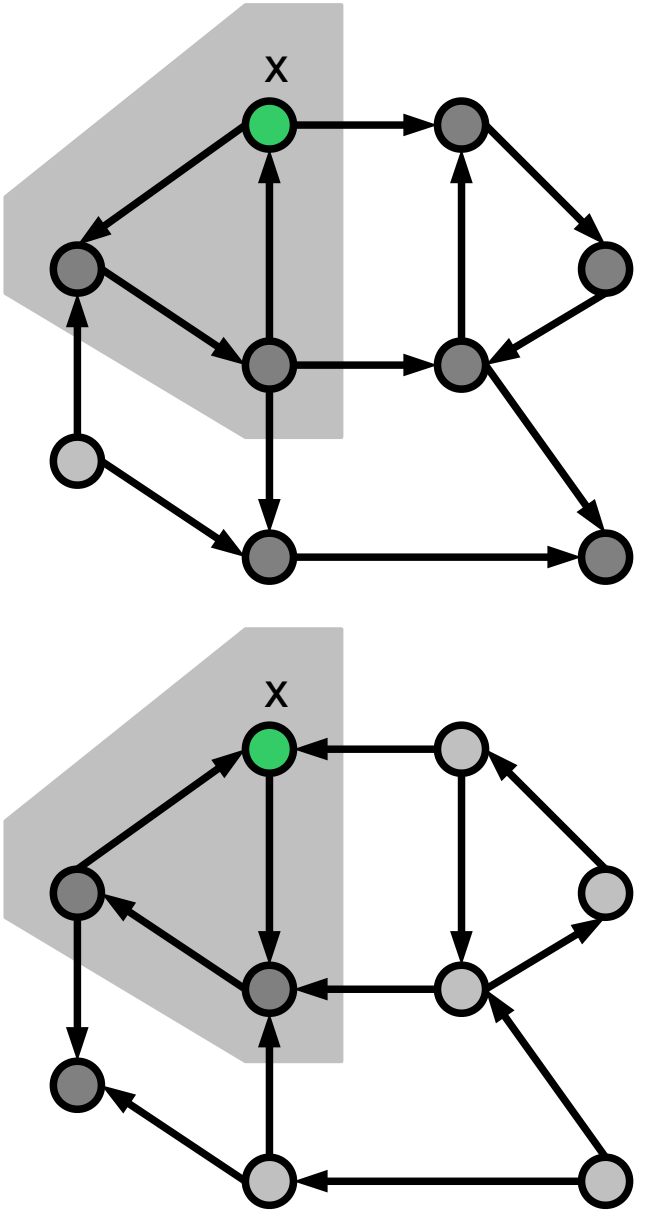
- Due nodi *u* e *v* appartengono alla stessa componente fortemente connessa se e solo se valgono entrambe le seguenti proprietà
  - Esiste un cammino u$\rightarrow\rightarrow$v
    - cioè v è discendente di u in una visita DFS che usa u come sorgente
  - Esiste un cammino v$\rightarrow\rightarrow$u
    - cioè u è discendente di v in una visita DFS che usa v come sorgente

Idea

- $A(x)$ = insieme degli antenati del nodo x
  - cioè insieme di tutti i nodi da cui si può raggiungere x
- $D(x)$ = insieme dei discendenti del nodo x
  - cioè insieme di tutti i nodi che si possono raggiungere da x
- Per individuare la componente fortemente connessa cui appartiene x, è sufficiente calcolare l'intersezione $A(x) \cap D(x)$

Idea

- Come calcolare D(x)?
  - D(x) include i nodi raggiungibili da una visita (DFS o BFS) usando x come sorgente
- Come calcolare A(x)?
  - È sufficiente invertire la direzione di tutti gli archi, ed effettuare una nuova visita (DFS o BFS) usando ancora x come sorgente
- Nota: il calcolo di A(x) o D(x) richiede tempo O(n+m)



Schema di soluzione

```
algoritmo SCC(Grafo G, nodo x)→lista di nodi
    L := lista vuota di nodi
(1) Esegui DFS(G, x) marcando i nodi visitati
(2) Calcola il grafo trasposto G^T
        (inverti la direzione degli archi di G)
(3) Esegui DFS(G^T, x), mettendo in L i nodi
        visitati che sono stati marcati durante (1)
    return L
```

- (1) costa O(n+m)
- (2) costa O(n+m)
- (3) costa O(n+m)

Calcolo di tutte le componenti fortemente connesse

- Per calcolare tutte le SCC di un grafo G è necessario eseguire l'algoritmo SCC(G, x) per ogni nodo $x \in V$
  - Ogni esecuzione di SCC(G, x) costa O(n+m)
- Costo complessivo: O(nm+n^2)
  - Esiste un algoritmo più sofisticato che elenca tutte le SCC di un grafo G in tempo complessivo O(n+m)