Algoritmi di Visita di Grafi

Gianluigi Zavattaro
Dip. di Informatica – Scienza e Ingegneria
Università di Bologna
gianluigi.zavattaro@unibo.it

Attraversamento grafi

- Definizione del problema
  - Dato un grafo *G=<V,E>* ed un vertice *s* di V (detto *sorgente*), visitare ogni vertice raggiungibile nel grafo dal vertice *s*
  - Ogni nodo deve essere visitato una volta sola
- Visita in ampiezza (breadth-first search)
  - Visita i nodi "espandendo" la frontiera fra nodi scoperti / da scoprire
  - Es: *Cammini di lunghezza minima 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"

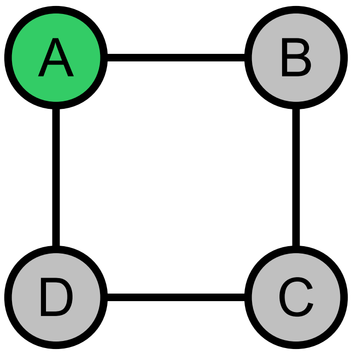
- Prendere ispirazione dalla visita degli alberi
- Ad esempio:
  - utilizziamo una visita BFS basata su coda
  - trattiamo i "vertici adiacenti" come se fossero i "figli"

```
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?

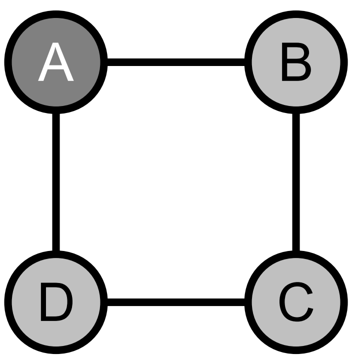
```
algoritmo non-visita(grafo G, nodo s)
  coda := {s}
  while (coda ≠ ∅) do
    u = coda.dequeue()
    "visita u"
    for each "nodo v adiacente a u" do
      coda.enqueue(v)
    endfor
  endwhile
```



coda = { A }

Perché non funziona?

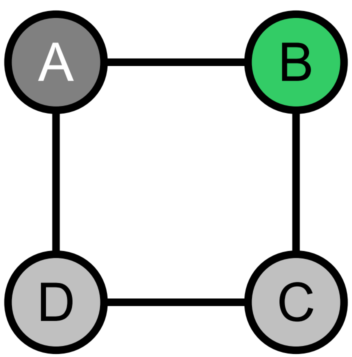
```
algoritmo non-visita(grafo G, nodo s)
  coda := {s}
  while (coda ≠ ∅) do
    u = coda.dequeue()
    "visita u"
    for each "nodo v adiacente a u" do
      coda.enqueue(v)
    endfor
  endwhile
```



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

Perché non funziona?

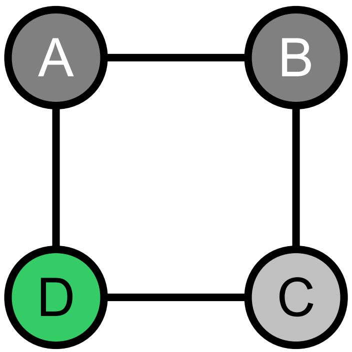
```
algoritmo non-visita(grafo G, nodo s)
  coda := {s}
  while (coda ≠ ∅) do
    u = coda.dequeue()
    "visita u"
    for each "nodo v adiacente a u" do
      coda.enqueue(v)
    endfor
  endwhile
```



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 <u>non termina</u> se applicato a grafi con cicli

Idea

- L'algoritmo esplora il grafo a partire da un nodo s
  - Costruisce passo-passo un albero T radicato in s che al termine contiene tutti i nodi raggiungibili a partire da s
- Durante l'algoritmo ogni vertice del grafo può essere
  - inesplorato: Il vertice non è ancora stato incontrato
  - aperto: l'algoritmo ha incontrato il vertice la prima volta
  - chiuso: il vertice è stato visitato completamente (tutti gli archi incidenti sono stati esplorati)
- L'algoritmo mantiene un sottoinsieme frontiera $F \subseteq T$
  - Se un nodo *v* sta in T-F, significa che tutti gli archi incidenti sono stati esplorati (*v* è chiuso)
  - Se un nodo *v* sta in F, non tutti gli archi sono stati esplorati (*v* è aperto)
  - Se un nodo non sta in T, allora è inesplorato

Algoritmo generico per la visita

```
algoritmo visita(G, s)→ albero
  rendi "non marcati" tutti i vertici
  T := s
  F := { s }
  "marca" il vertice s
  while (F ≠ ∅)  do
    u := F.extract()
    "visita il vertice u"
    for each v adiacente a u do
      if (v non è marcato) then
        marca il vertice v
        T := T ∪ 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*

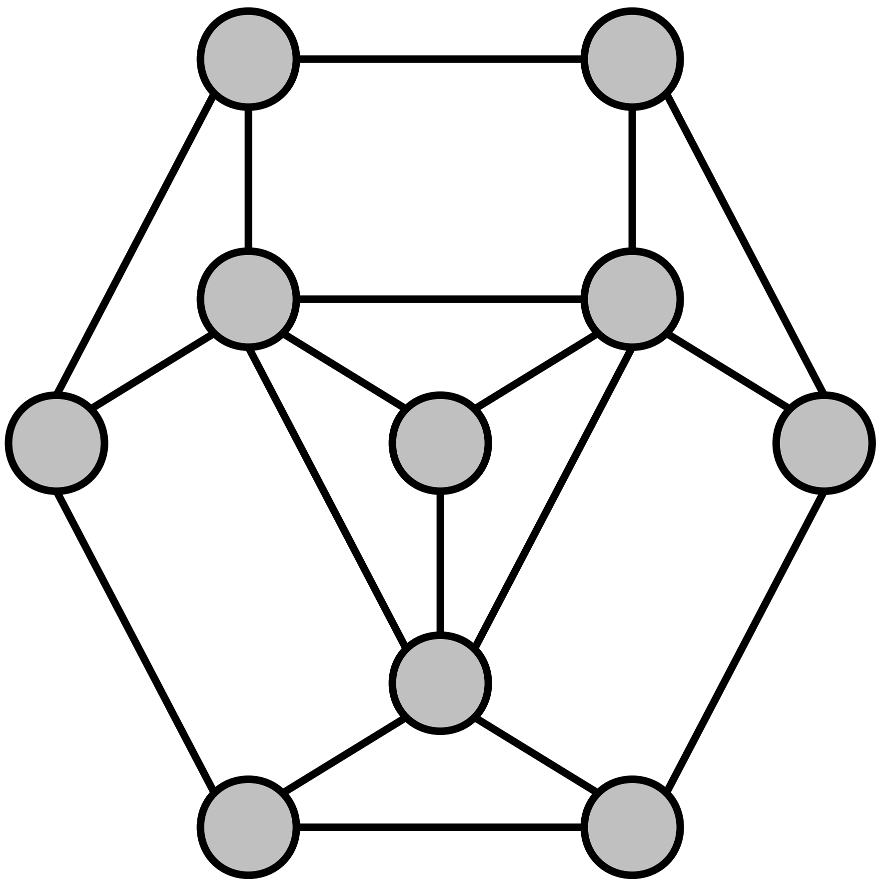
Algoritmo generico per la visita

- 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
- Complessità
  - $O(n+m)$ liste di adiacenza
  - $O(n^2)$ matrice di adiacenza
  - n è il numero di vertici, m è il numero di archi

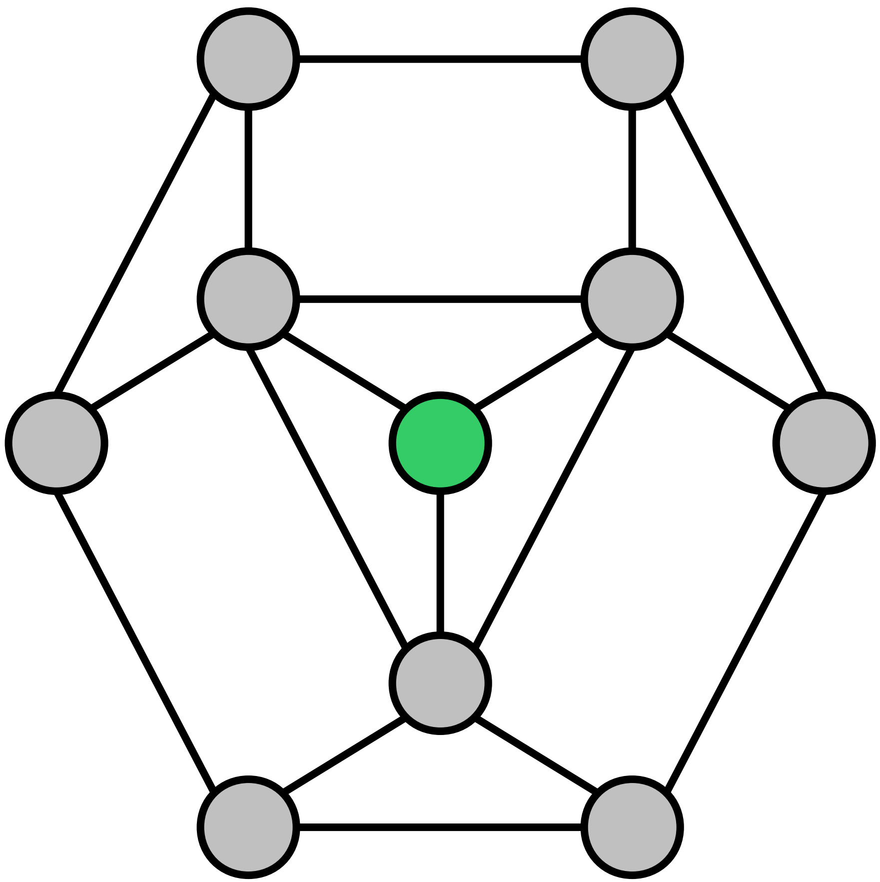
Visita in ampiezza (breadth first search, BFS)

- Cosa vogliamo fare?
  - Visitare i nodi a distanze crescenti dalla sorgente
    - visitare i nodi a distanza k prima di visitare i nodi a distanza k+1
  - Generare 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 corrisponde al cammino più breve nel grafo
  - Calcolare la distanza minima da *s* a tutti i vertici raggiungibili
    - numero di archi attraversati per andare da *s* ad un vertice raggiungibile a partire da *s*

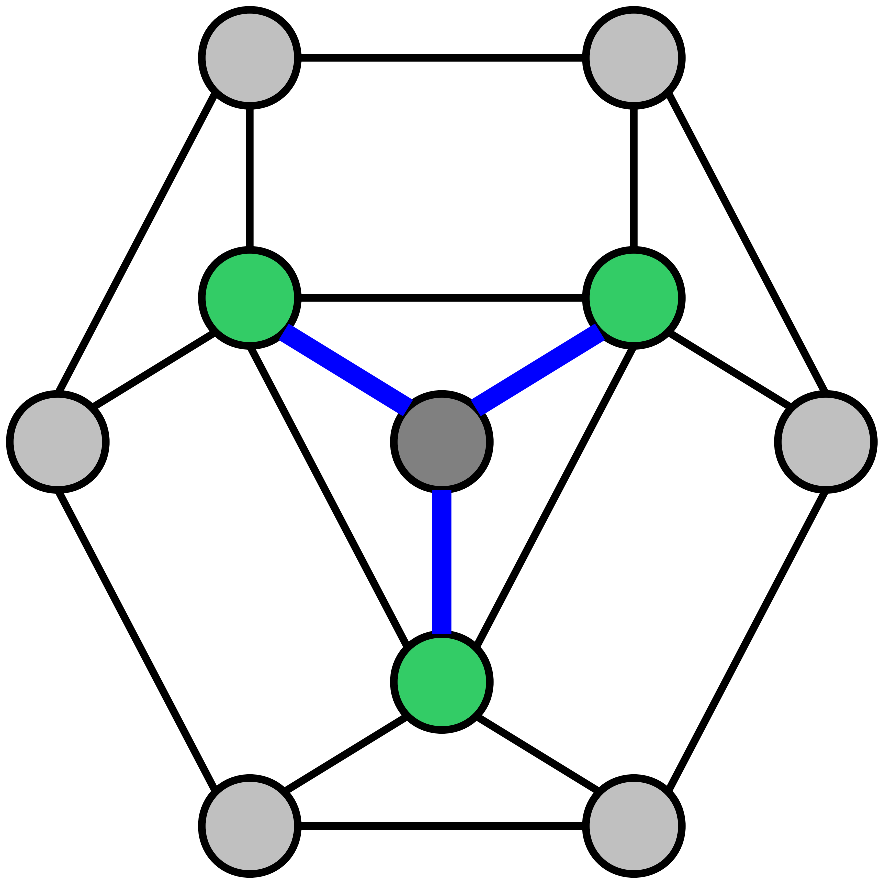
Visita in ampiezza (breadth first search, BFS)



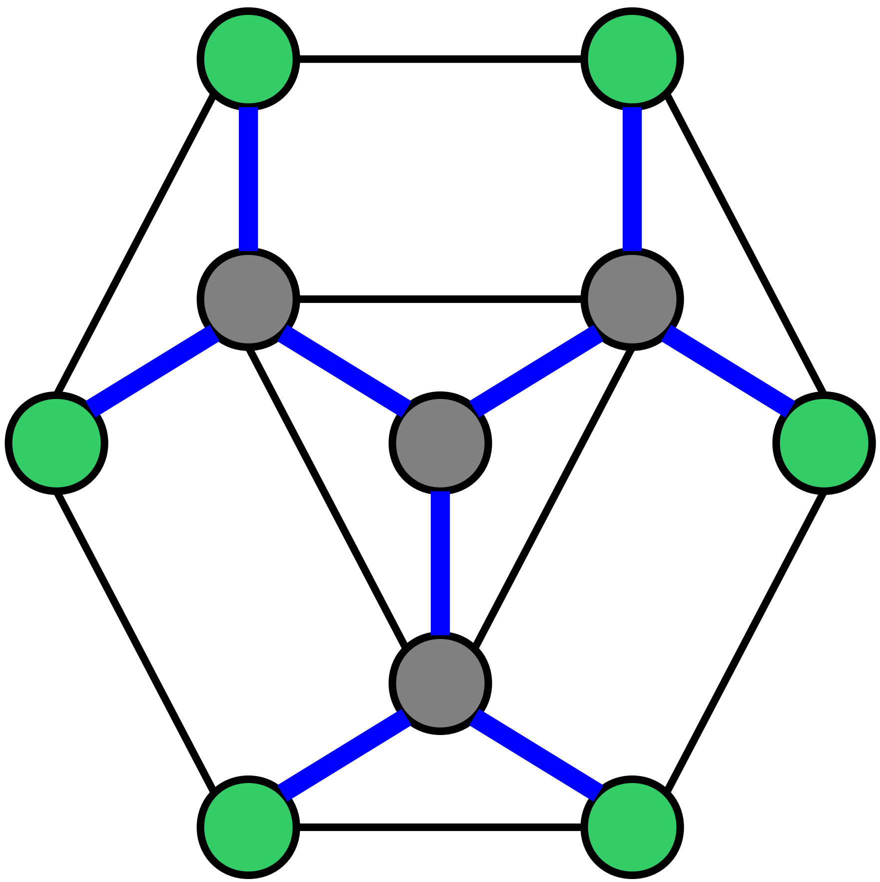
Visita in ampiezza
(breadth first search, BFS)



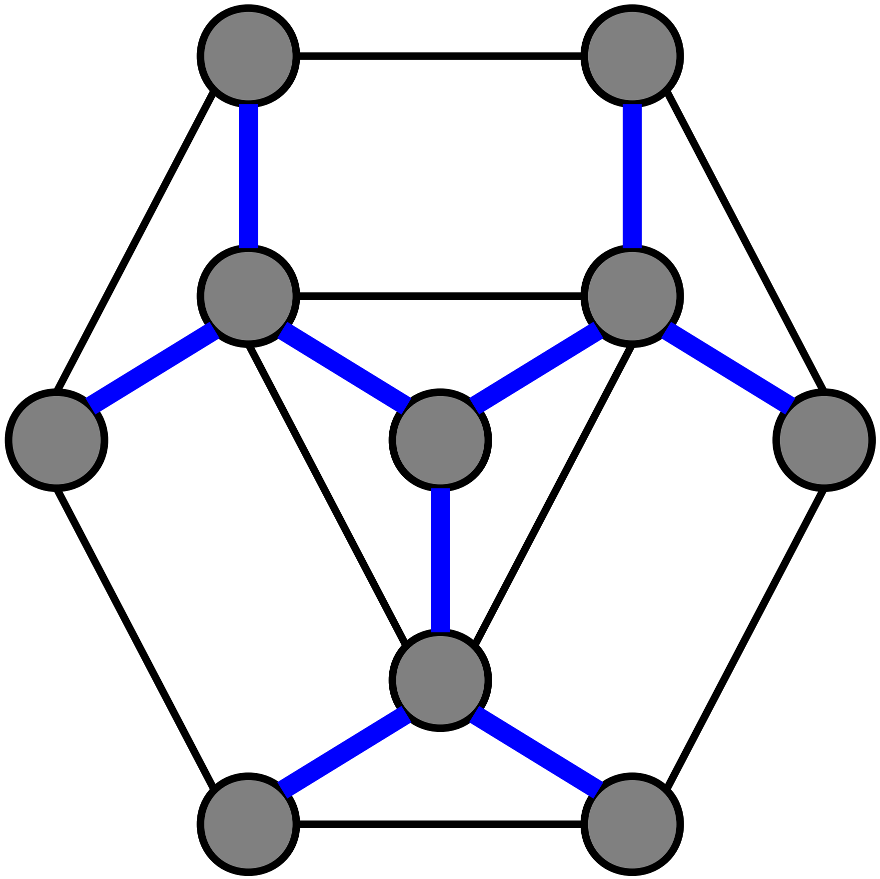
Visita in ampiezza (breadth first search, BFS)



Visita in ampiezza (breadth first search, BFS)



Visita in ampiezza (breadth first search, BFS)

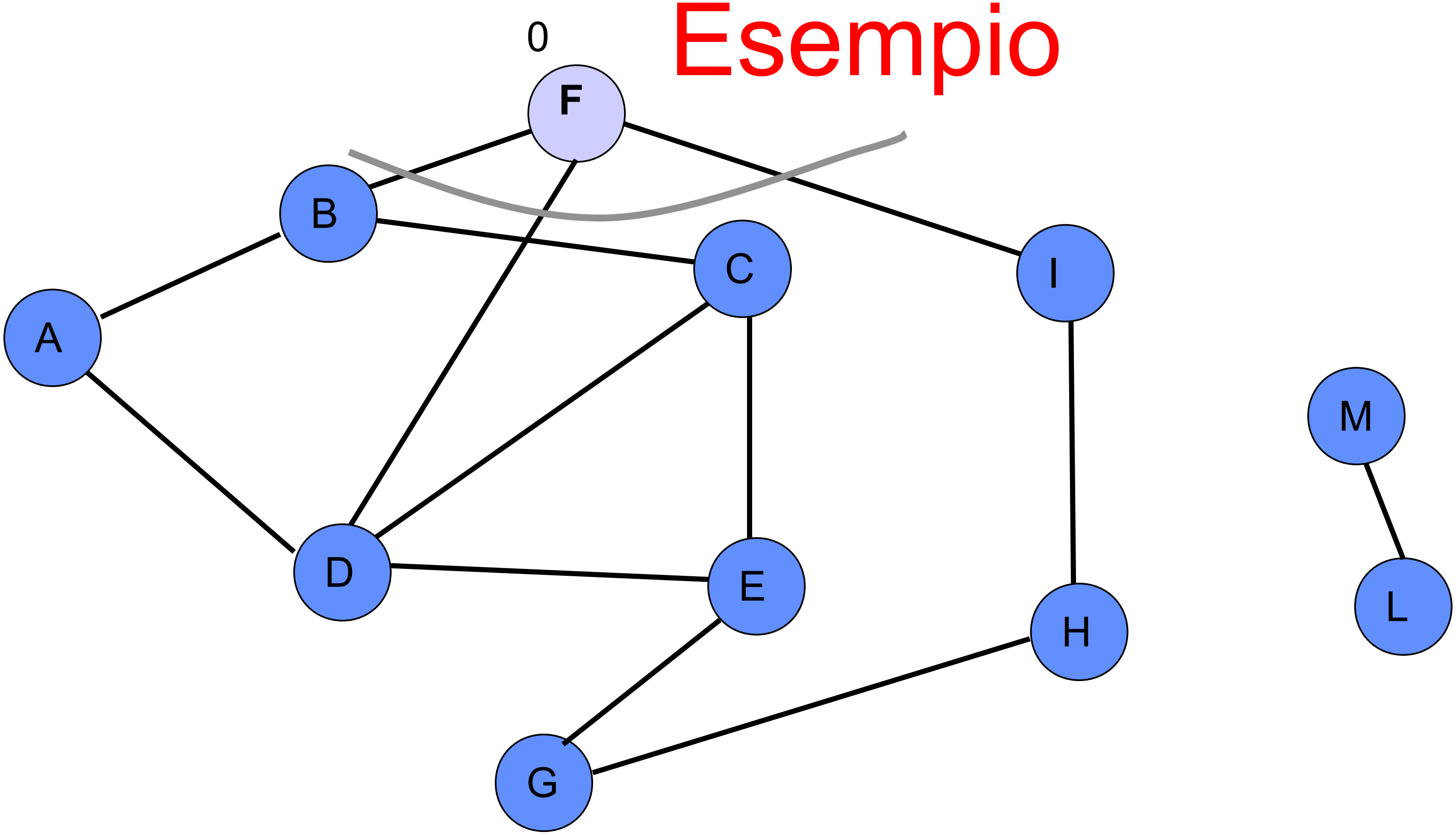


Visita in ampiezza (breadth first search, BFS)

```
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
```

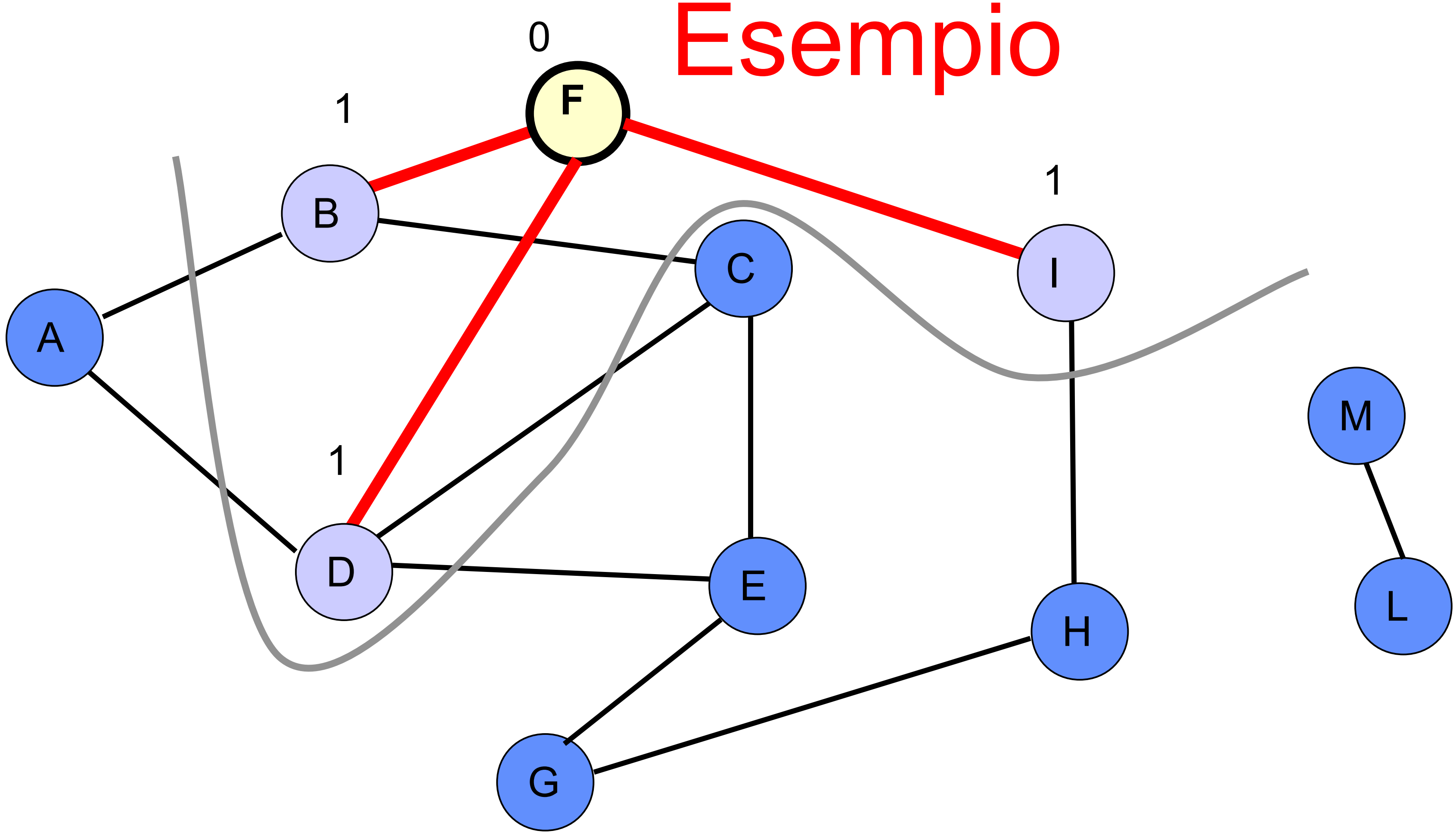
- Insieme *F* gestito tramite una coda
- *v.mark* è la marcatura del nodo *v*
- *v.dist* è la distanza del nodo *v* dal vertice *s*

Esempio



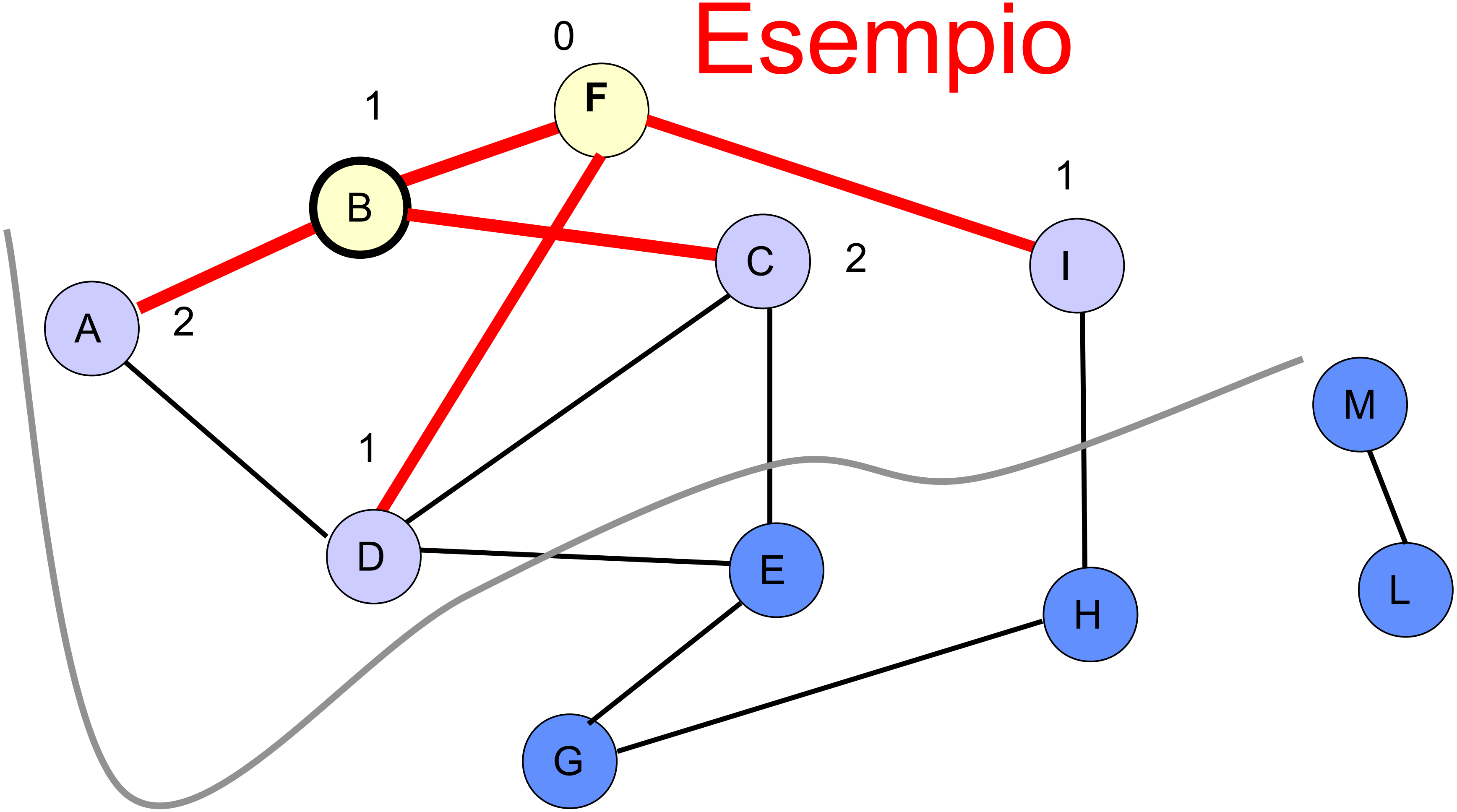
Coda:{F}

Esempio



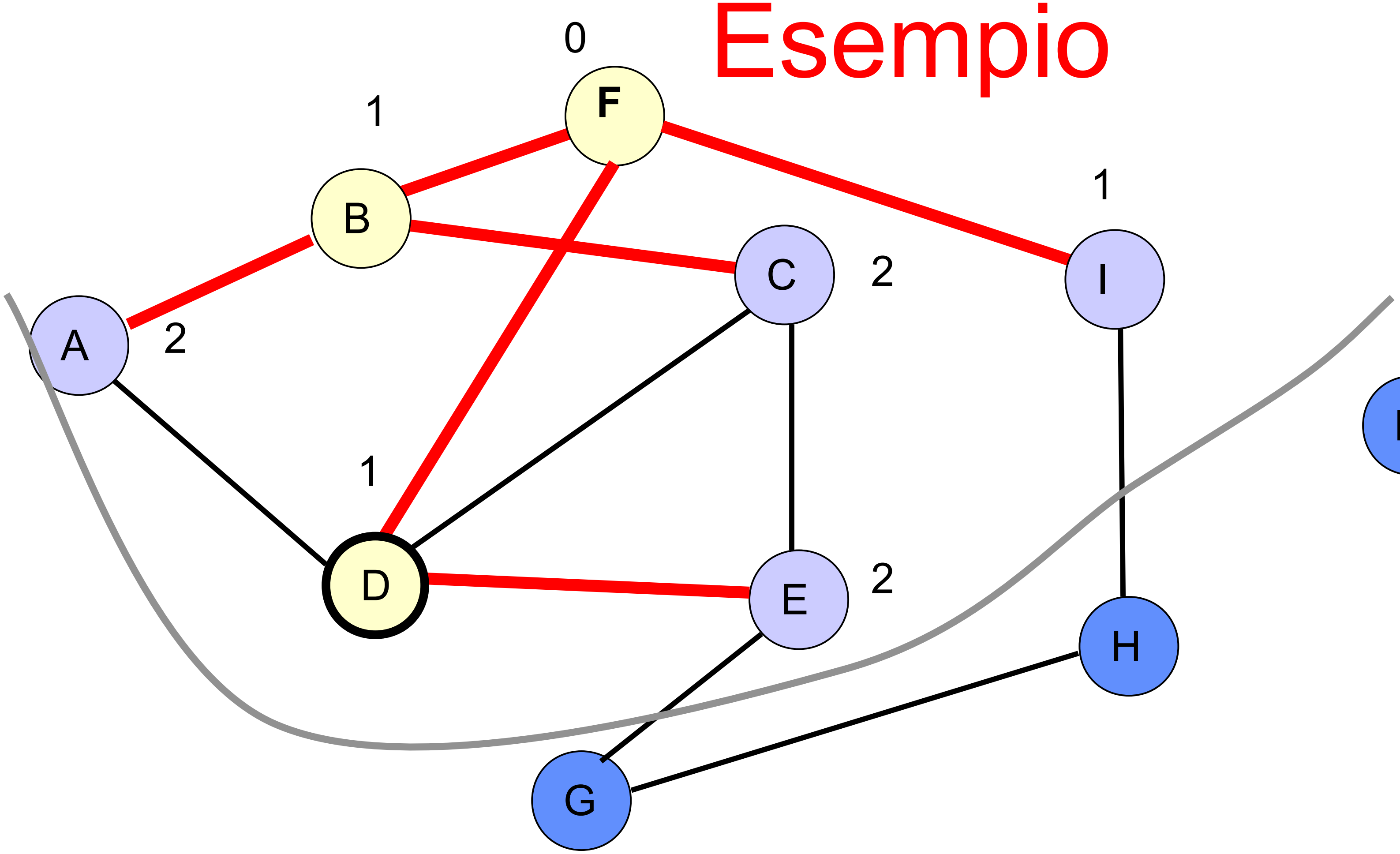
Coda:{B,D,I}

Esempio



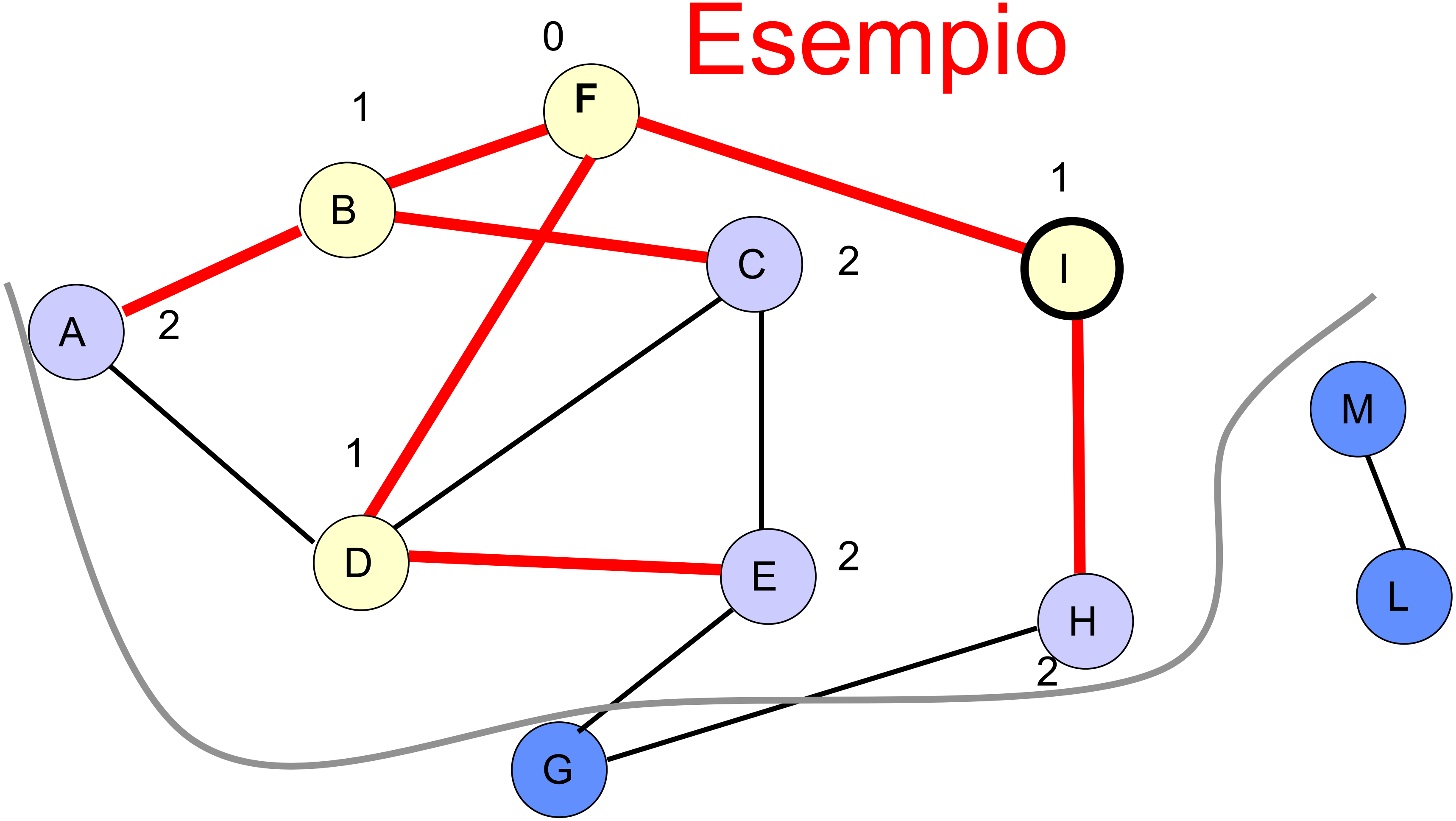
Coda:{D,I,C,A}

Esempio



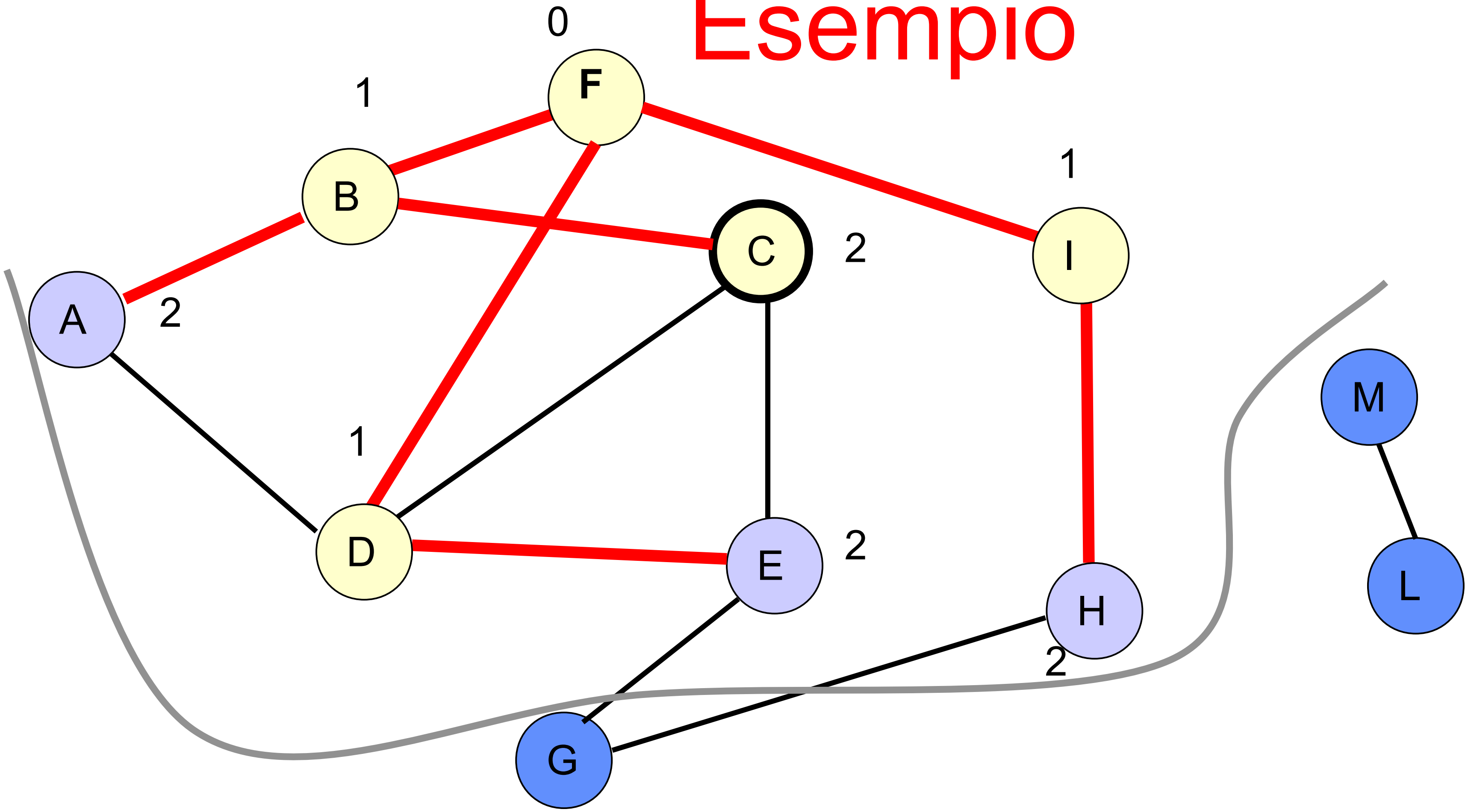
Coda:{I,C,A,E}

Esempio



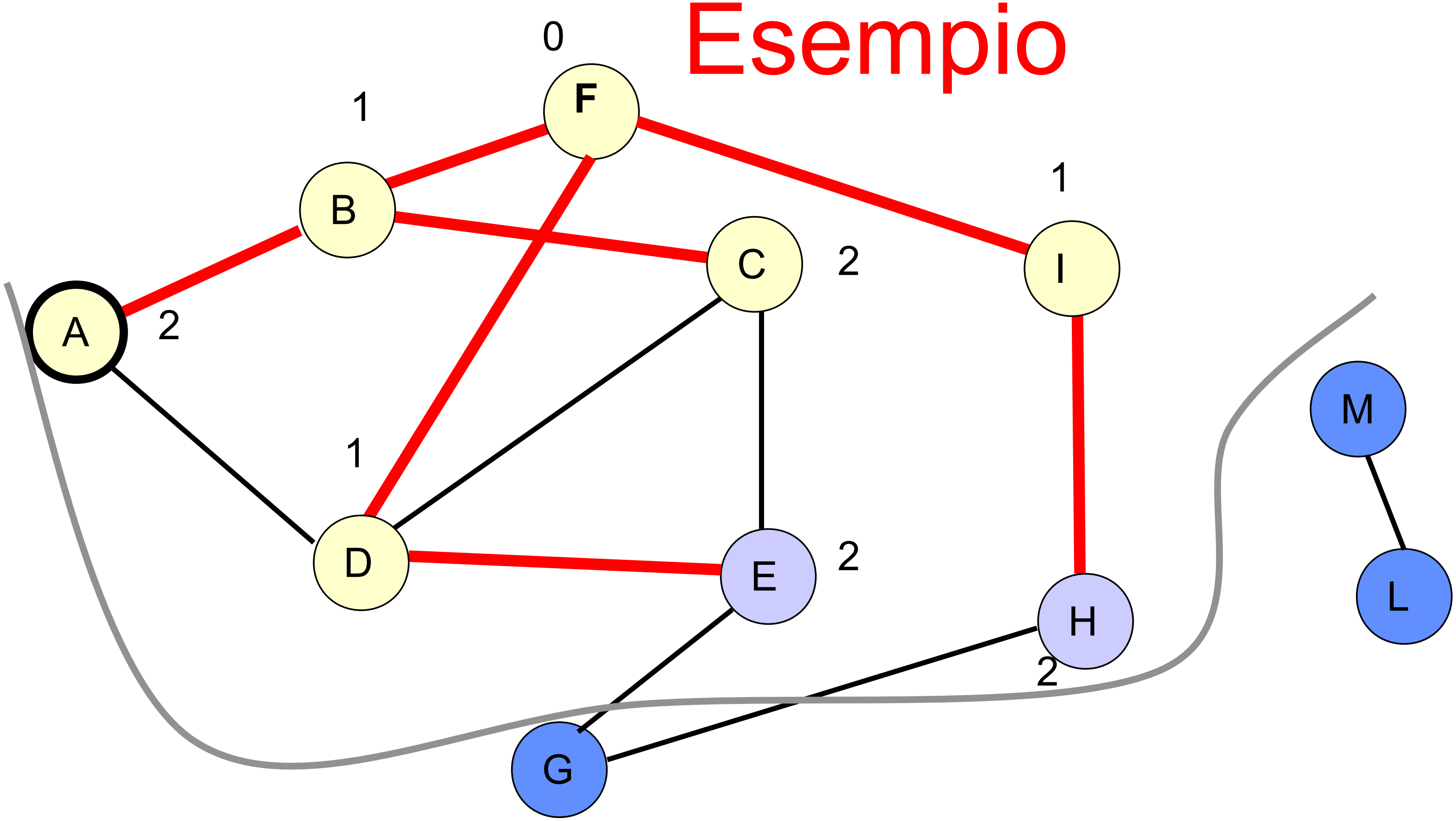
Coda:{C,A,E,H}

Esempio



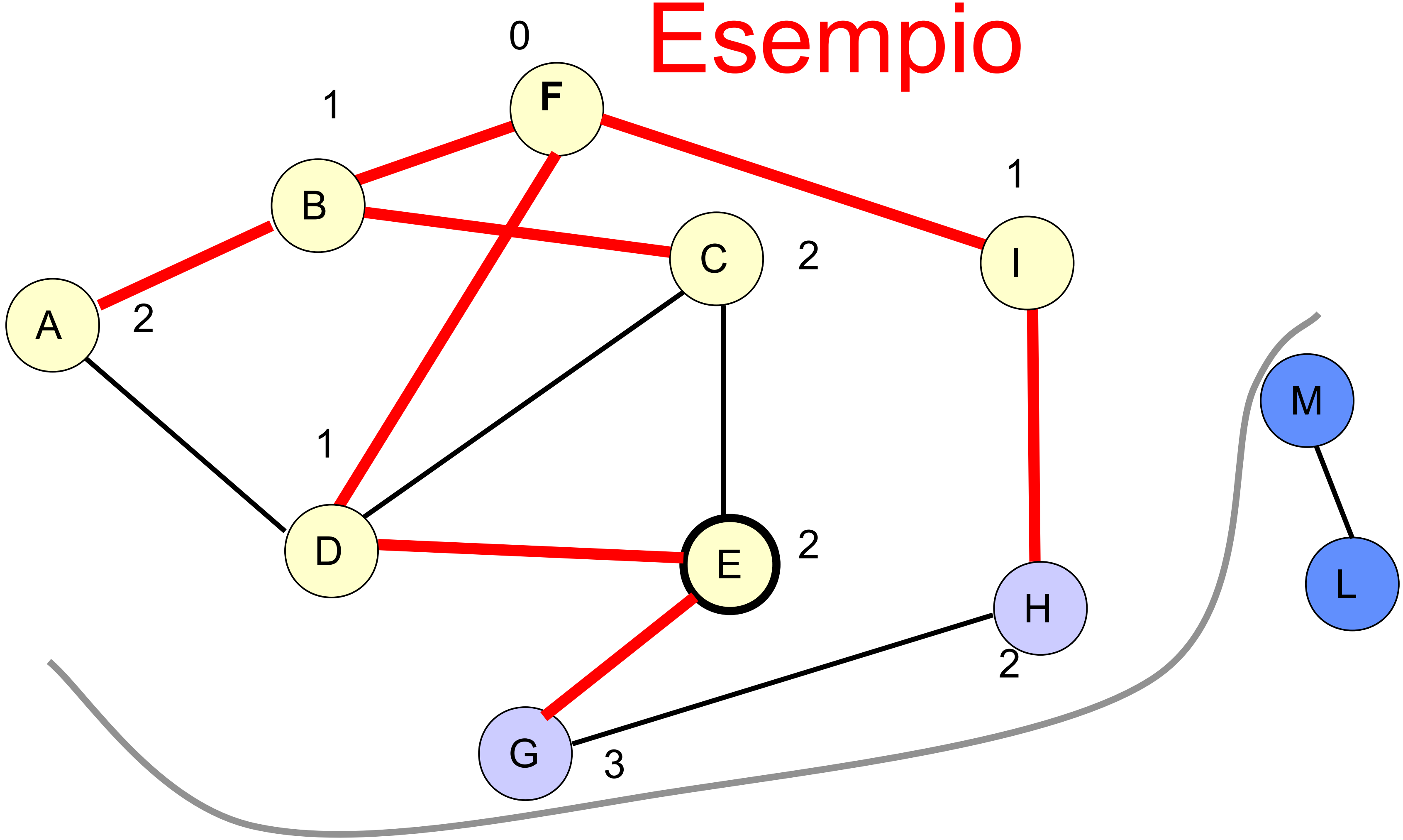
Coda:{A,E,H}

Esempio



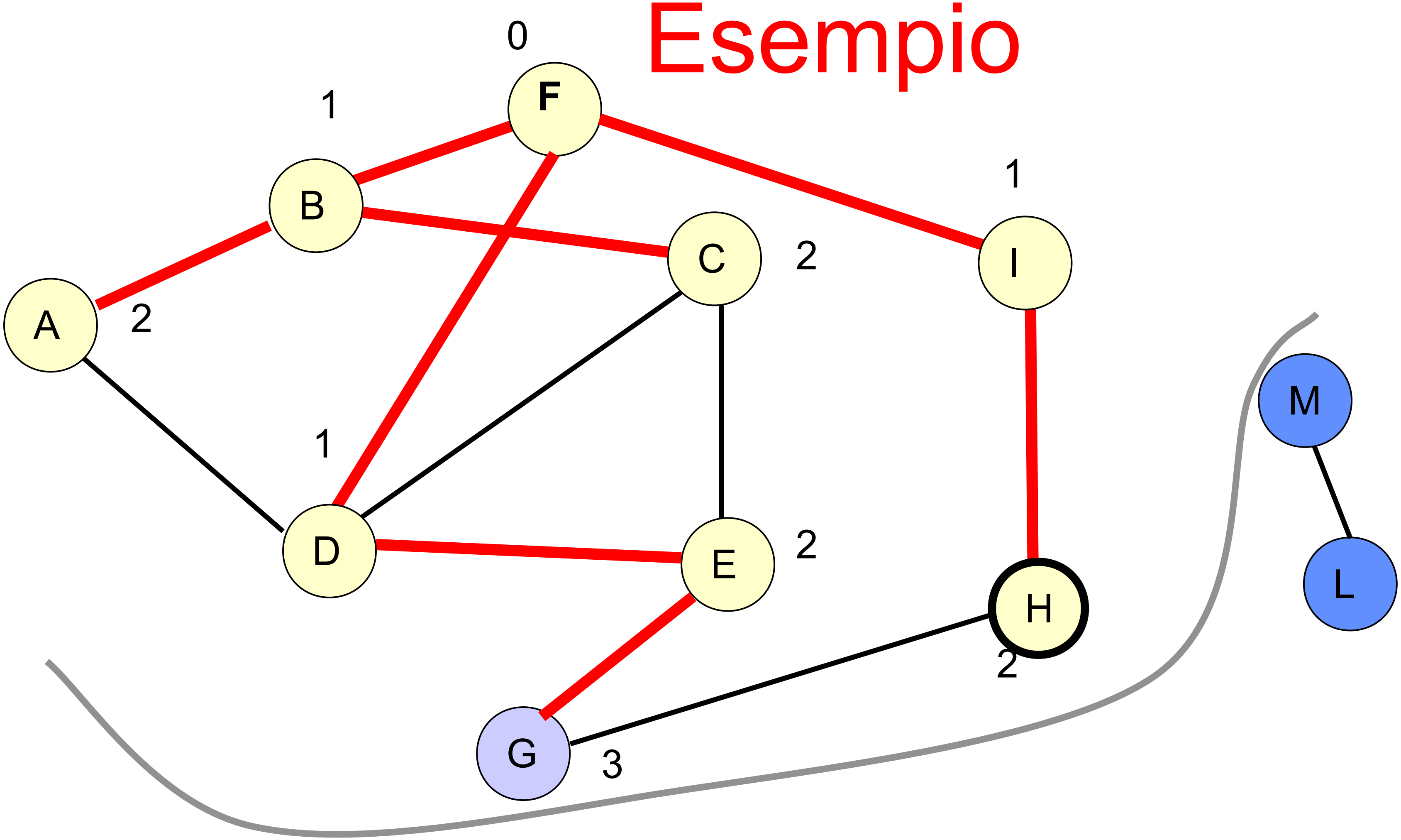
`Coda:{E,H}`

Esempio



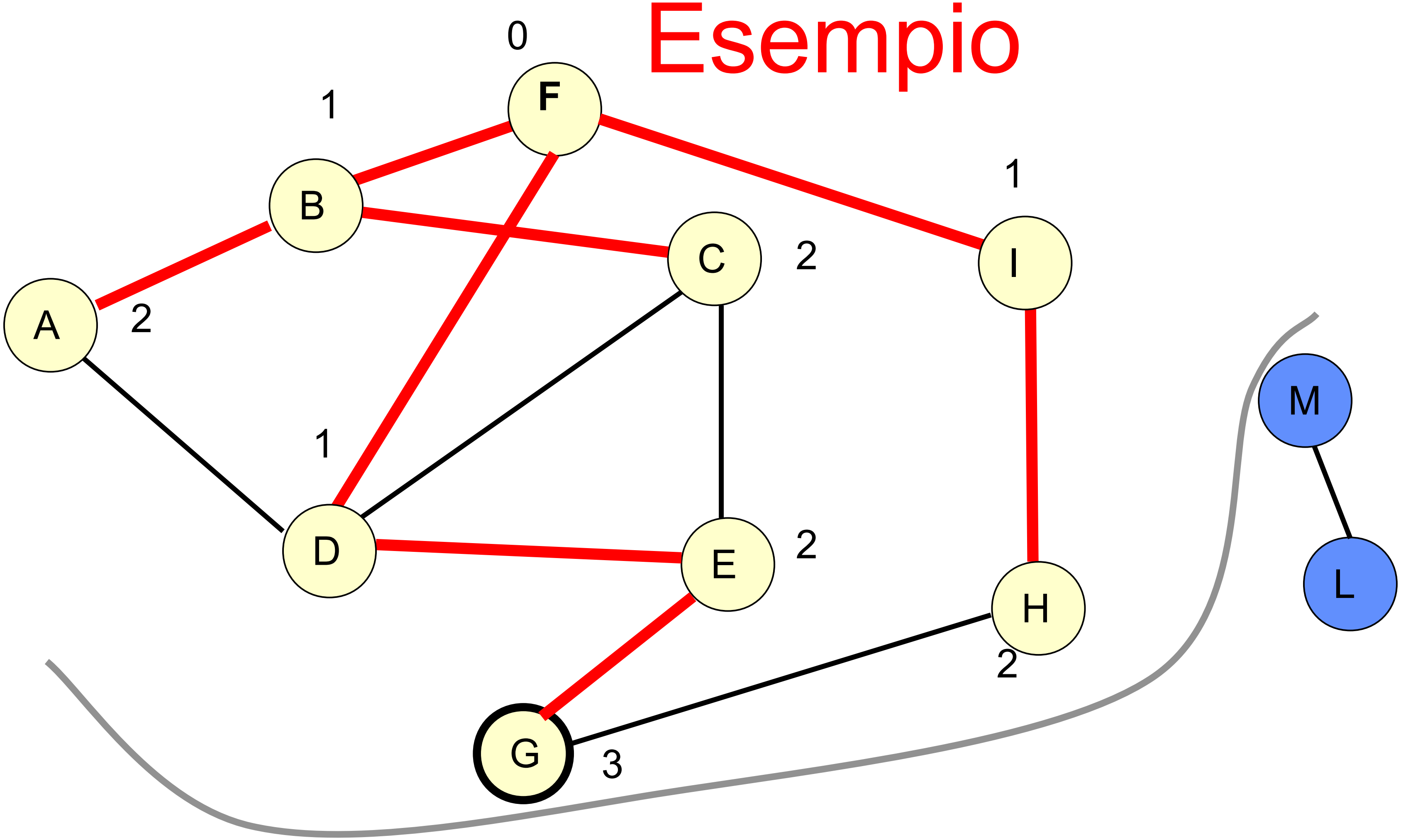
Coda:{H,G}

Esempio



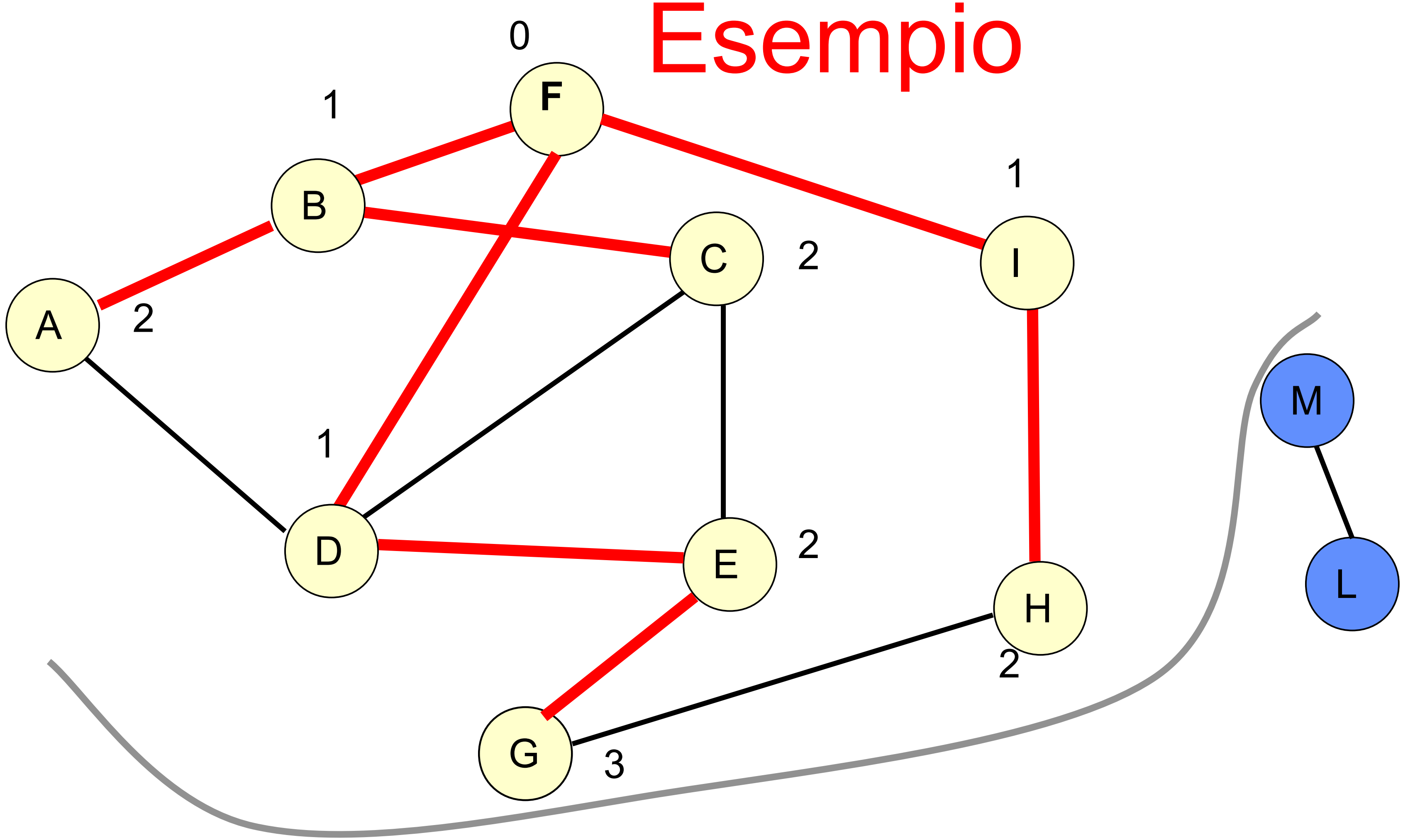
`Coda:{G}`

Esempio



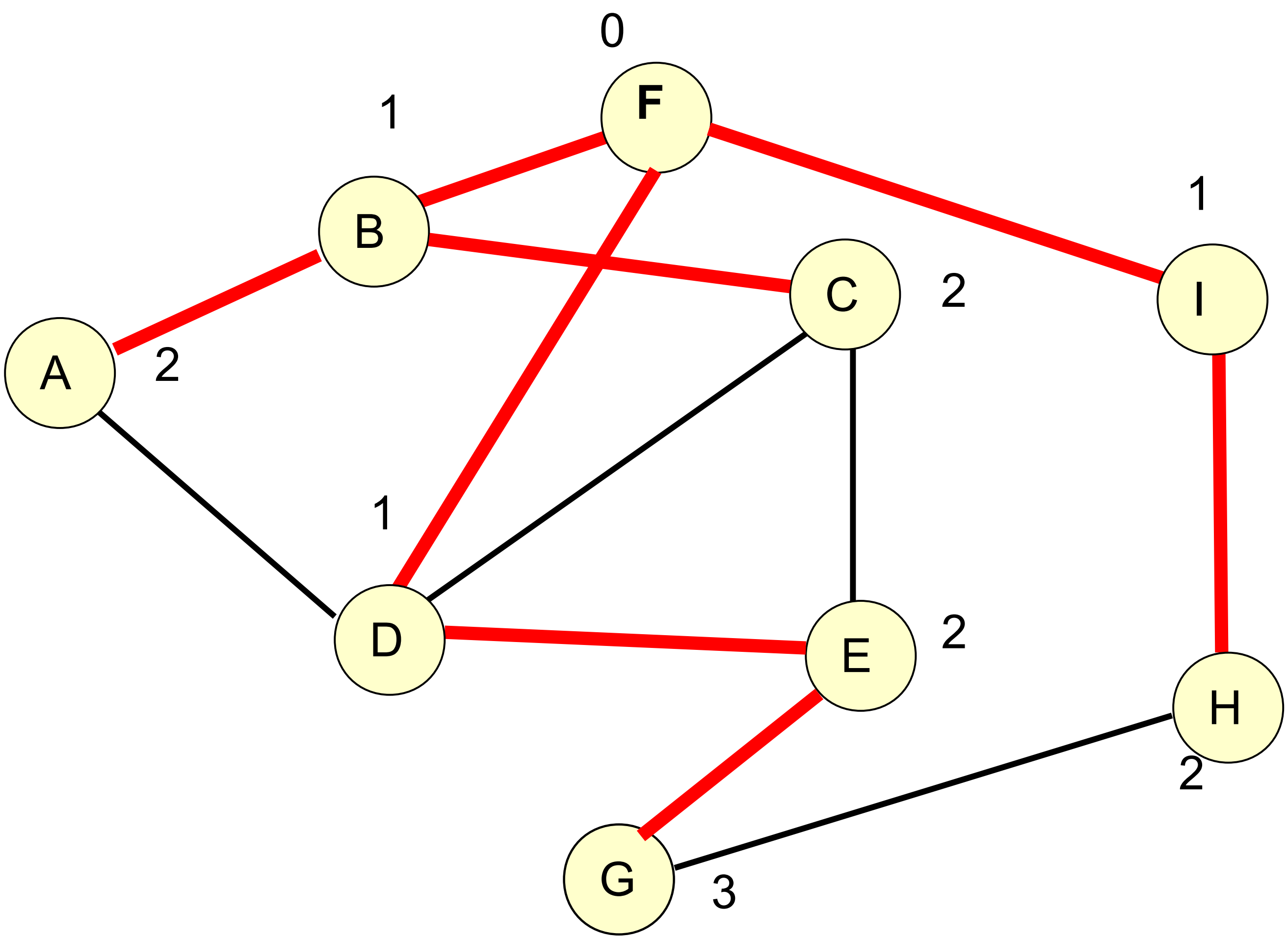
Coda:{}

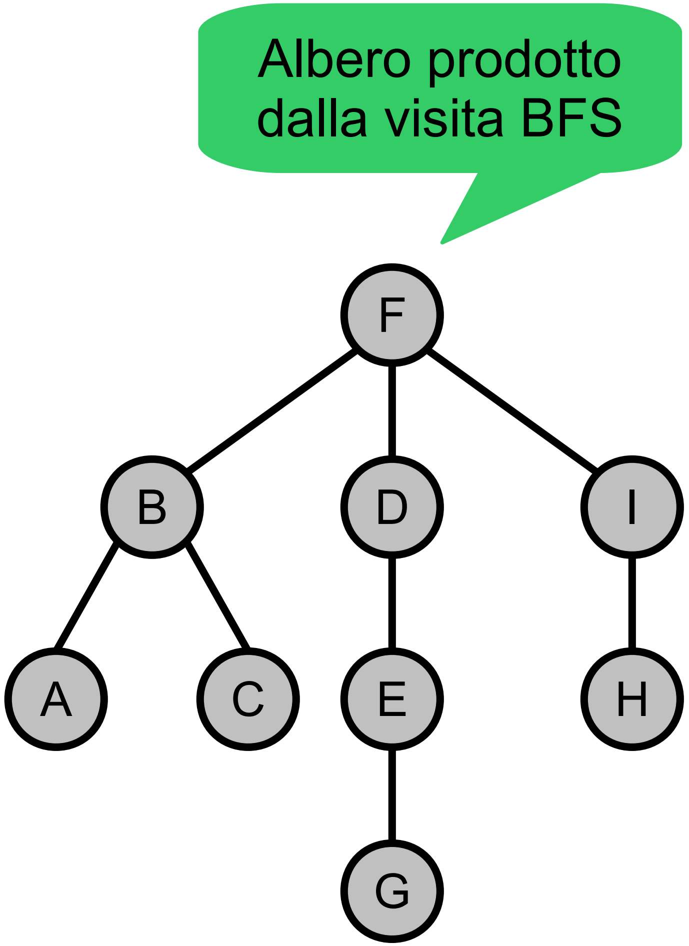
Esempio



Coda:{}

Esempio


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

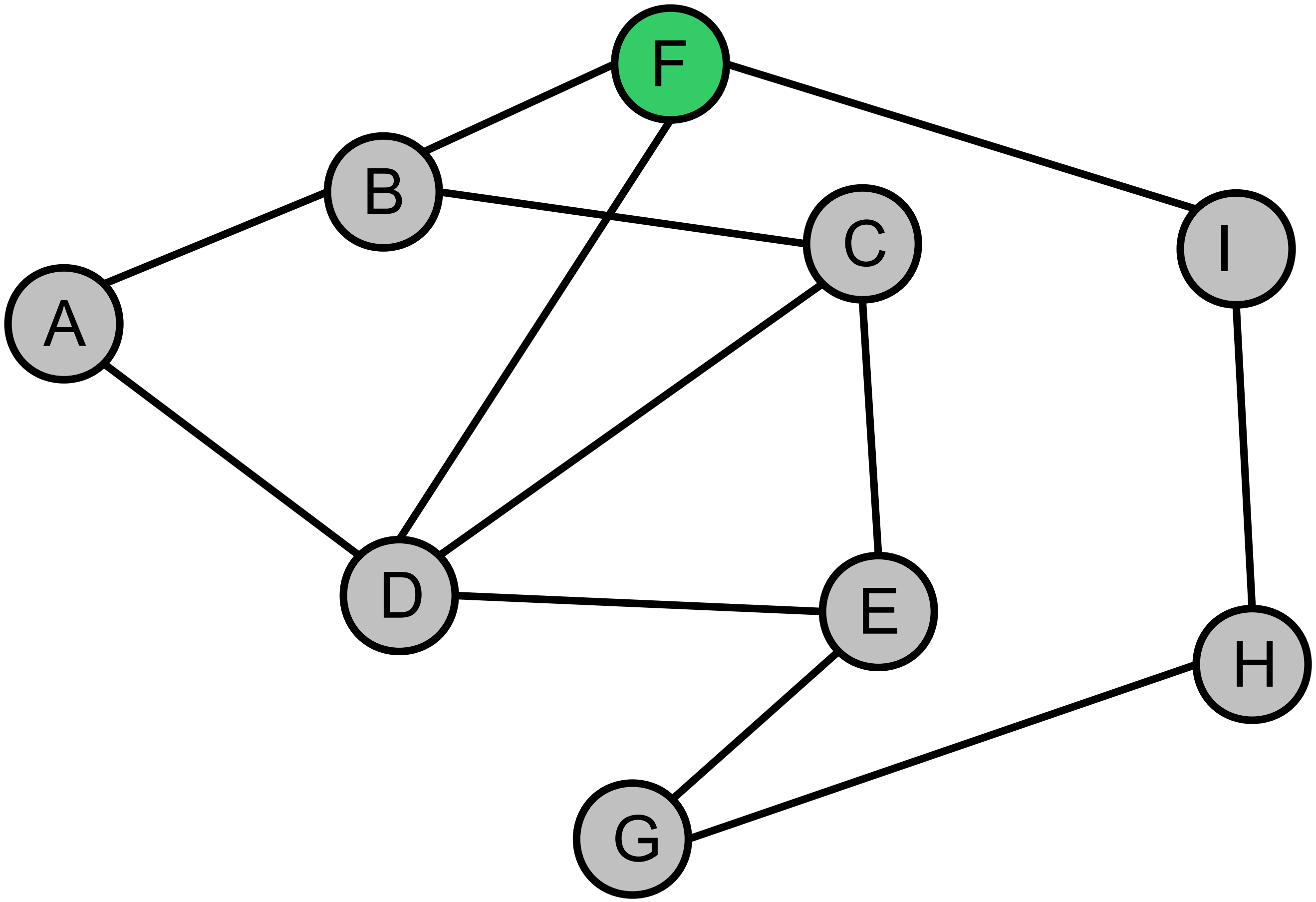

Albero prodotto dalla visita BFS
F
B
D
I
A
C
E
H
G

Applicazioni

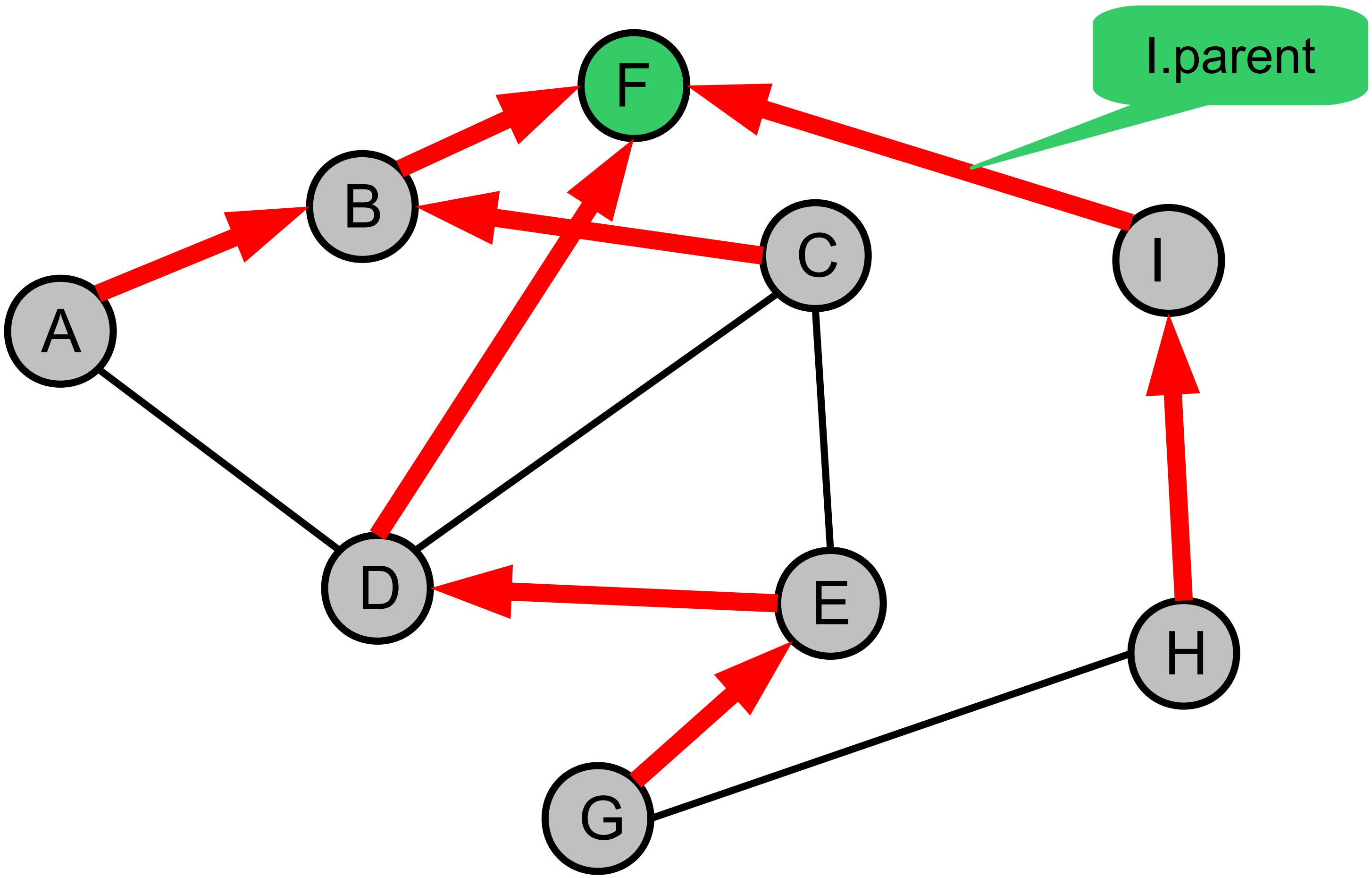
- La visita BFS può essere utilizzata per ottenere il percorso più breve (minor numero di archi attraversati) fra due vertici
- Ad esempio, il seguente pseudocodice stampa un cammino piú breve tra due nodi *s* e *v*
  - il grafo G è stato precedentemente visitato con l'algoritmo BFS a partire da *s* e l'albero della visita T è stato creato

```
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
```

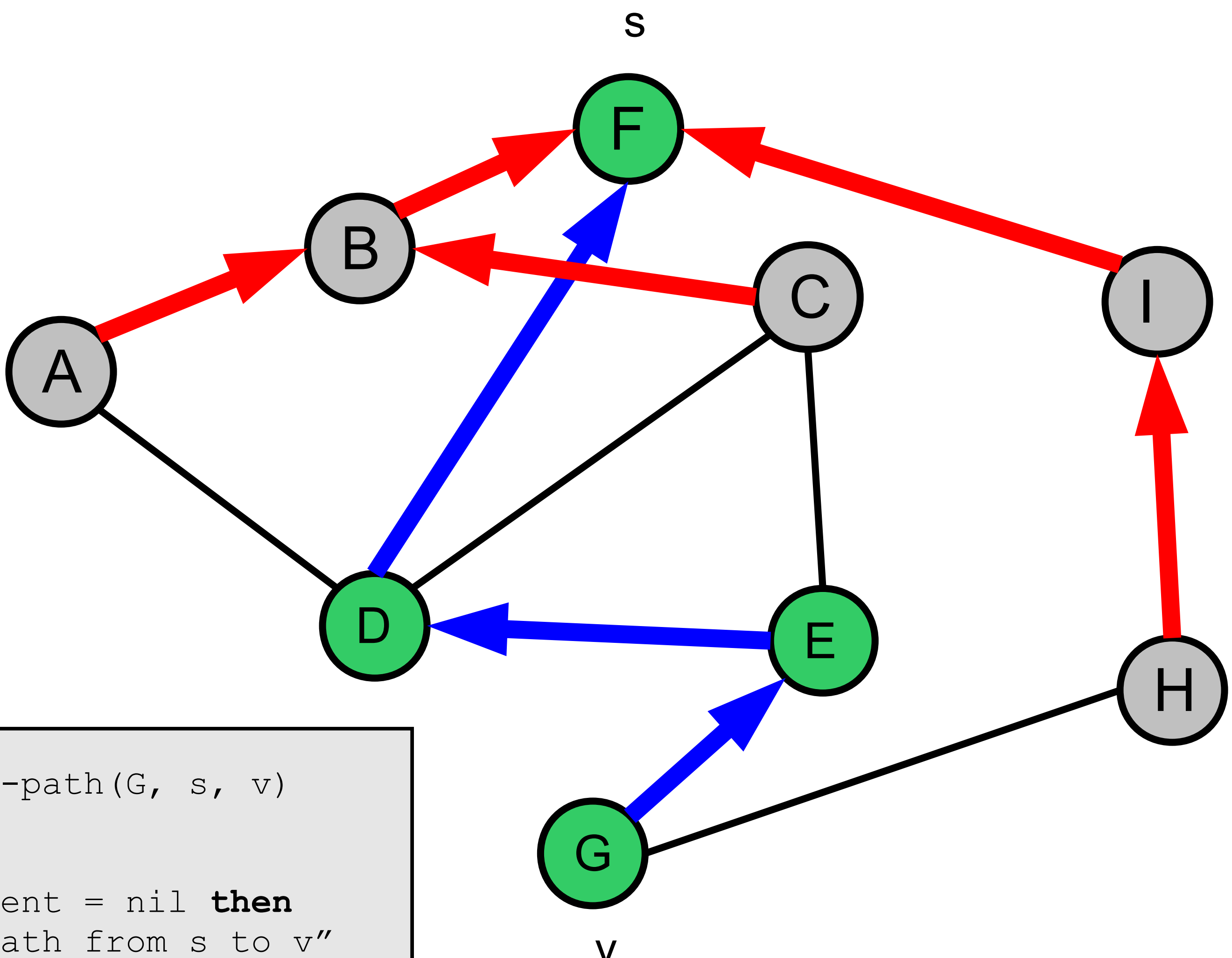
Esempio
cammino piú breve da F a G



Esempio
cammino piú breve da F a G

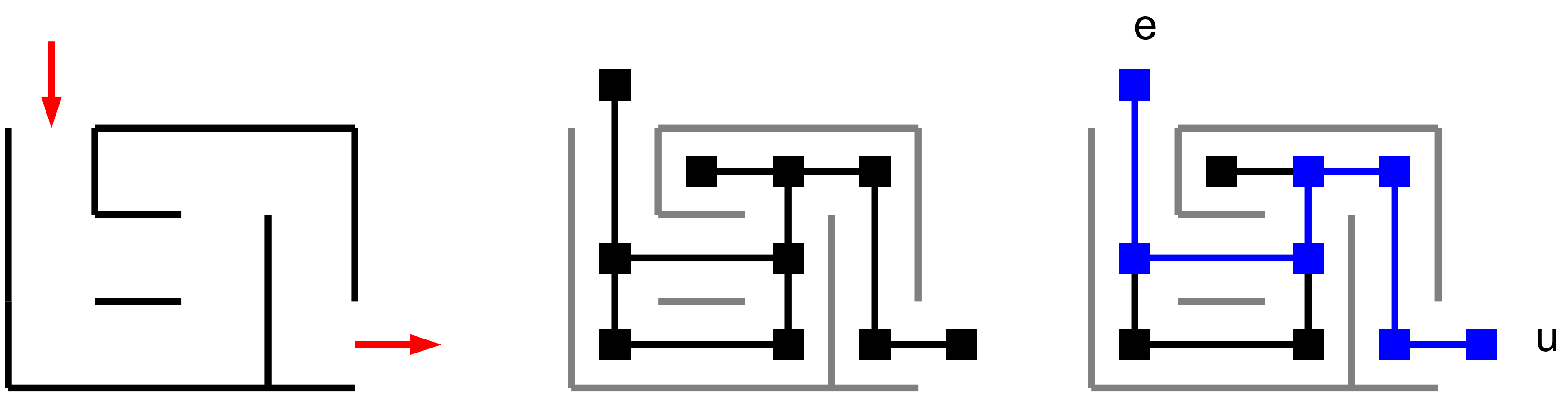


Esempio
print-path(G, s, v)



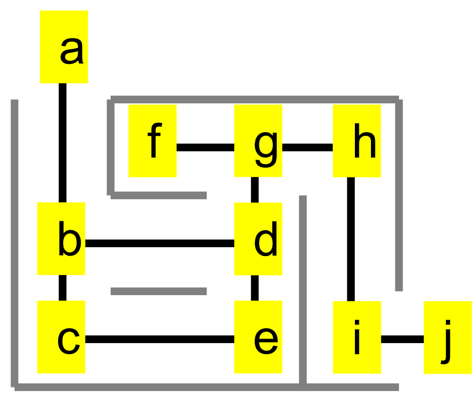

```
algoritmo print-path(G, s, v)
  if v = s then
    print s
  else if v.parent = nil then
    print "no path from s to v"
  else
    print-path(G,s,v.parent)
    print v
  endif
```

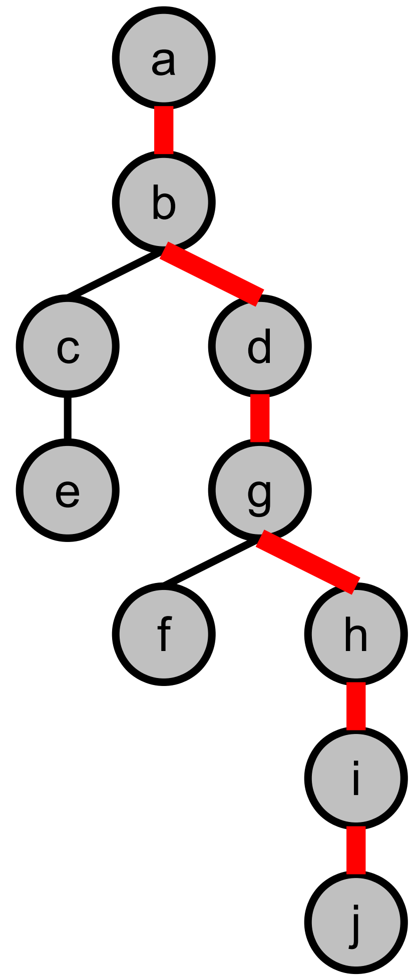
Esempio
Percorso con meno corridoi per uscire dal labirinto?



Esempio

Percorso con meno corridoi per uscire dal labirinto?





Visita in profondità (depth first search, DFS)

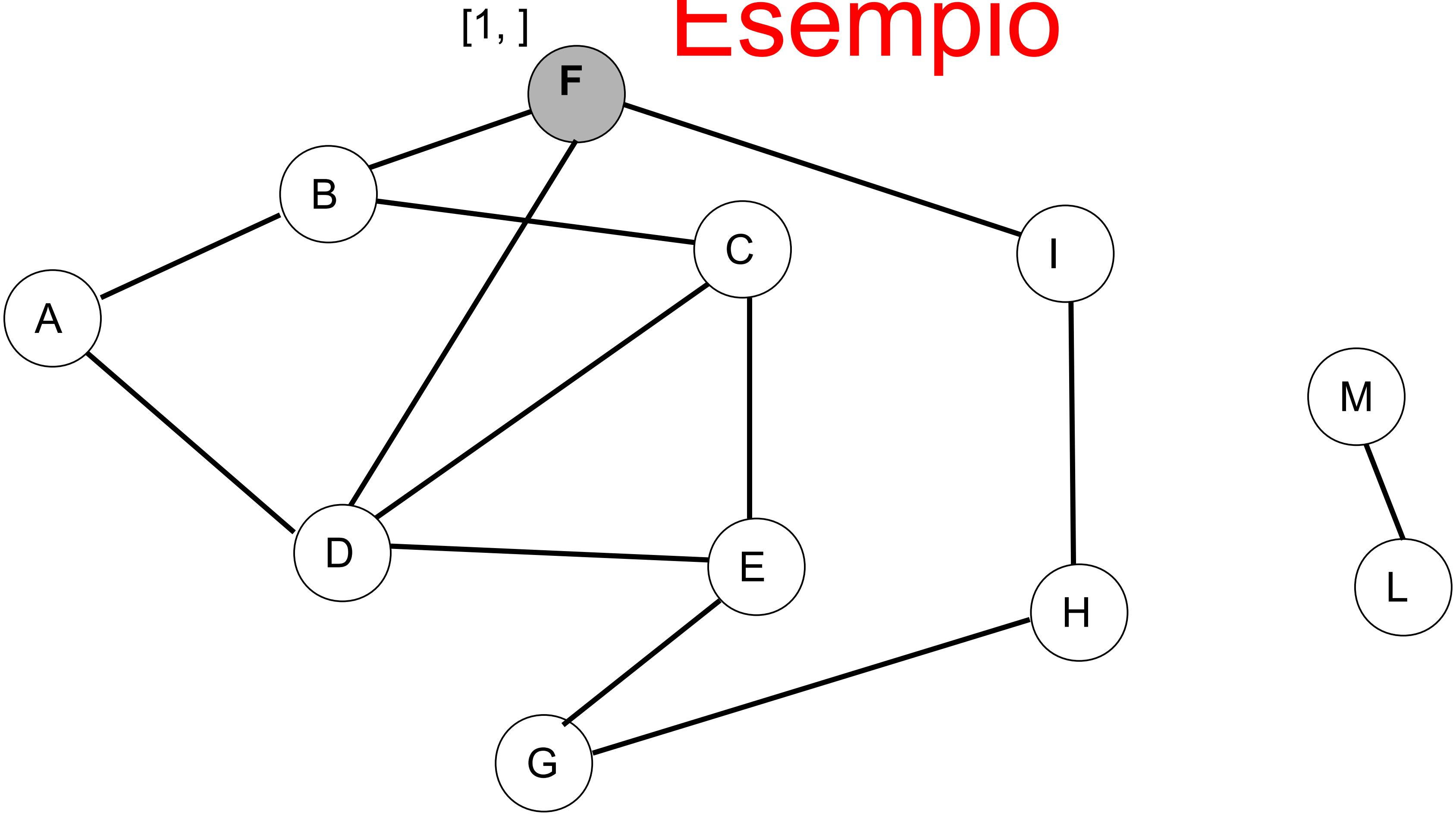
- Visita in profondità
  - Utilizzata per coprire l'intero grafo, non solo i nodi raggiungibili da una singola sorgente (diversamente da BFS)
- Output
  - Invece di un albero, una foresta DF (depth-first) $G_\pi=(V,E_\pi)$
    - Contenente un insieme di alberi DF
  - Informazioni addizionali sul tempo di visita
    - Tempo di scoperta di un nodo
    - Tempo di “terminazione” di un nodo

```
global time := 0; //var.globale

algoritmo DFS(Grafo G)
  for each u in V do
    u.mark := white;
    u.parent := nil;
  endfor
  for each u in V do
    if (u.mark == white) then
      DFS-visit(u);
    endif
  enddfor

algoritmo DFS-visit(vertice u)
  u.mark := gray;
  time := time+1;
  u.dt := time;
  for each v adiacente a u do
    if (v.mark = white) then
      v.parent := u;
      DFS-visit(v);
    endif
  endfor
  "visita il vertice u"
  time := time+1;
  u.ft := time;
  u.mark := black;
```

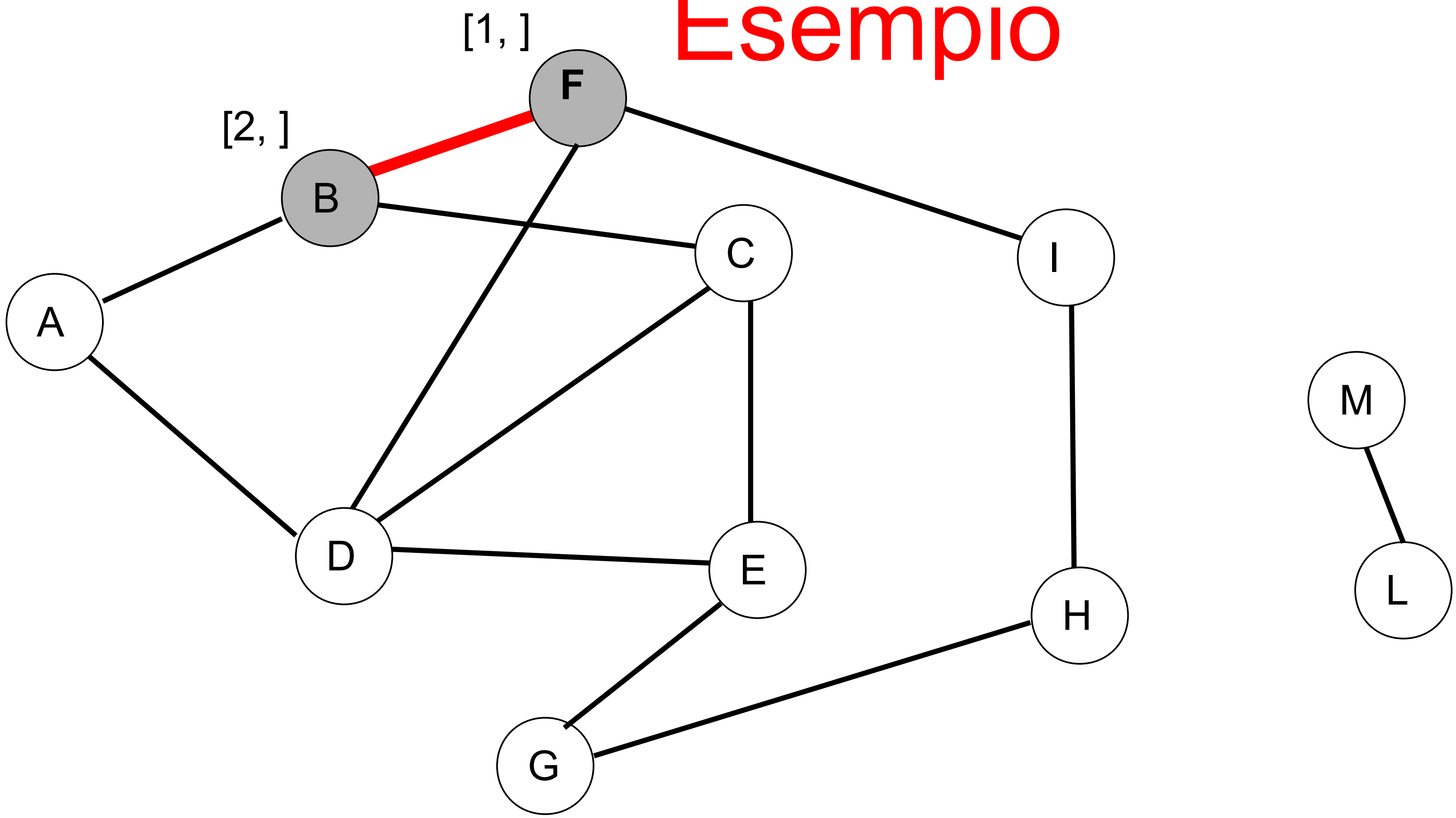
Visita in profondità
(depth first search, DFS)

- Versione ricorsiva
- *time* è una variabile globale che contiene il numero di "passi" dell'algoritmo
- *v.dt* (*discovery time*): tempo in cui il nodo è stato scoperto
- *v.ft* (*finish time*): tempo in cui la visita del nodo termina
- bianchi = inesplorati
- grigi = aperti
- neri = chiusi

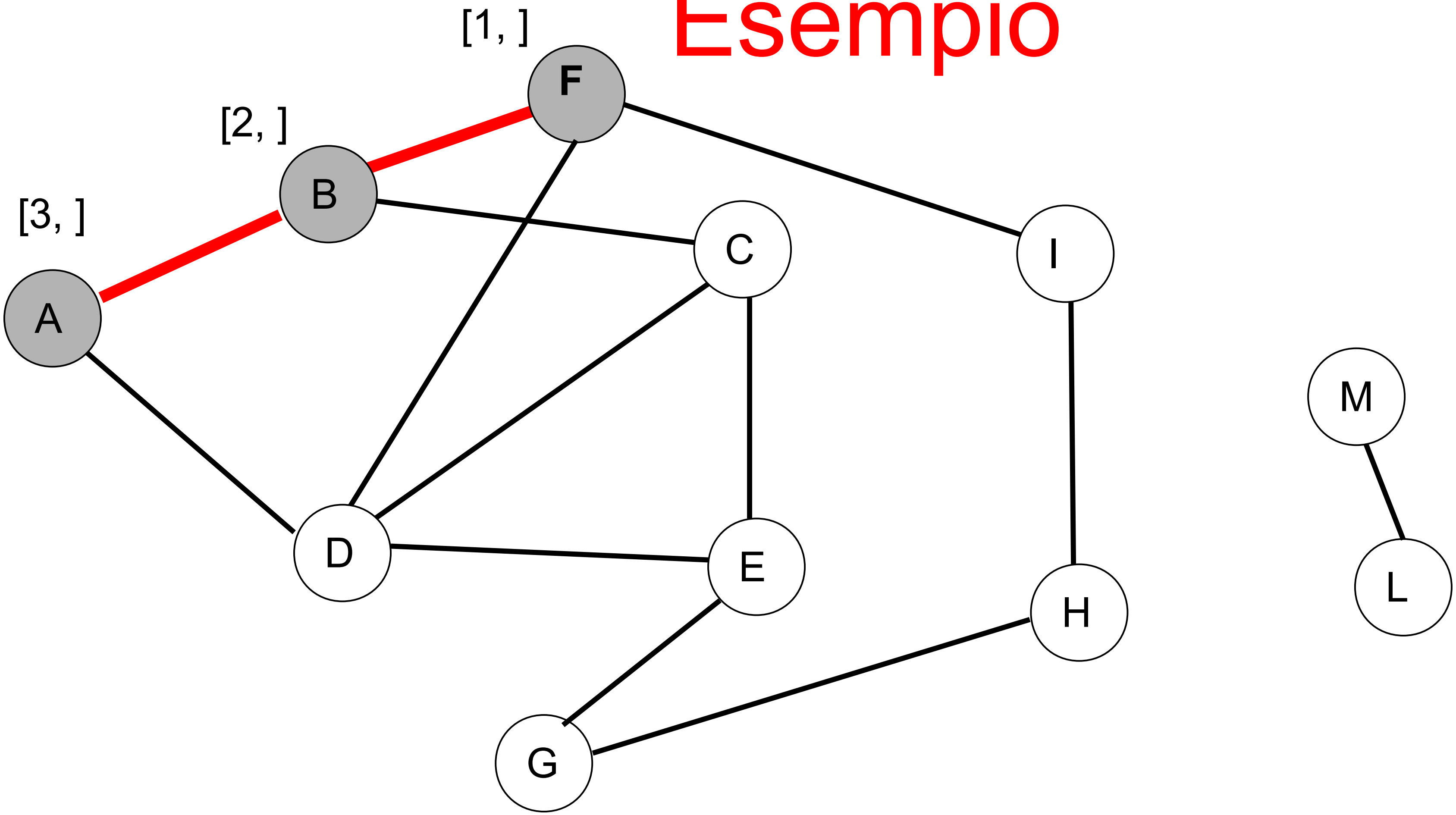
Esempio



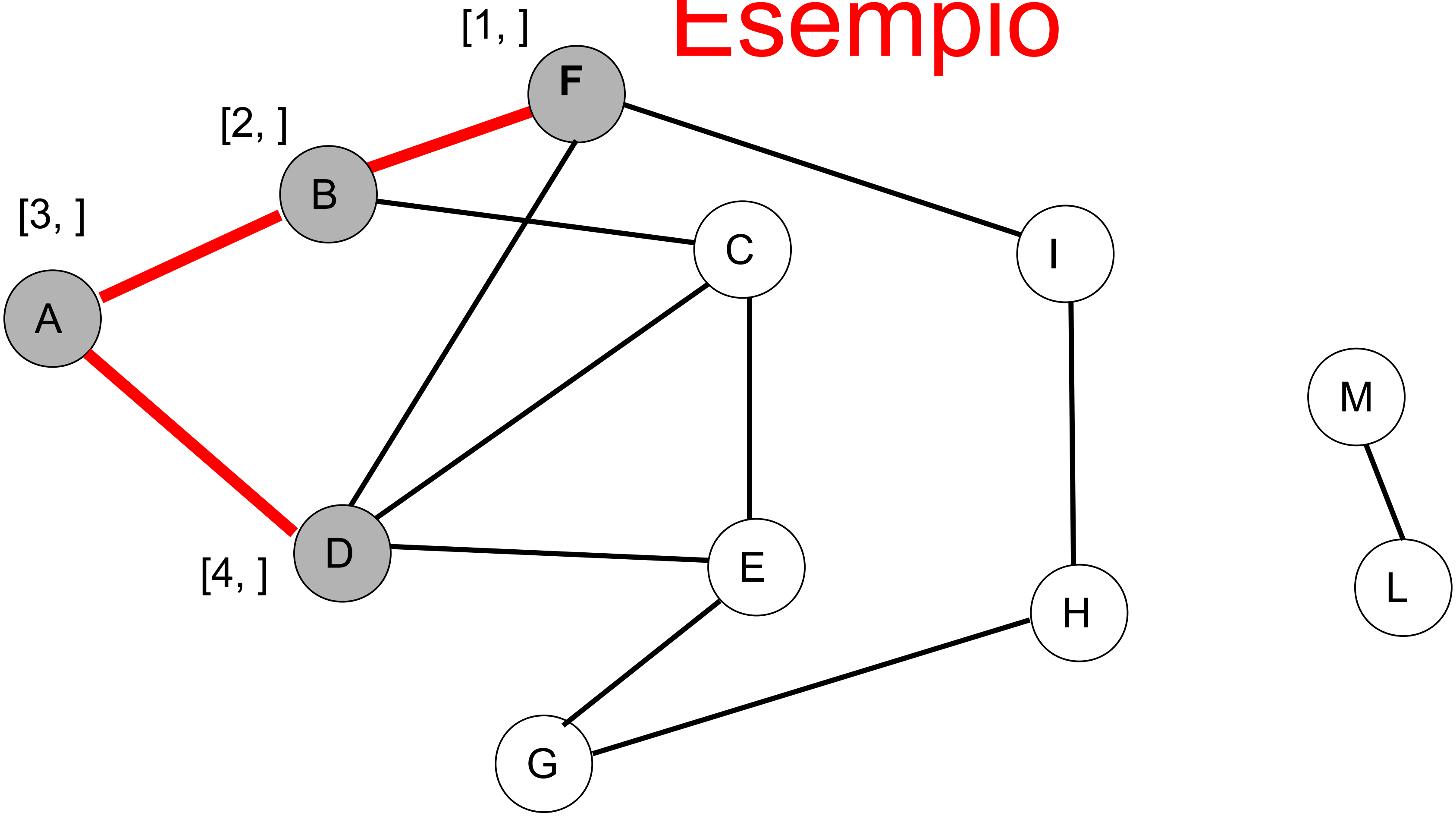
Esempio



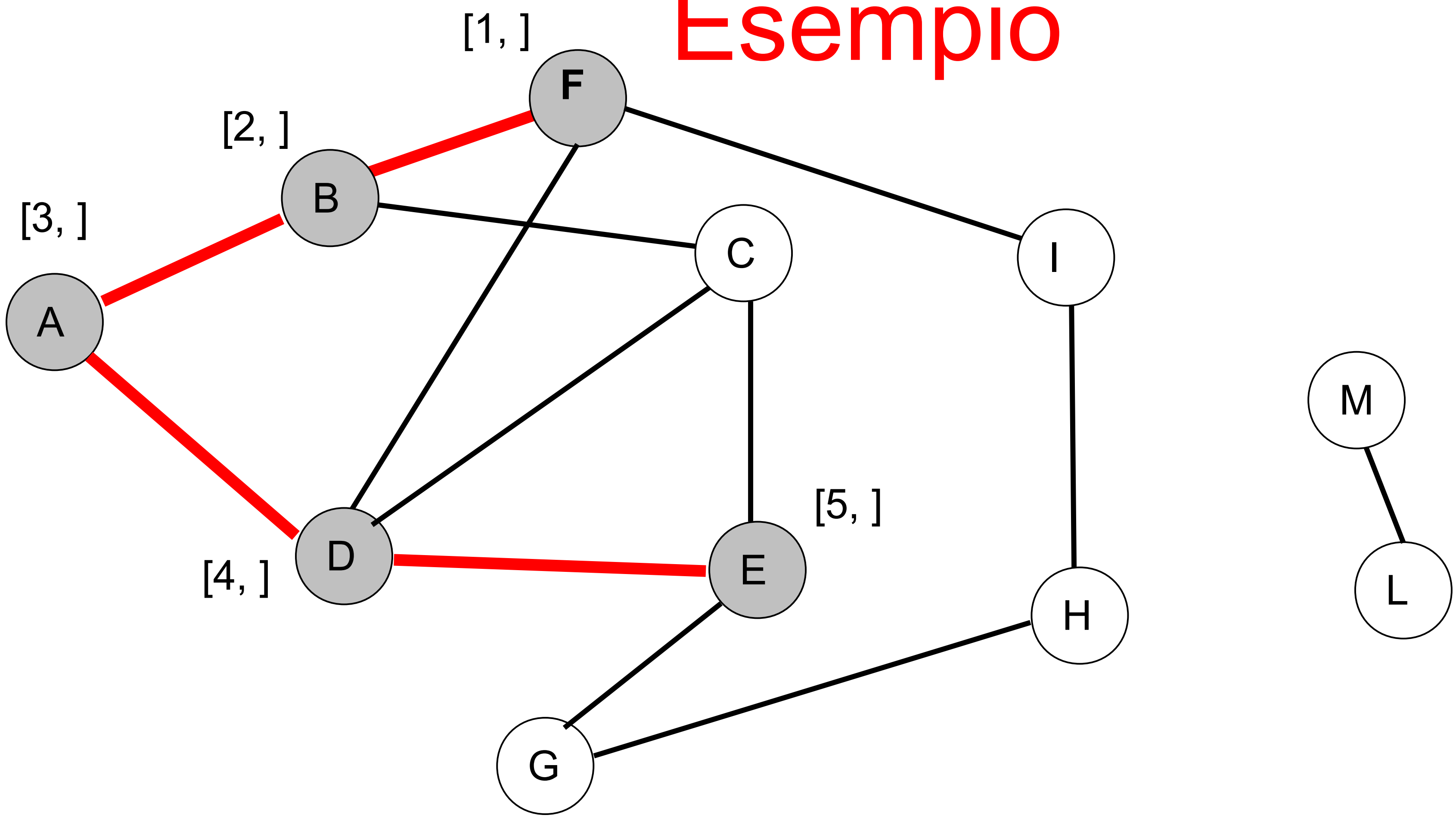
Esempio



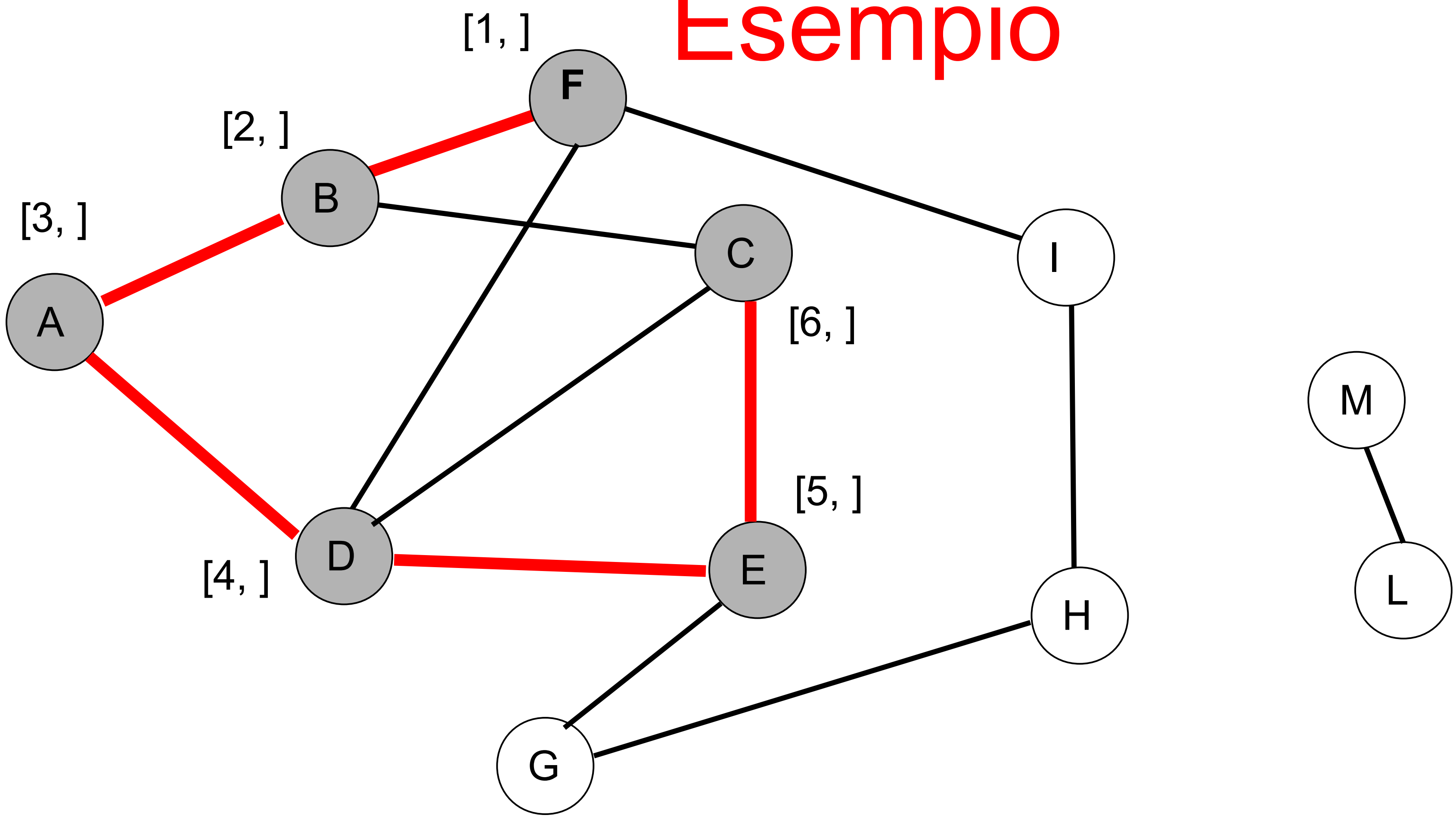
Esempio



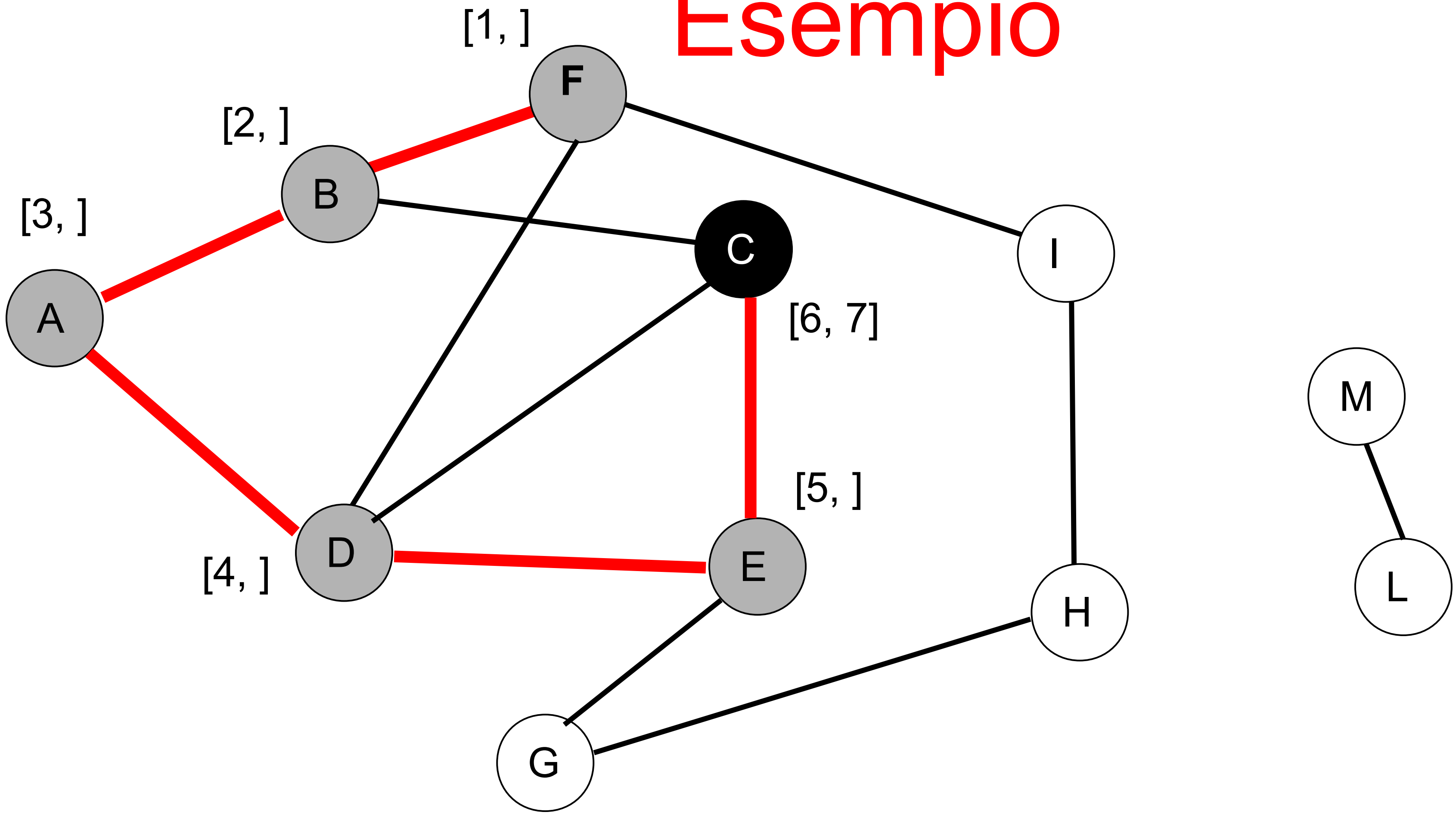
Esempio



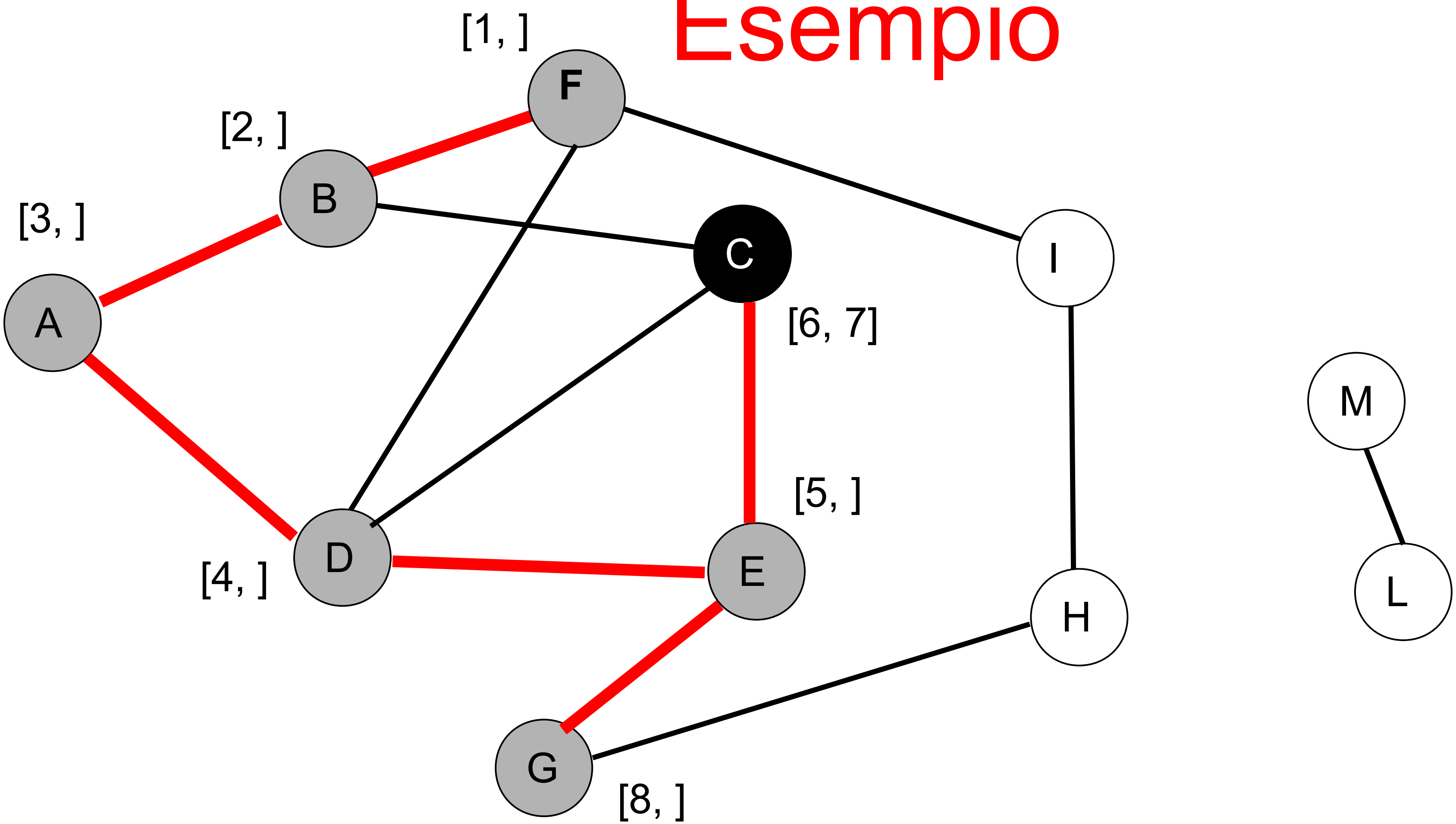
Esempio



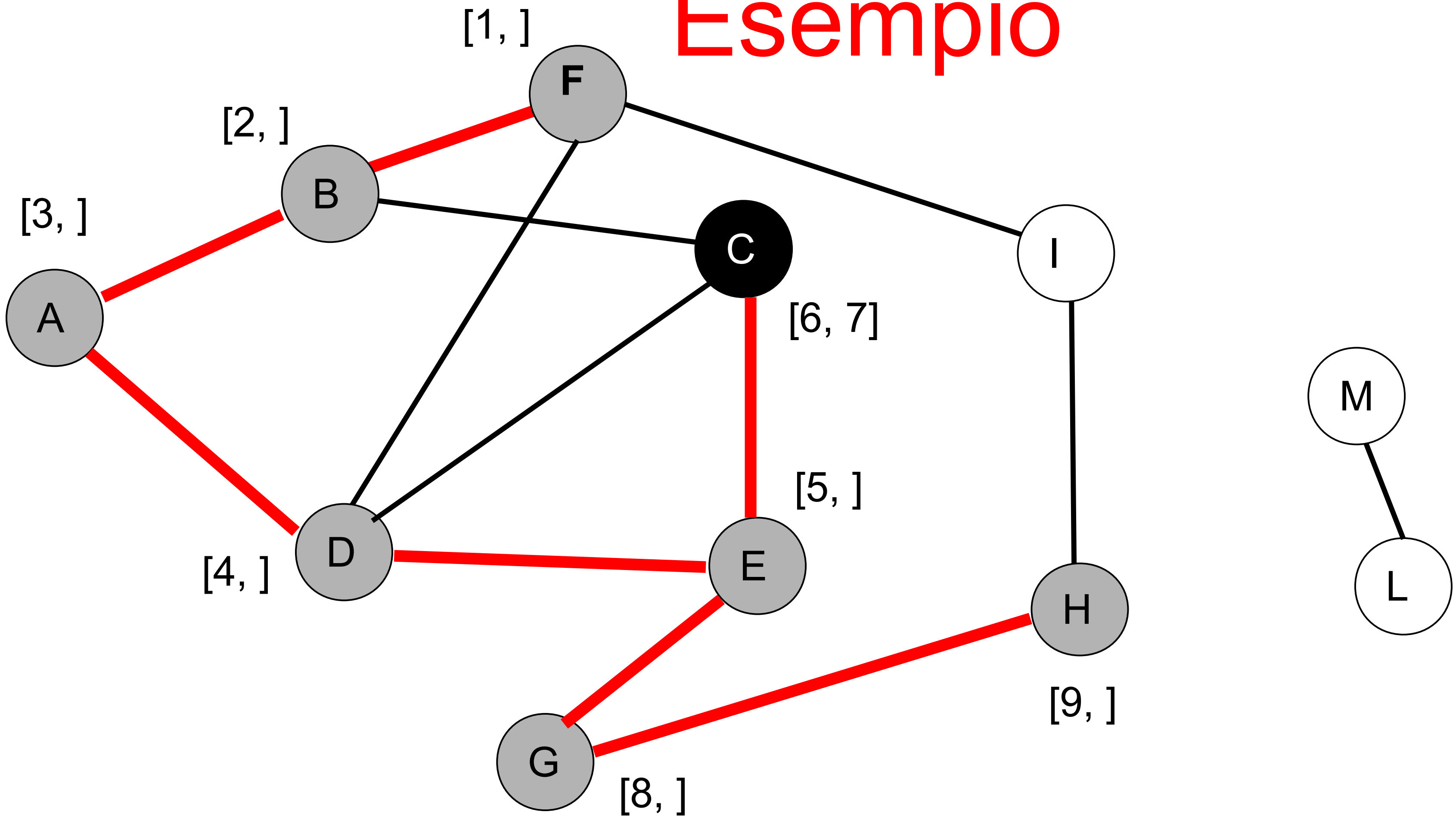
Esempio



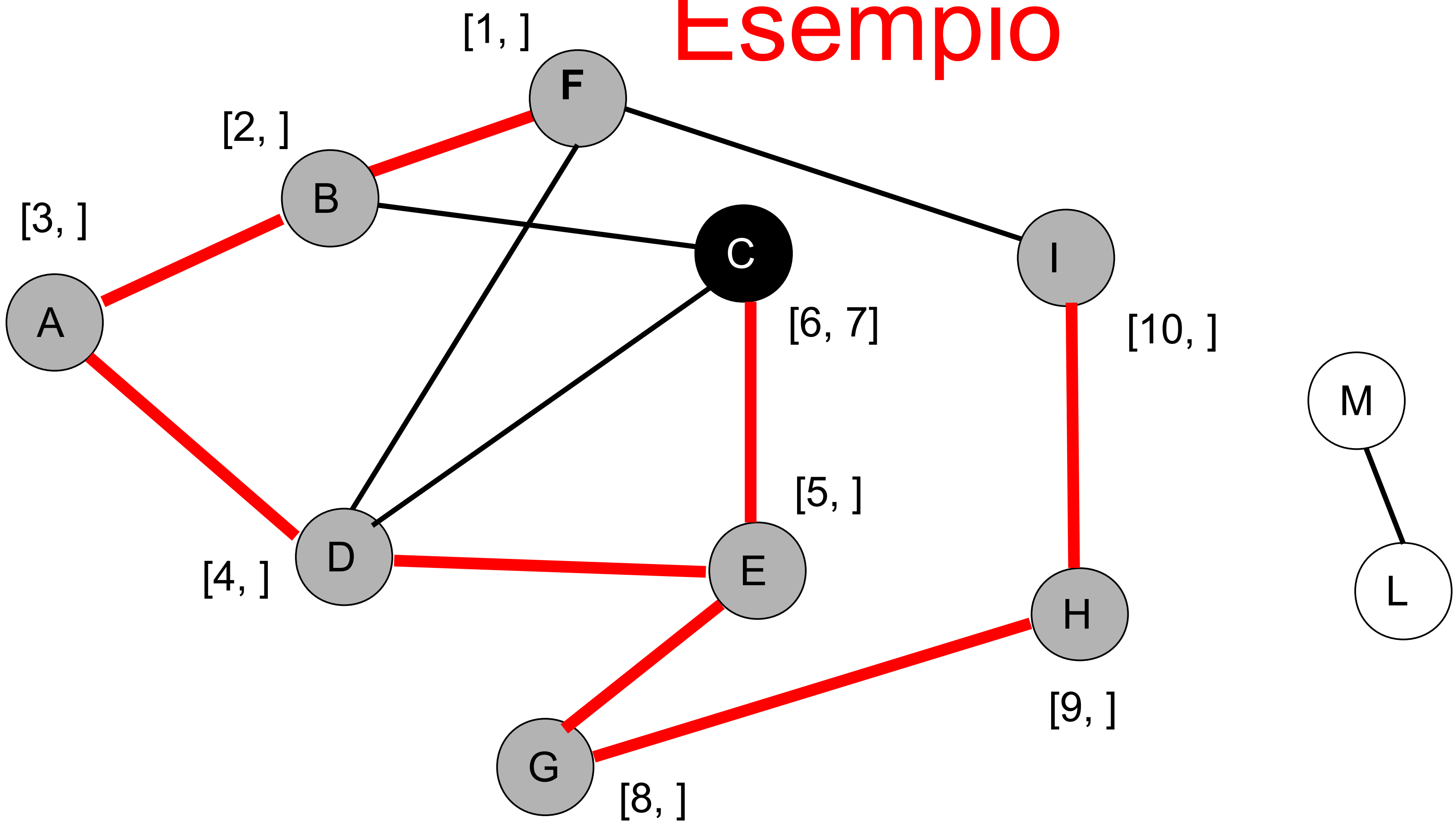
Esempio



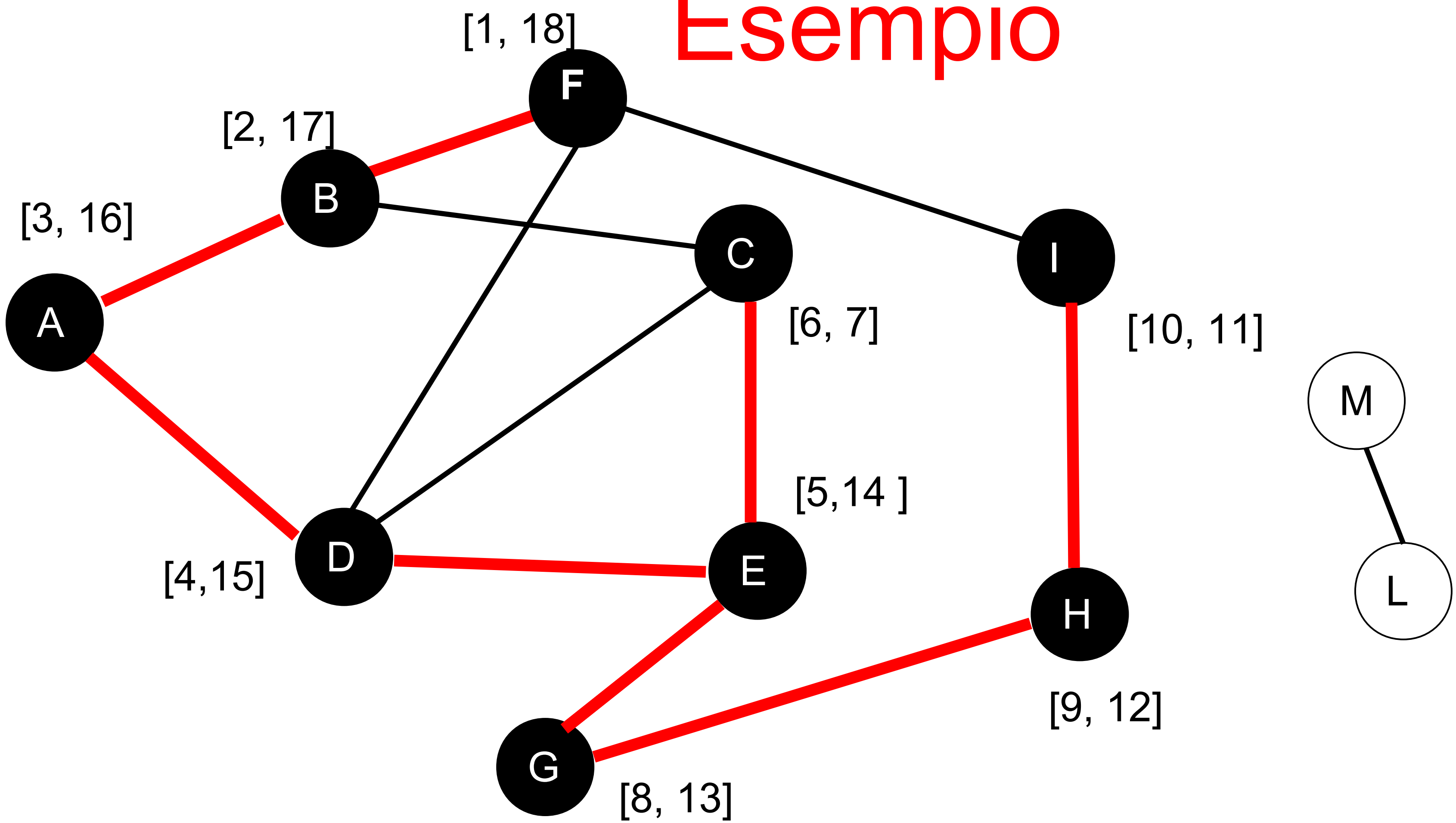
Esempio



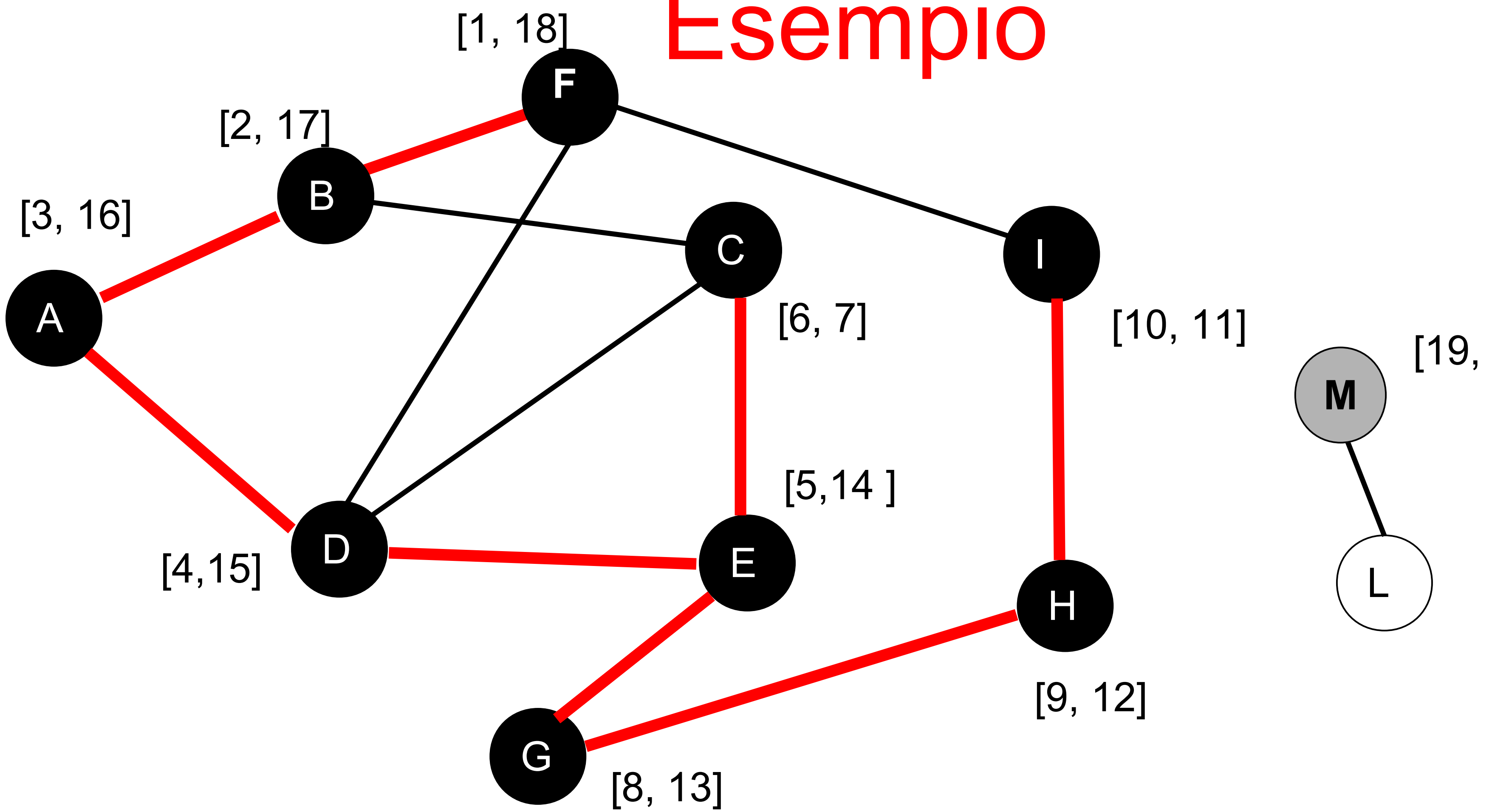
Esempio



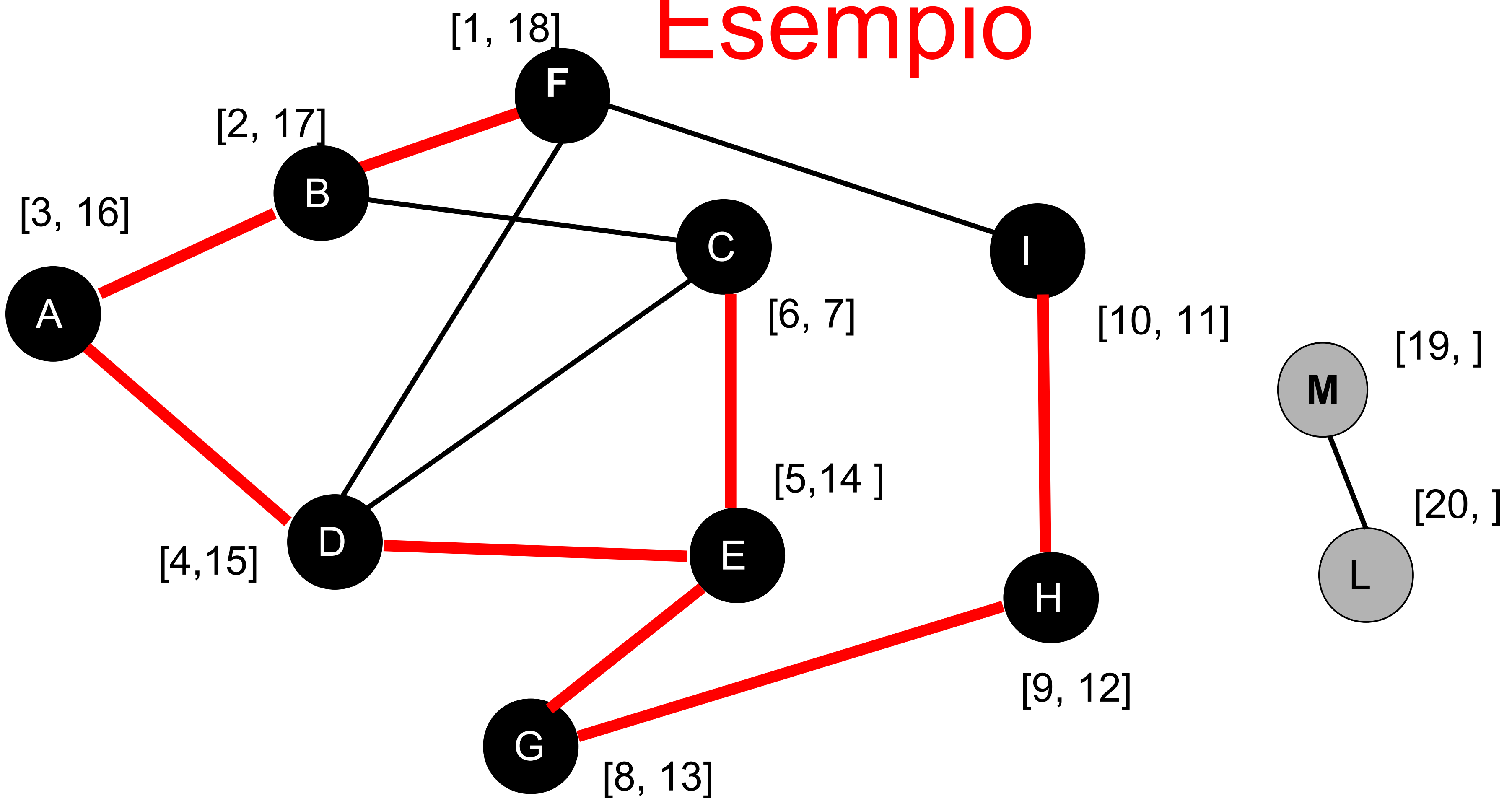
Esempio



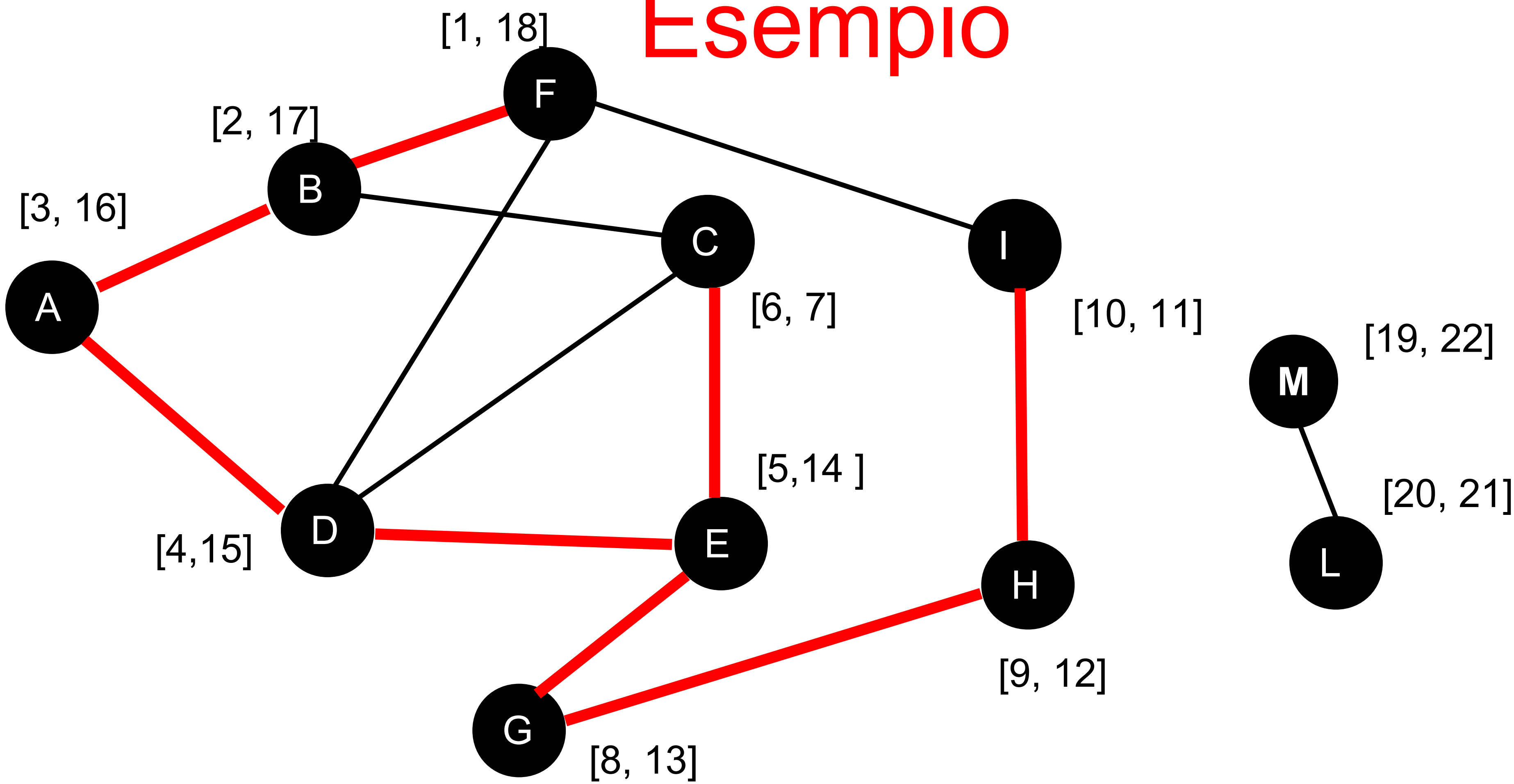
Esempio



Esempio



Esempio

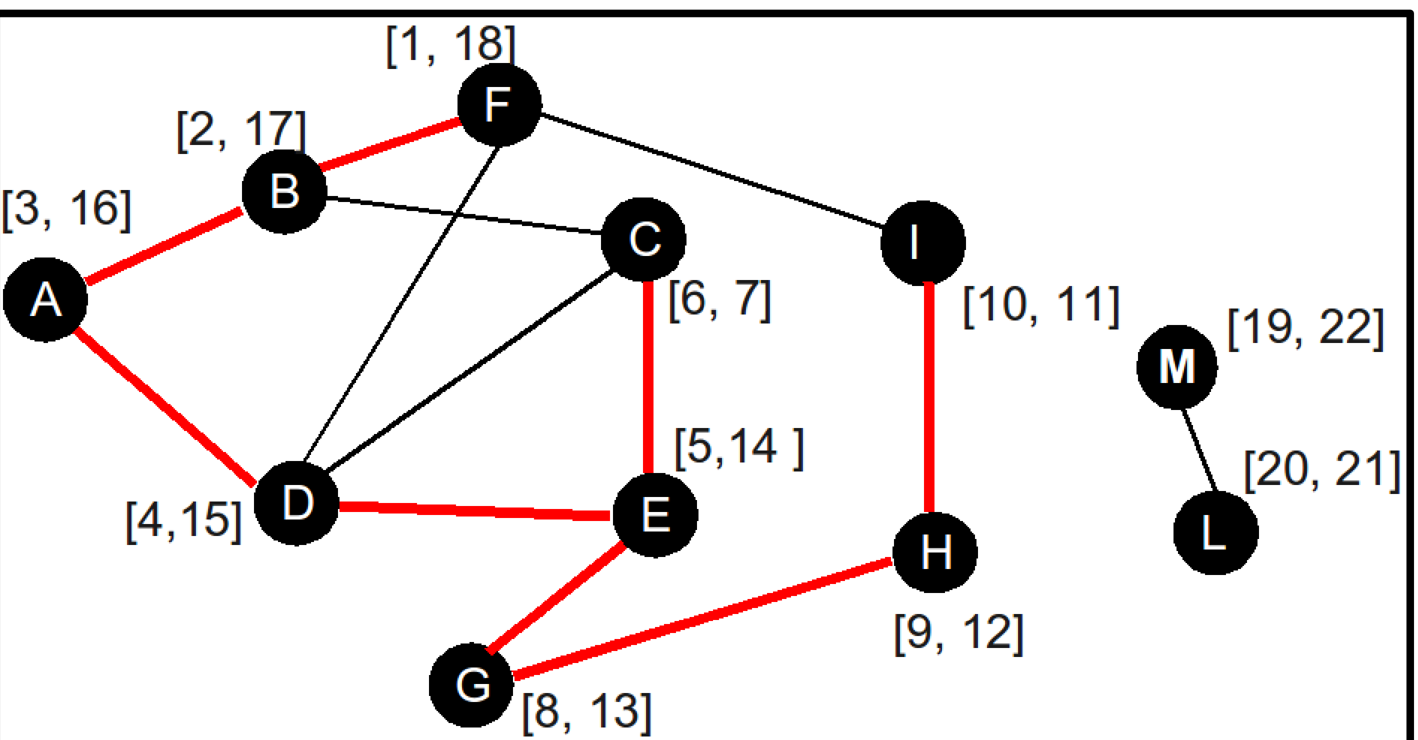


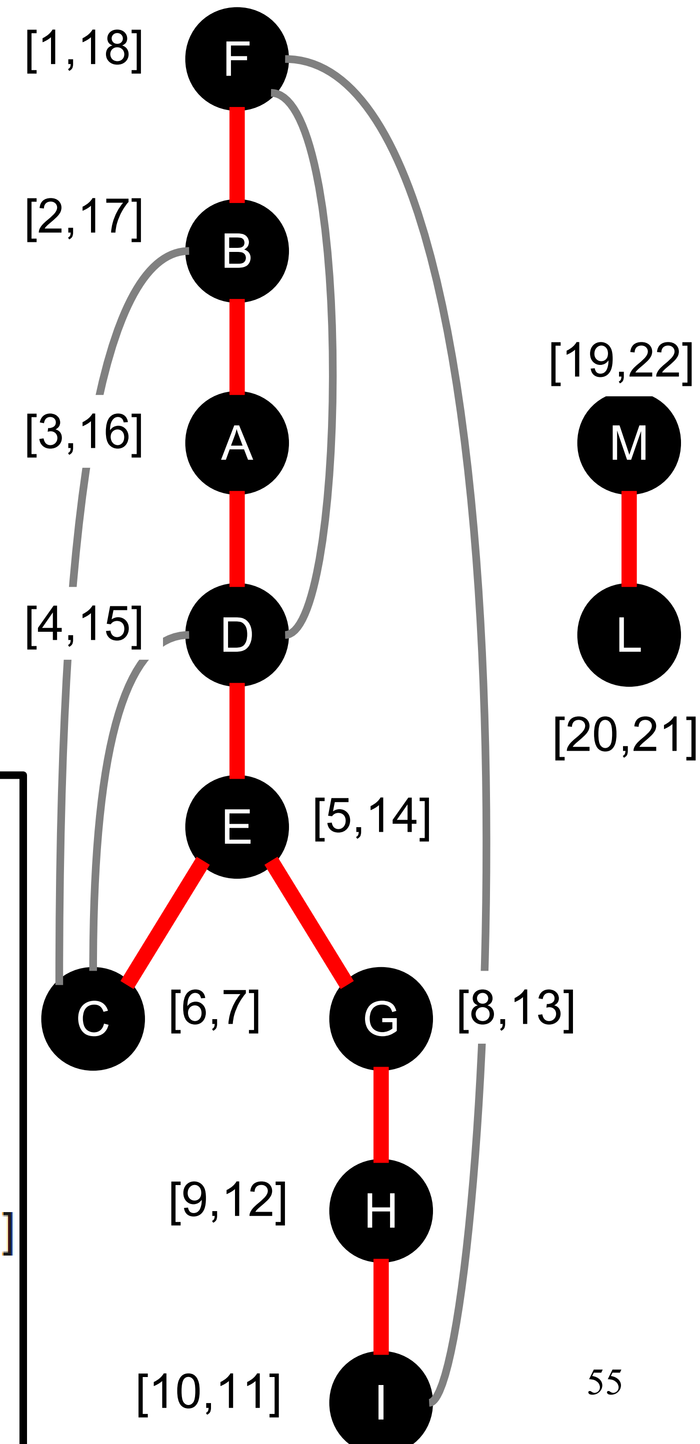
Proprietà della visita DFS Teorema delle parentesi

- In una qualsiasi visita di profondità di un grafo G=(V,E), per ogni coppia di vertici u,v, <u>una e una sola</u> delle seguenti condizioni è vera:
  - Gli intervalli *[u.dt, u.ft]* e *[v.dt, v.ft]* sono disgiunti
    $\Rightarrow$ *u,v* non sono discendenti l'uno dell'altro nella foresta DF
  - L'intervallo *[u.dt, u.ft]* è interamente contenuto in *[v.dt, v.ft]*
    $\Rightarrow$ *u* è discendente di *v* in un albero DF
  - L'intervallo *[v.dt, v.ft]* è interamente contenuto in *[u.dt, u.ft]*
    $\Rightarrow$ *v* è discendente di *u* in un albero DF
- Corollario
  - Il vertice *v* è un discendente del vertice *u* nella foresta DF per un grafo *G* se e soltanto se: $u.dt < v.dt < v.ft < u.ft$

Foresta DF

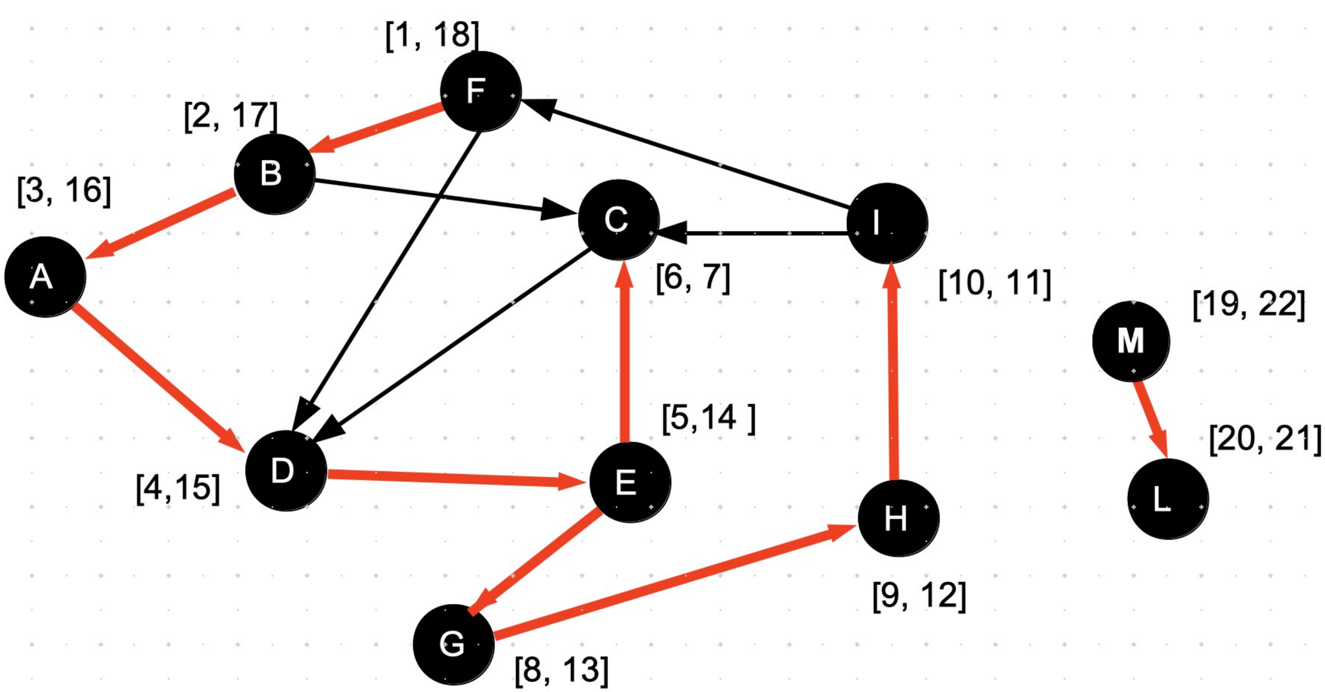
- [u.dt, u.ft] e [v.dt, v.ft] sono disgiunti
  ⇒ *u,v* non sono discendenti nella foresta DF
- [u.dt, u.ft] è interamente contenuto in [v.dt, v.ft]
  ⇒ *u* è discendente di *v* in un albero DF
- [v.dt, v.ft] è interamente contenuto in [u.dt, u.ft]
  ⇒ *v* è discendente di *u* in un albero DF

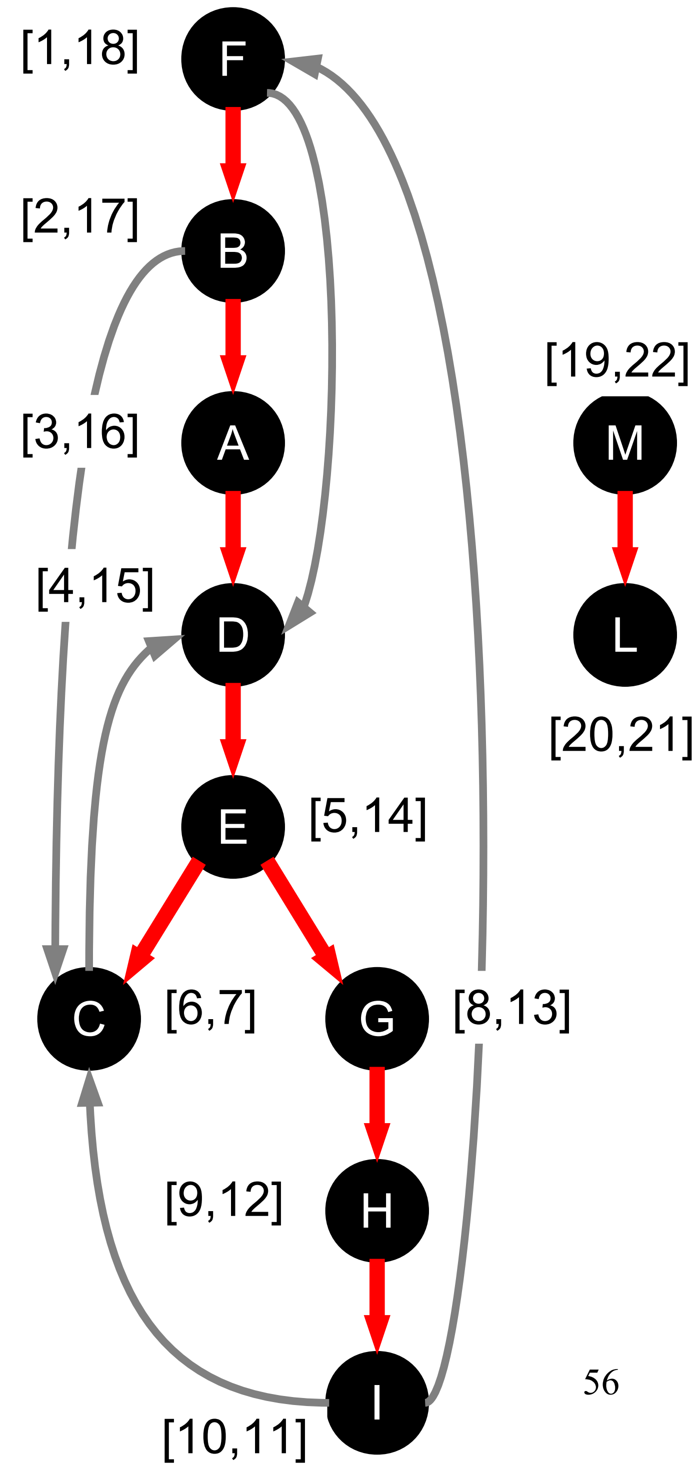




Grafi orientati

- Consideriamo un grafo orientato e un arco (u,v) non incluso nella foresta DF
- Se *v.dt < u.dt* e *u.ft < v.ft* l'arco (u,v) è *all'indietro*
- Se *u.dt < v.dt* e *v.ft < u.ft* l'arco (u,v) è *in avanti*
- Se *v.ft < u.dt* l'arco (u,v) è *di attraversamento a sx*
- NOTA: non possono esistere altri casi! (ovvero *u.ft < v.dt*)



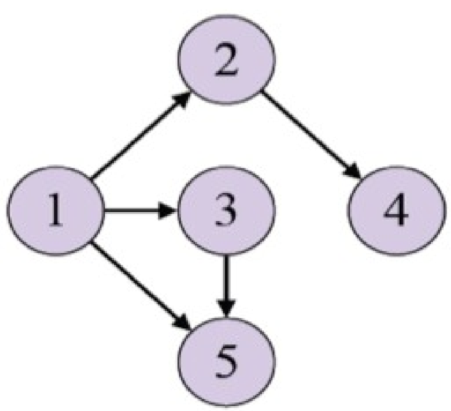


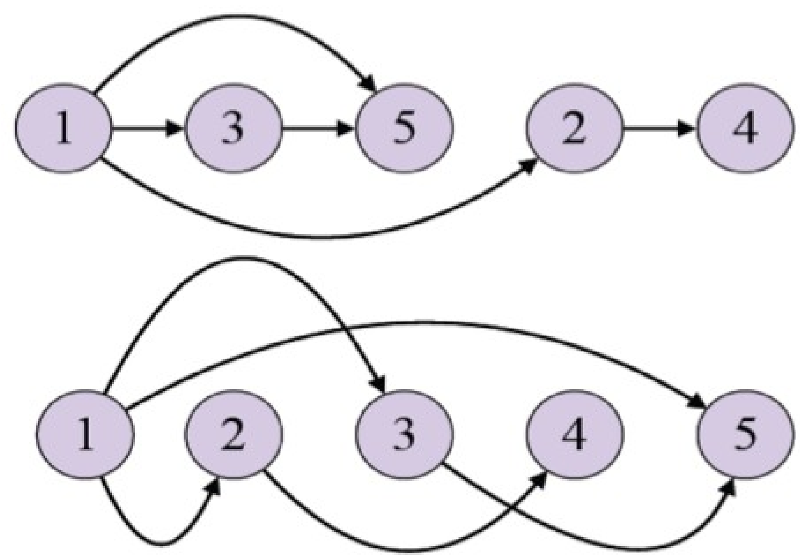
Applicazioni DFS

- Verificare DAG (Direct Acyclic Graph):
  - Basta verificare che non ci siano archi all'indietro
- Ordinamento topologico (in DAG)
- Individuare
  - le componenti connesse di un grafo non orientato
  - le componenti fortemente connesse di un grafo orientato (algoritmo “avanzato” che potete studiare dal libro di testo)

Ordinamento topologico

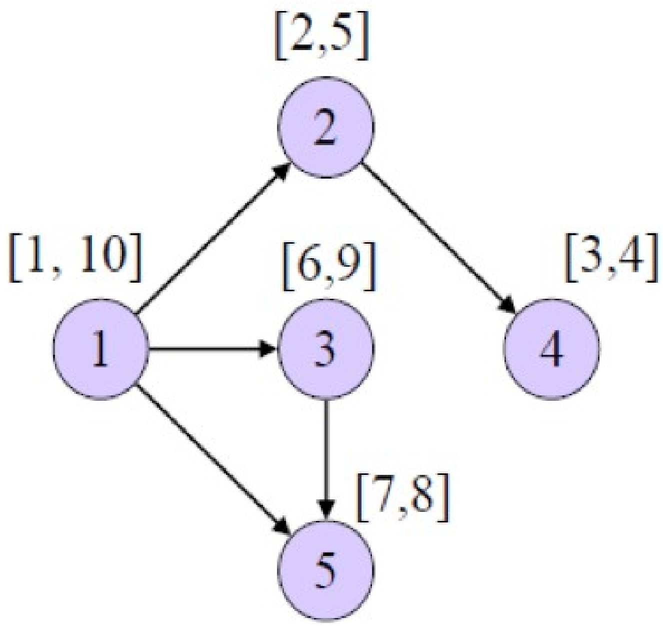
- Dato un DAG G (direct acyclic graph), un ordinamento topologico su G è un ordinamento lineare dei suoi vertici tale per cui:
  - Se G contiene l’arco (u,v), allora u compare prima di v nell’ordinamento
  - Per transitività, ne consegue che se v è raggiungibile da u, allora u compare prima di v nell'ordinamento
- Nota: possono esserci più ordinamenti topologici





Algoritmo per ordinamento topologico

- Algoritmo:
  - Si effettua una DFS
  - L'operazione di visita aggiunge il nodo alla testa di una lista "at finish time"
  - Restituire la lista di vertici
- Output
  - Sequenza ordinata di vertici, in ordine inverso di finish time
  - Domanda: quale sarà l'output nell'esempio?



Componenti connesse (grafo non orientato)

- Due vertici u e v appartengono alla stessa componente connessa se u è raggiungibile da v
- La relazione “u è raggiungibile da v” è di equivalenza
  - Riflessiva
    - u è raggiungibile da se stesso
  - Simmetrica
    - Se u è raggiungibile da v, allora esiste un cammino che connette u e v. Tale cammino può essere percorso a ritroso per dimostrare che v è raggiungibile da u
  - Transitiva
    - Se u è raggiungibile da v, e v è raggiungibile da w, allora u è raggiungibile da w.

Componenti connesse (grafo non orientato)

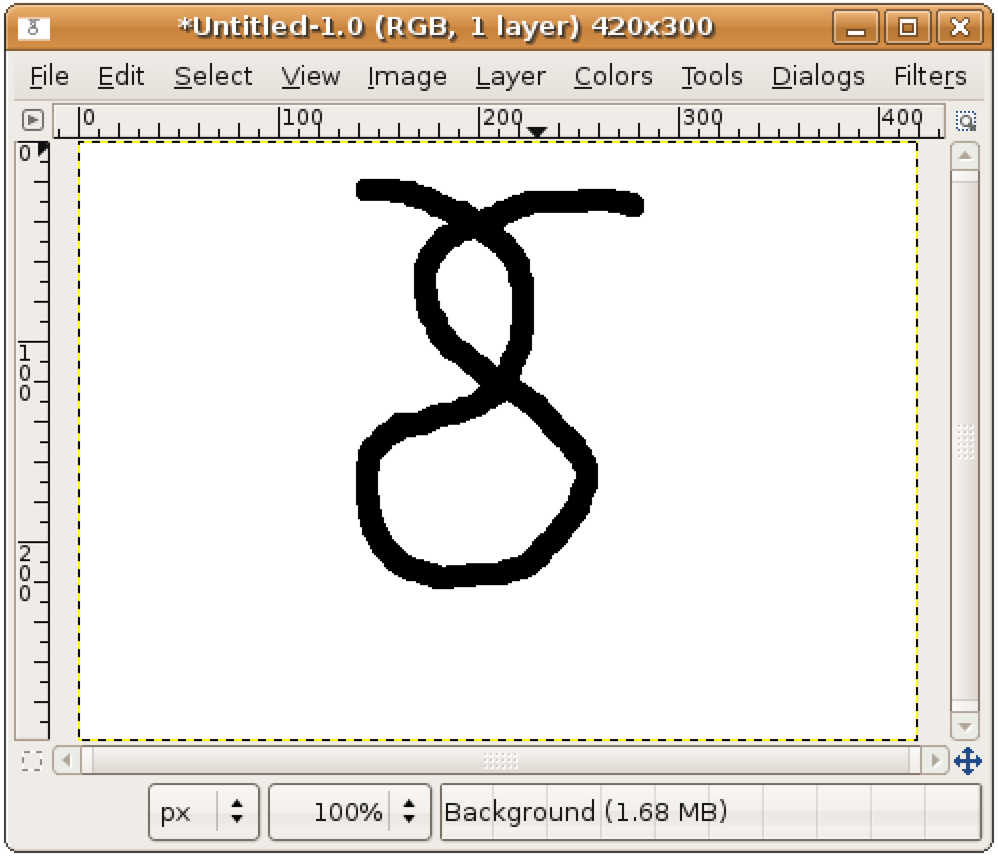
- Poiché la relazione di raggiungibilità è di equivalenza, possiamo concludere che tutti i nodi raggiungibili da un nodo sorgente u (incluso u) appartengono alla stessa componente connessa

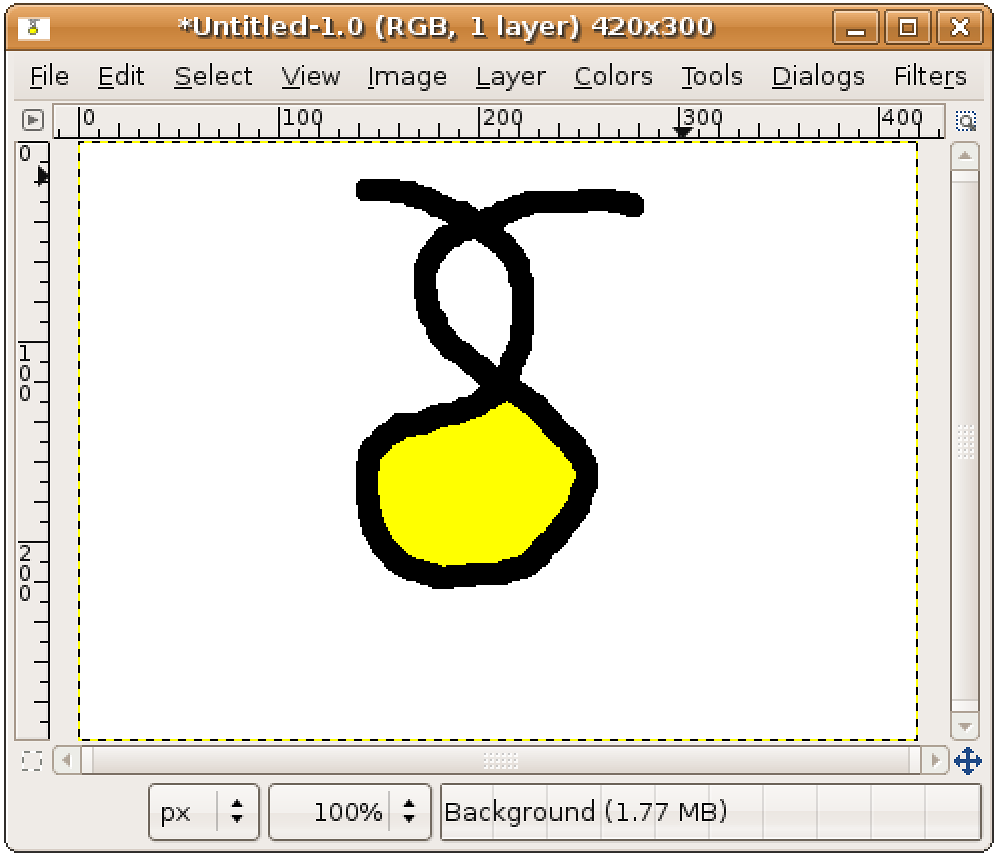
Componenti Connesse

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

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

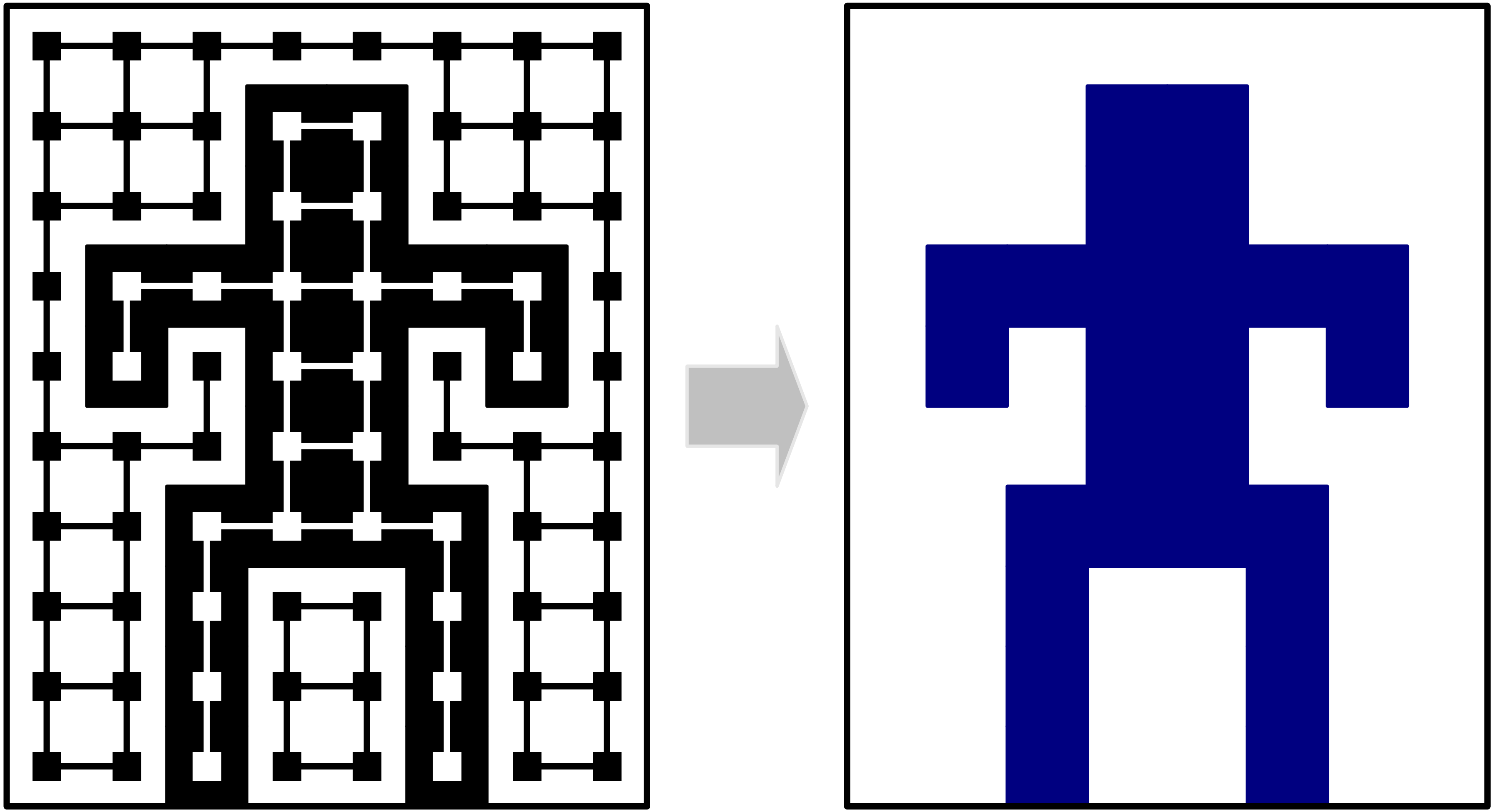
Etichetta con il valore *k* tutti i nodi della stessa componente connessa cui appartiene *u*

Applicazione: floodfill





Applicazione: floodfill

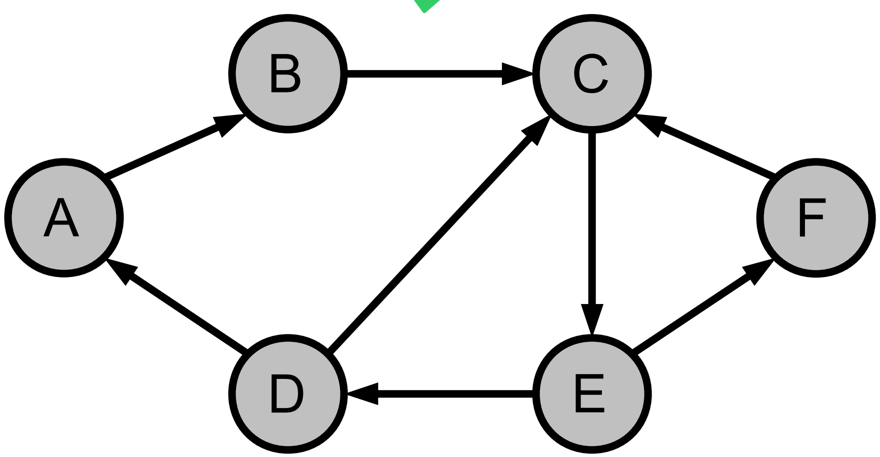


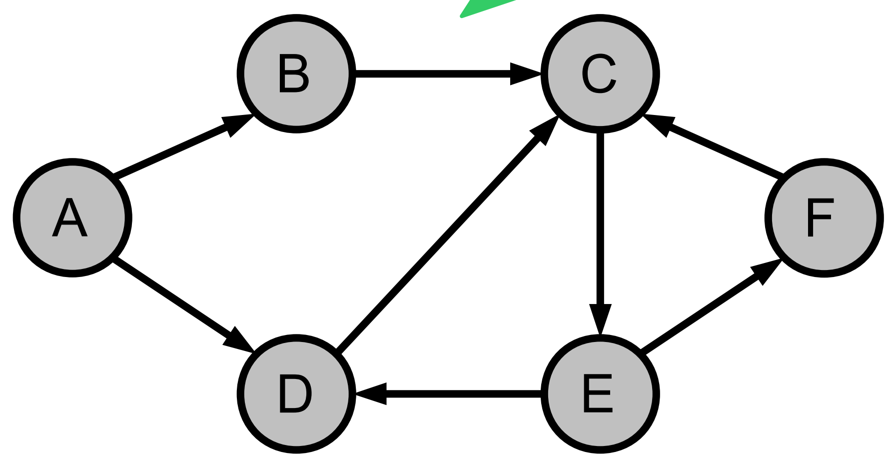
Componenti fortemente connesse
(Strongly Connected Components)

- Ricordiamo: un grafo <u>orientato</u> G è fortemente connesso se ogni coppia di vertici è connessa da un cammino

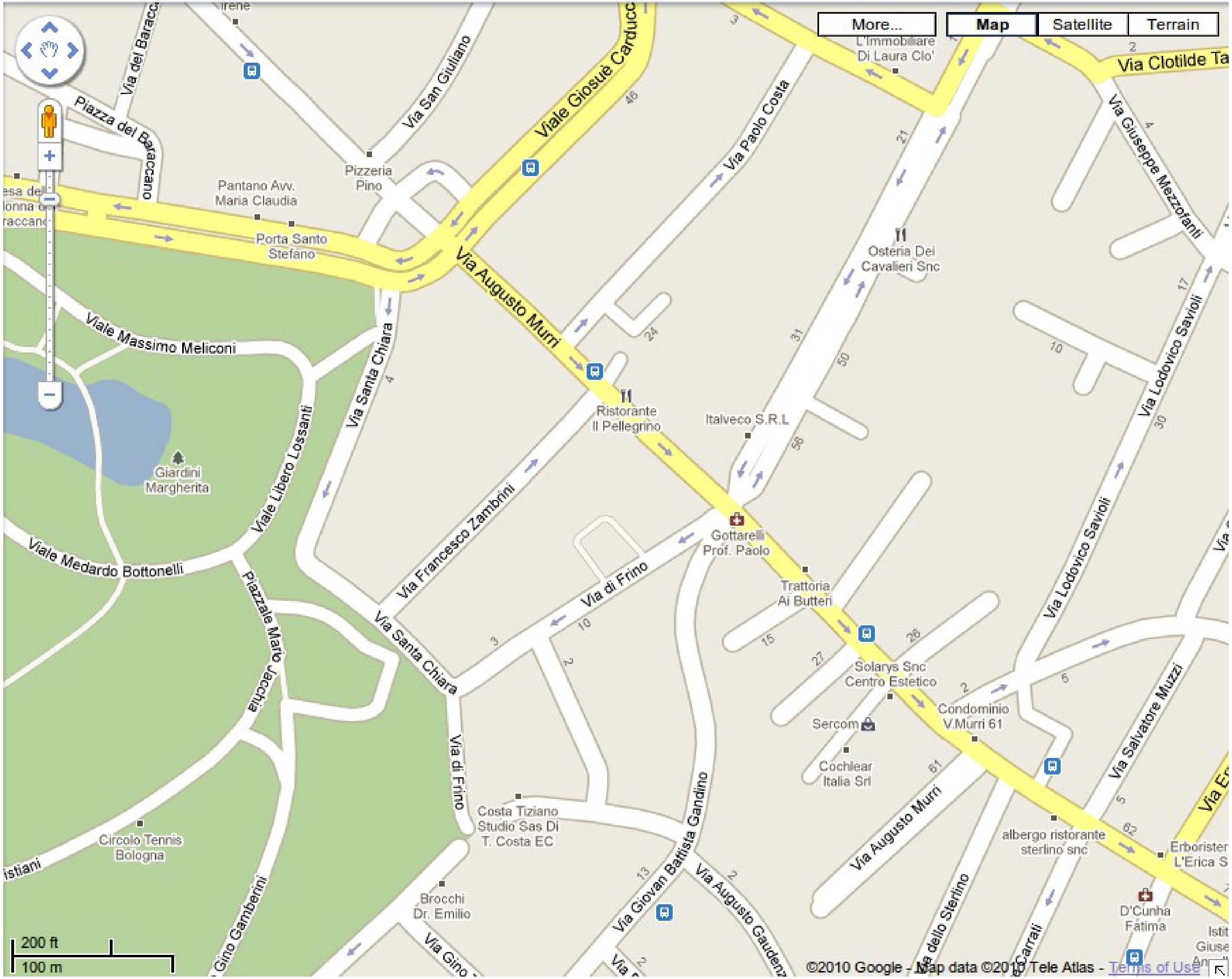
Questo grafo orientato ***è fortemente connesso.***

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

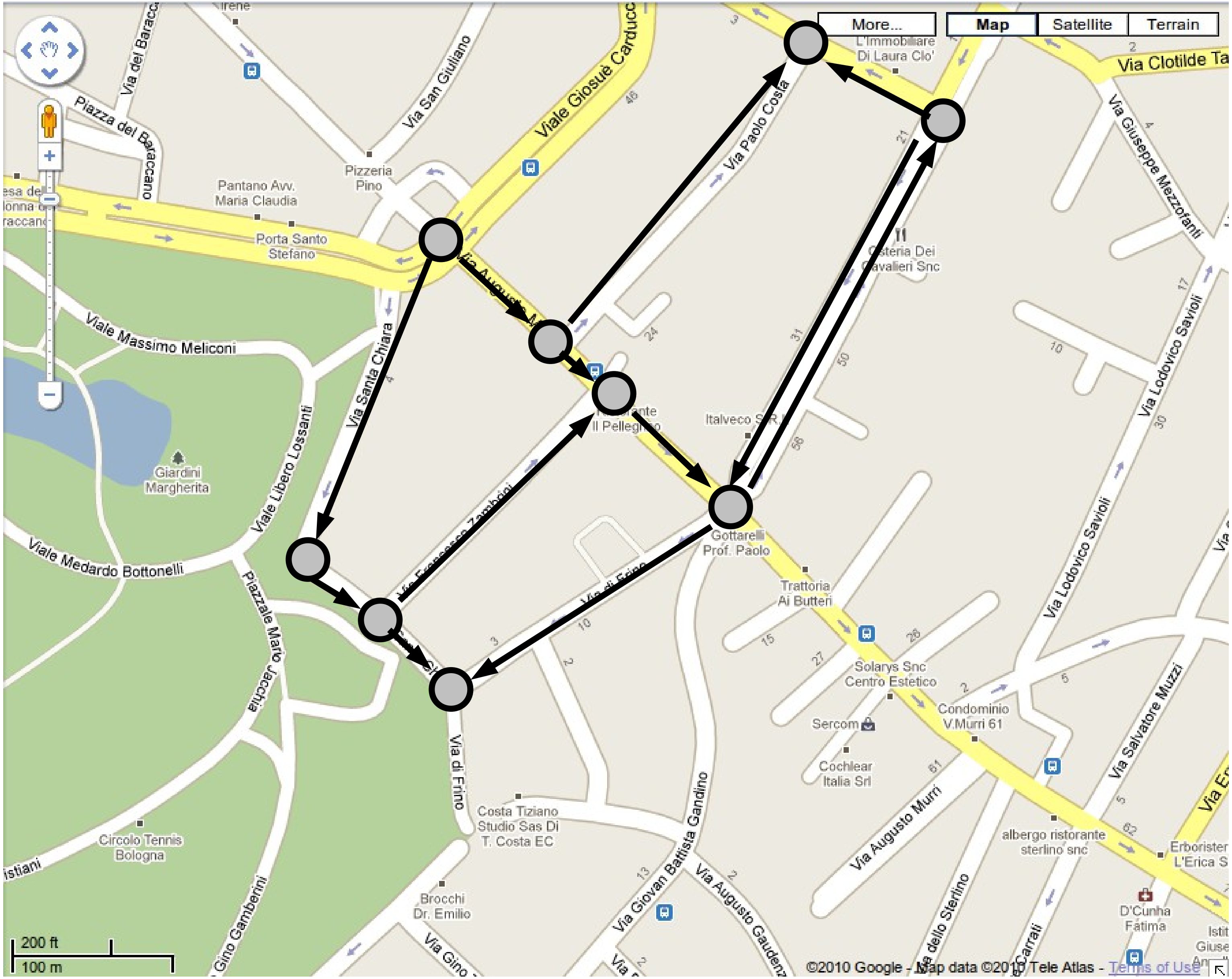




Nel mondo reale

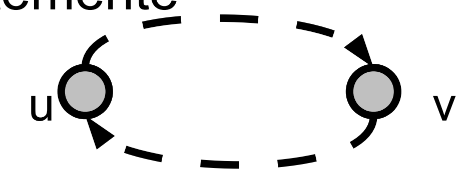


Nel mondo reale

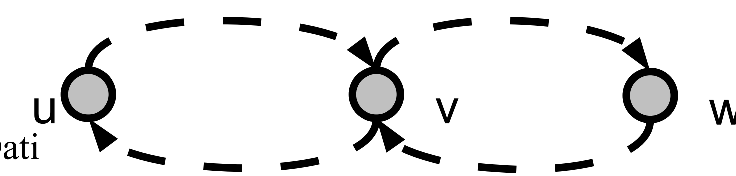


Componenti fortemente connesse (grafo orientato)

- 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.



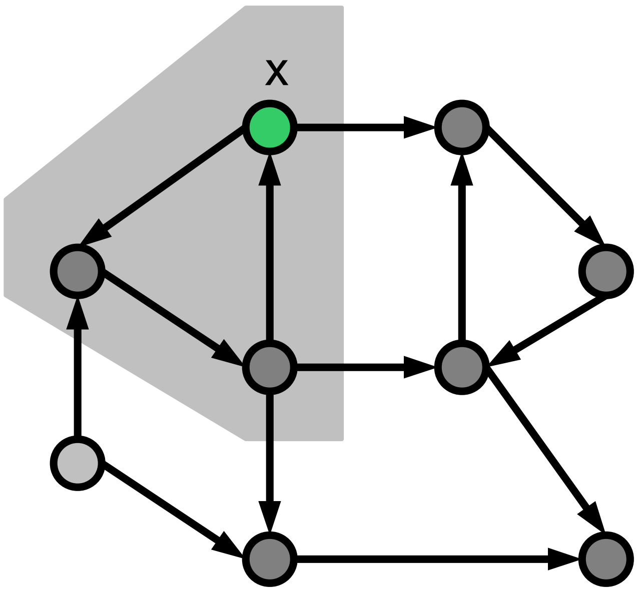
Idea

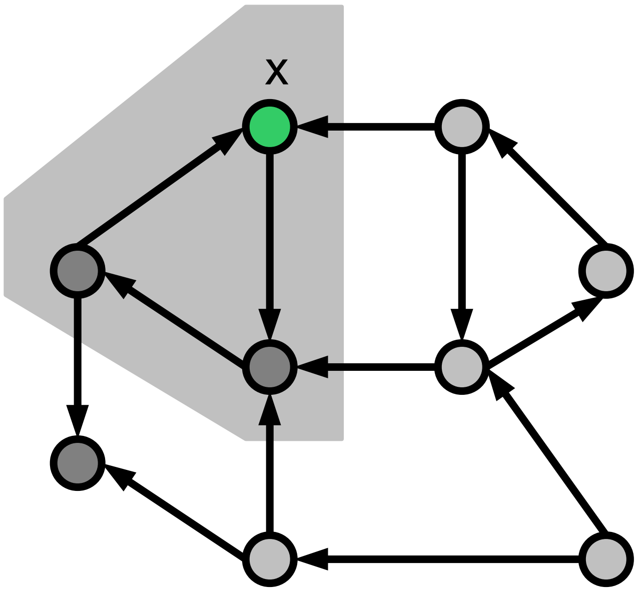
- Due nodi u e v appartengono alla stessa componente fortemente connessa se e solo se valgono entrambe le seguenti proprietà
  - Esiste un cammino u→→v
    - cioè v è discendente di u in una visita DF che usa u come sorgente
  - Esiste un cammino v→→u
    - cioè u è discendente di v in una visita DF 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 (ad esempio BFS) usando x come sorgente
- Come calcolare A(x)?
  - È sufficiente invertire la direzione di tutti gli archi, ed effettuare una nuova visita (ad esempio BFS) usando ancora x come sorgente
- Nota: il calcolo di A(x) o D(x) richiede tempo O(n+m)

Algoritmo (schematico)

```
algoritmo SCC(Grafo G, nodo x) → lista di nodi
    L := lista vuota di nodi
(1) Esegui BFS(G, x) marcando i nodi visitati
(2) Calcola il grafo trasposto G^T
        (inverti la direzione degli archi di G)
(3) Esegui BFS(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 <u>tutte</u> 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, basato specificatamente su DFS, che elenca tutte le SCC di un grafo G in tempo complessivo O(n+m). Potete studiarlo sul libro di testo.