Algoritmi di Visita di Grafi

Jocelyne Elias
https://www.unibo.it/sitoweb/jocelyne.elias
Moreno Marzolla
https://www.moreno.marzolla.name/

Dipartimento di Informatica—Scienza e Ingegneria (DISI)
Università di Bologna

Visita di Grafi

- Definizione del problema
  - Dato un grafo $G = (V, E)$ (può essere orientato oppure non orientato) ed un nodo s (detto *sorgente*), visitare ogni nodo raggiungibile da s
  - Ogni nodo deve essere visitato una sola volta

- Visita in ampiezza (Breadth-First Search, BFS)
  - Visita i nodi "espandendo" la frontiera dei nodi scoperti
- Visita in profondità (Depth-First Search, DFS)
  - Visita i nodi andando il "più lontano possibile" nel grafo

Visita in Ampiezza
(*Breadth-First Search*, ***BFS***)

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**
  - Distanza di un nodo = numero di archi attraversati per raggiungerlo a partire dalla sorgente *s*

- 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

Visita in ampiezza

```
BFS(Grafo G, nodo s)
  for each v in V do
    v.parent ← null;
    v.dist ← +infinity;
  endfor
  Queue F;
  s.dist ← 0;
  F.enqueue(s);
  while ( not F.isEmpty() ) do
    u ← F.dequeue();
    // fa qualcosa sul nodo u
    for each v adiacente a u do
      if ( v.dist = +infinity ) then
        v.dist ← u.dist + 1;
        v.parent ← u;
        F.enqueue(v);
      endif
    endfor
  endwhile
```

- *v.dist* è la distanza del nodo *v* da *s*
- *v.parent* indica il nodo di provenienza

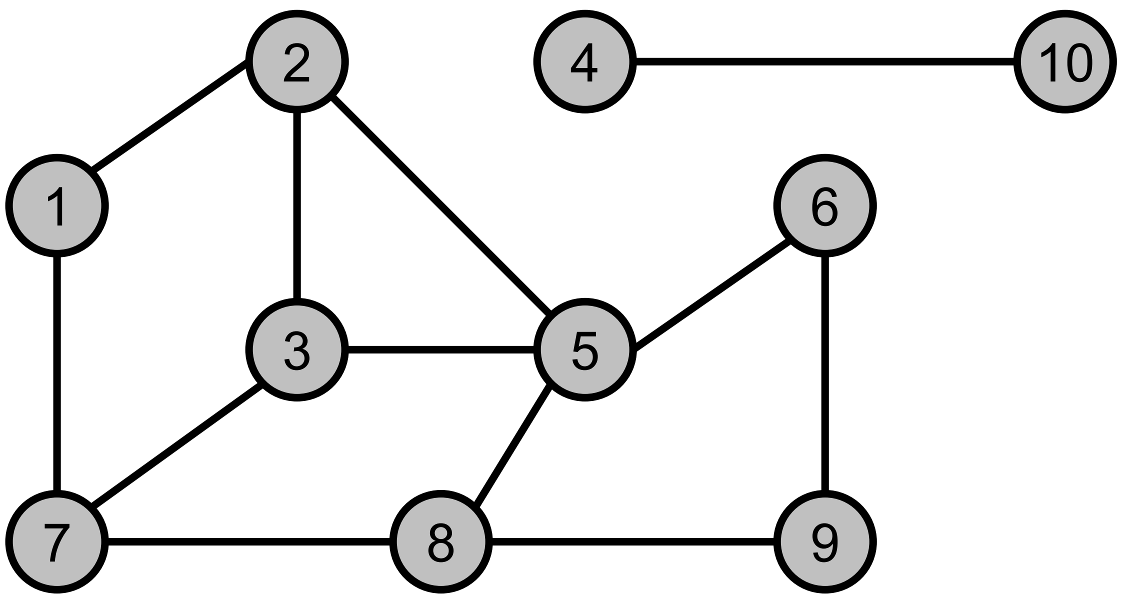
Ogni nodo deve essere visitato una sola volta

Algoritmo generico per la visita

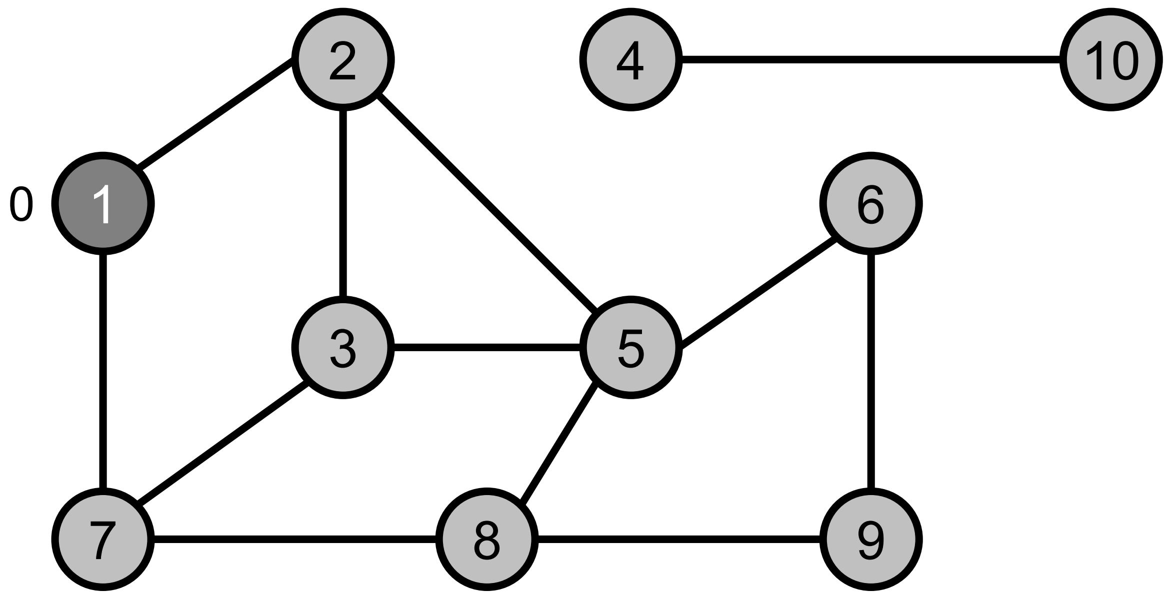
- Alcune cose da notare:
  - I nodi vengono visitati al più una volta
  - Una volta rimosso dalla coda, un nodo non viene più reinserito
  - Tutti i nodi raggiungibili da *s* vengono visitati
  - Ciascun arco viene "percorso" al più due volte nel caso dei grafi non orientati ({u,v}, {v,u}).
- Costo computazionale
  - $O(n + m)$ usando liste di adiacenza
  - $O(n^2)$ usando matrice di adiacenza
  - n è il numero di nodi, m è il numero di archi

Esempio di visita BFS

- Supponiamo che le liste di adiacenza siano ordinate rispetto all'id dei nodi

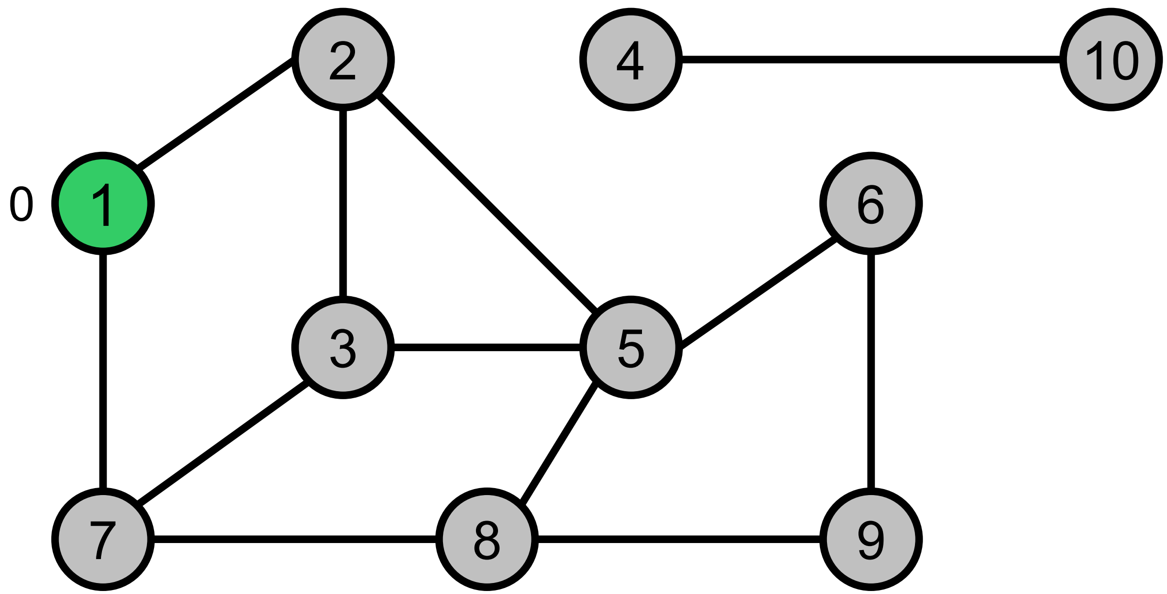


Esempio



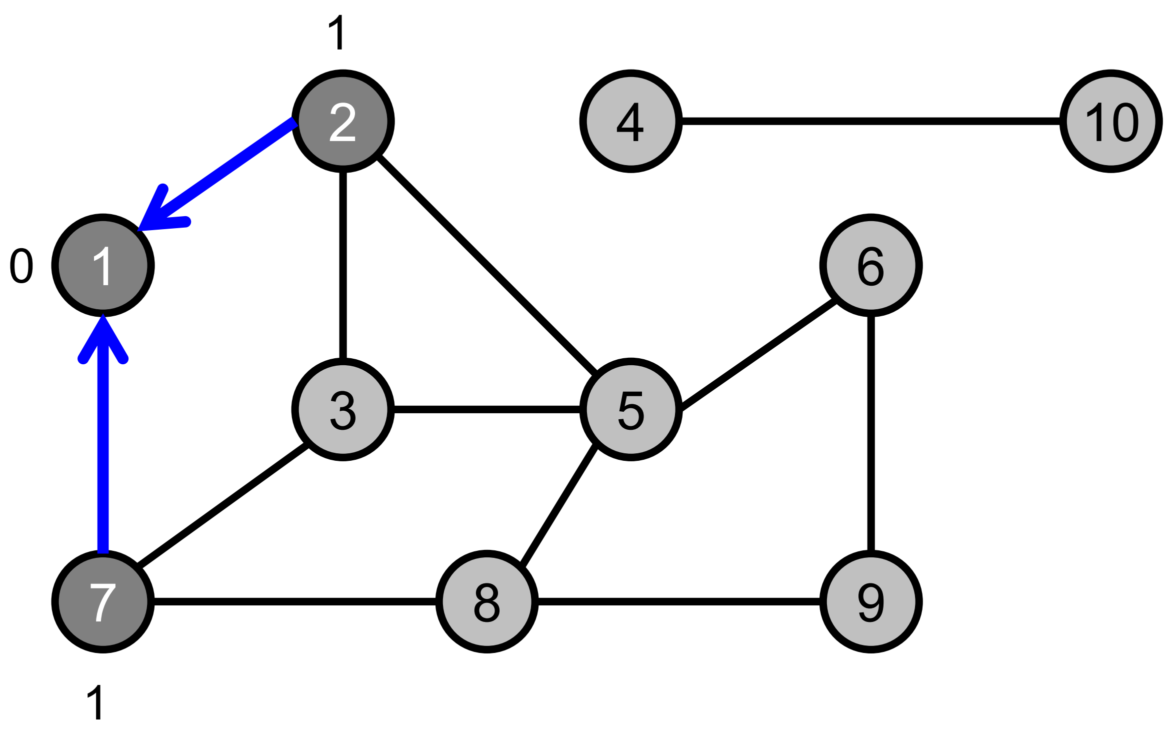
Coda = { 1 }

Esempio



Coda = { }

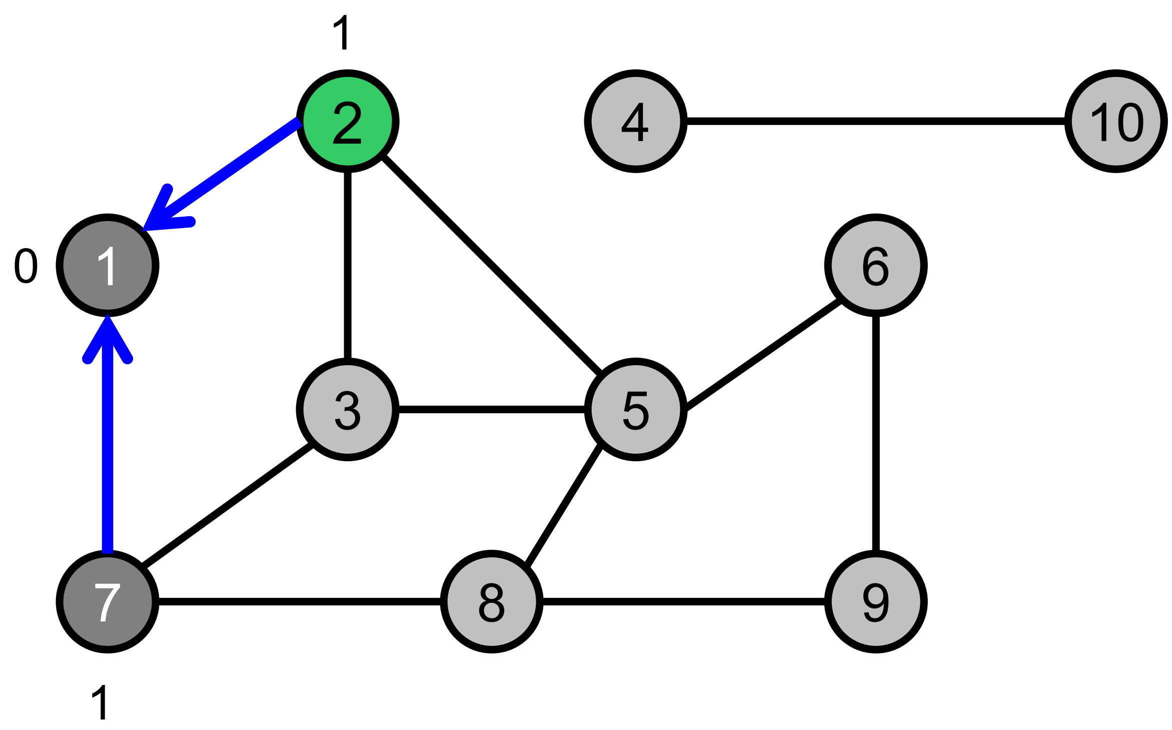
Esempio



Il nodo 1 è il nodo di provenienza del nodo 2 e 7

Coda = { 2, 7 }

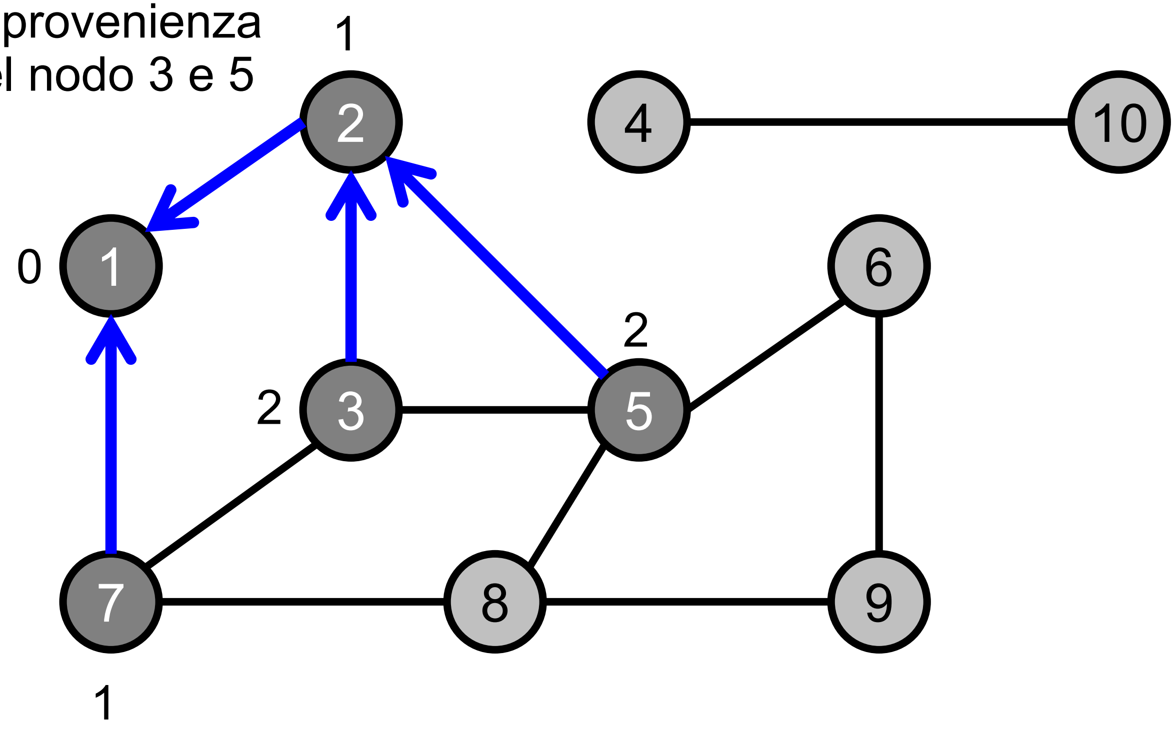
Esempio



Coda = { 7 }

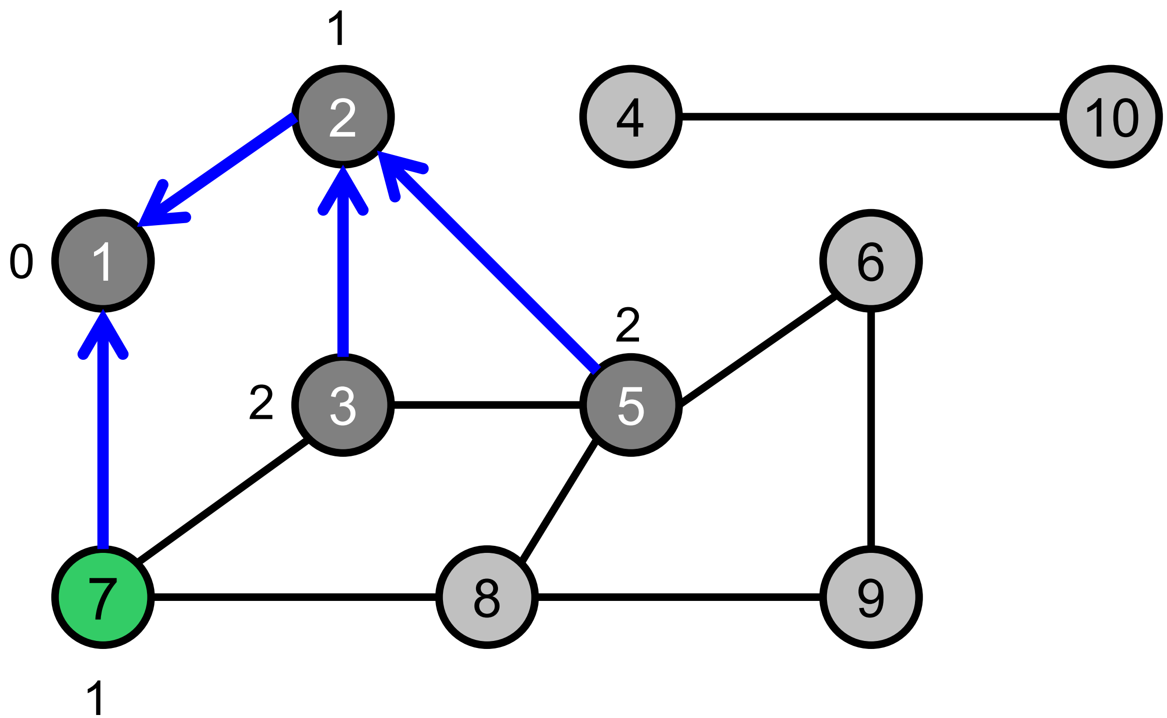
Esempio

Il nodo 2 è il nodo
di provenienza
del nodo 3 e 5



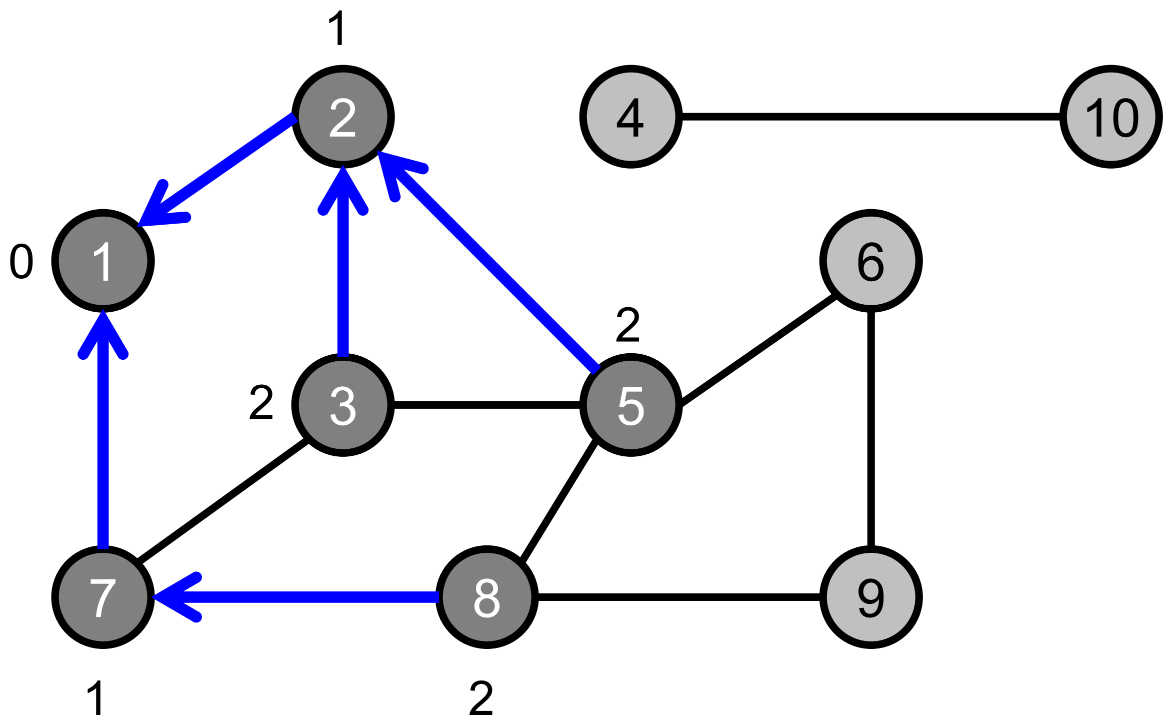
Coda = { 7, 3, 5 }

Esempio



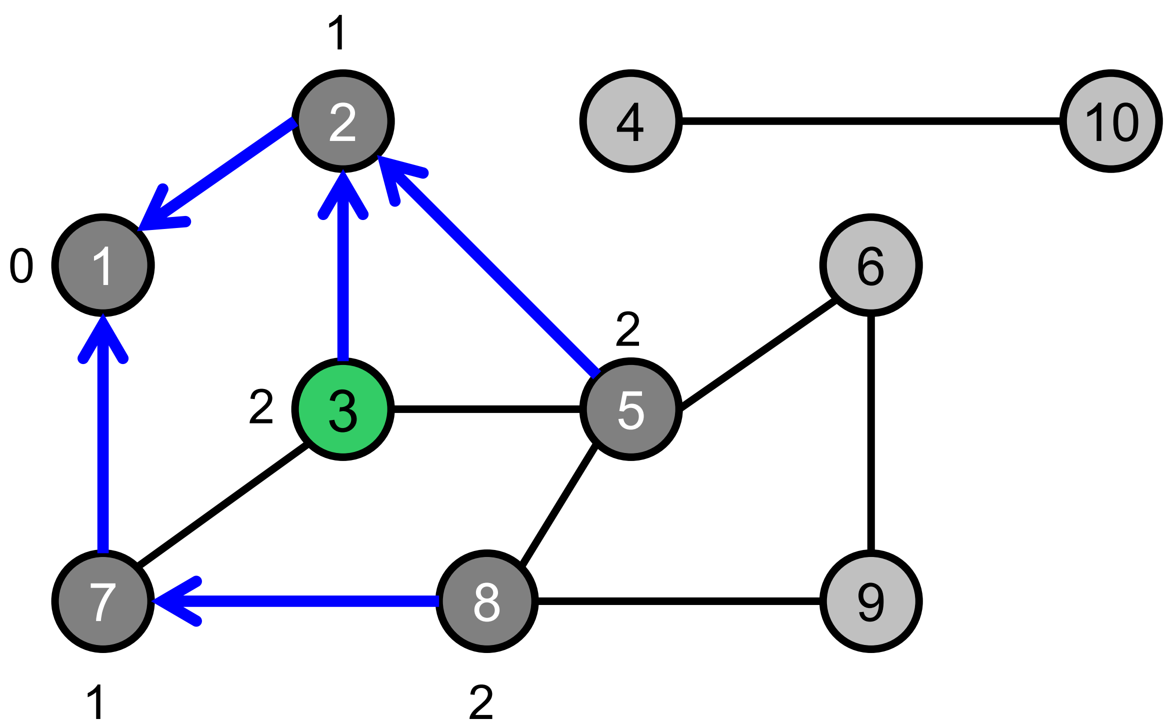
Coda = { 3, 5 }

Esempio



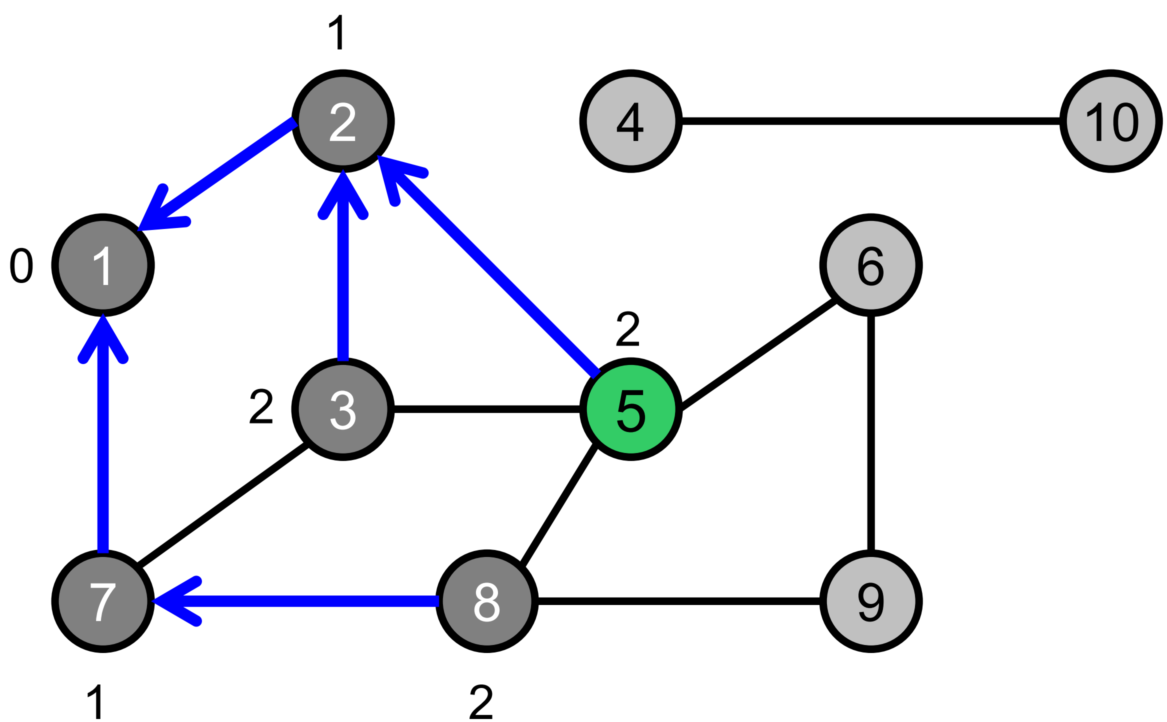
Coda = { 3, 5, 8 }

Esempio



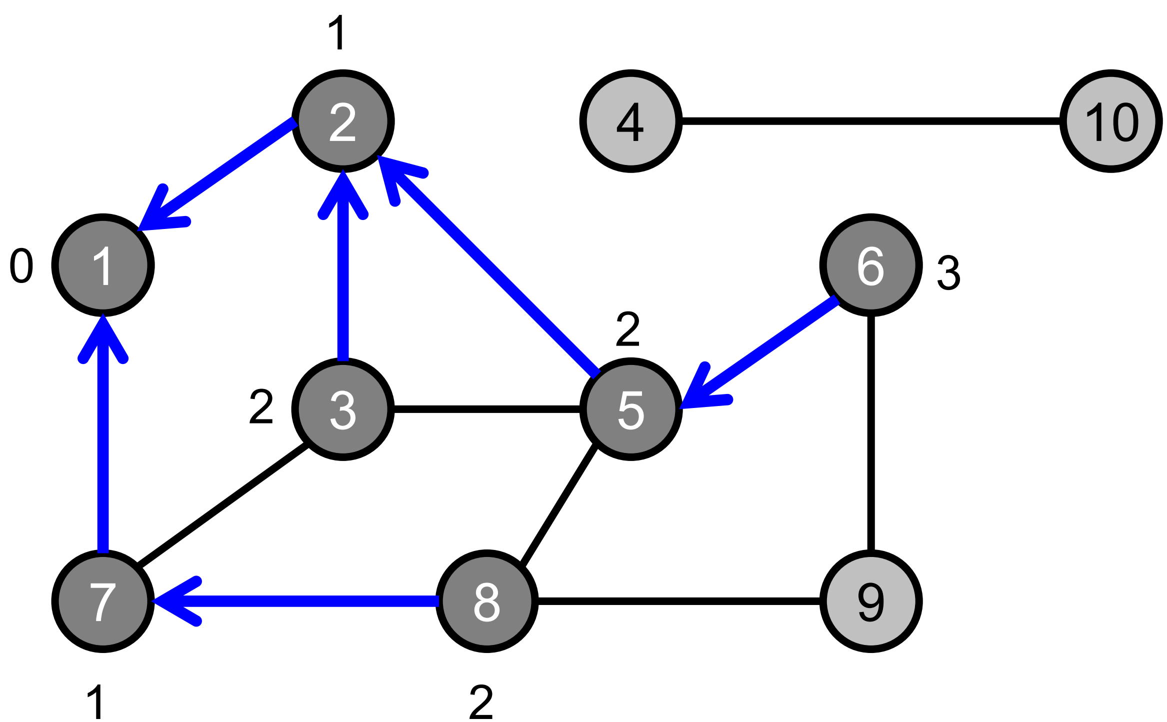
Coda = { 5, 8 }

Esempio



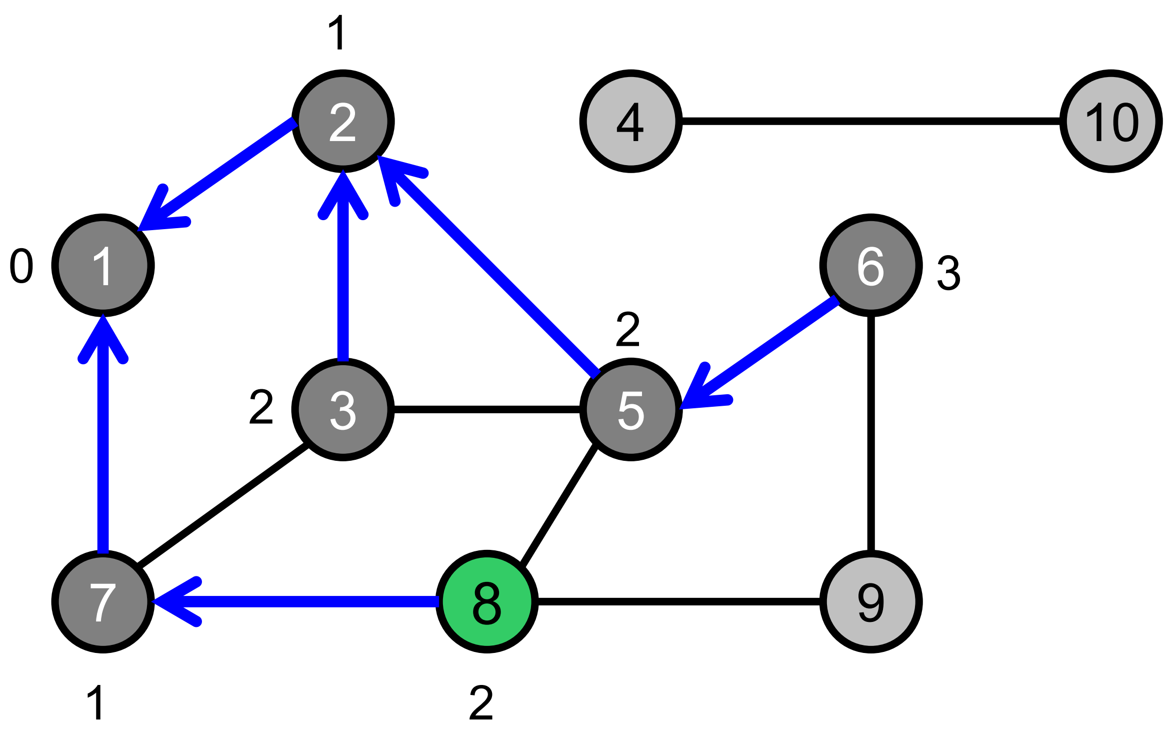
Coda = { 8 }

Esempio



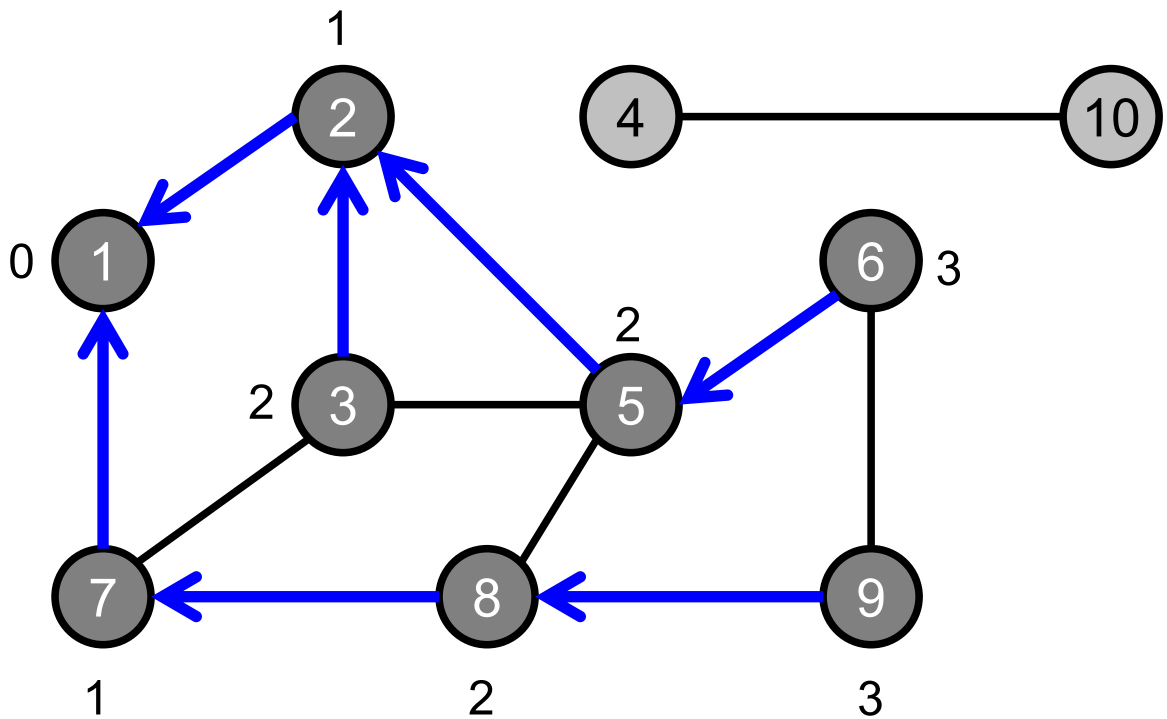
Coda = { 8, 6 }

Esempio



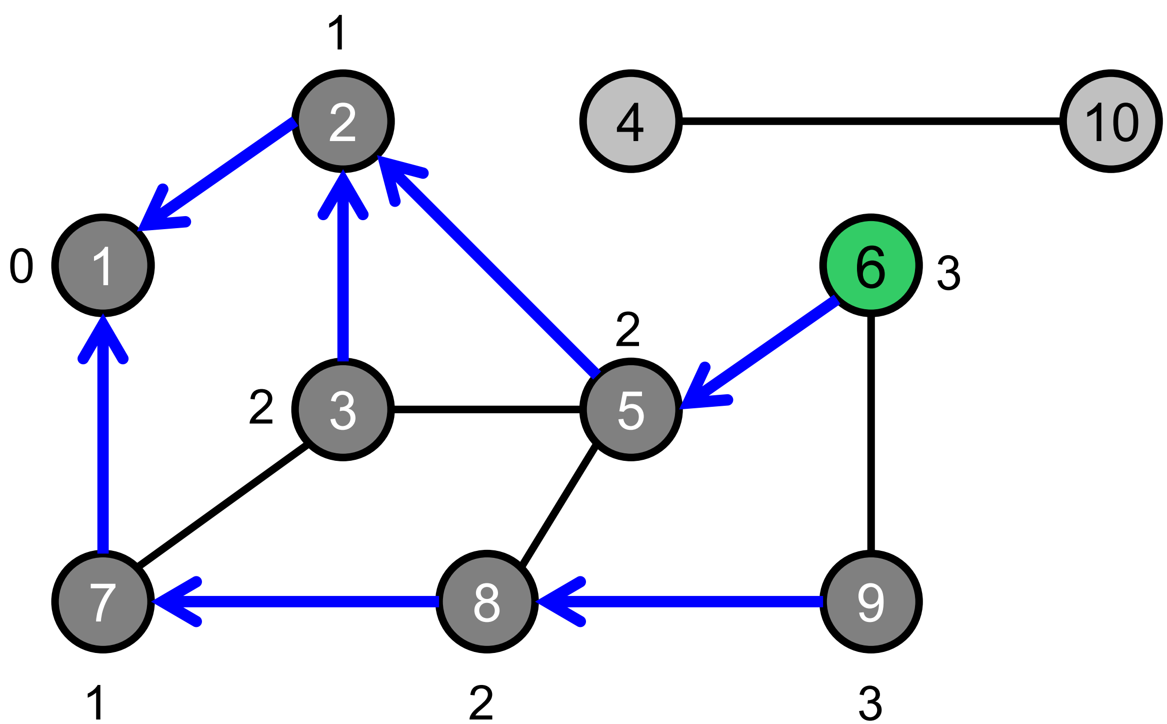
Coda = { 6 }

Esempio



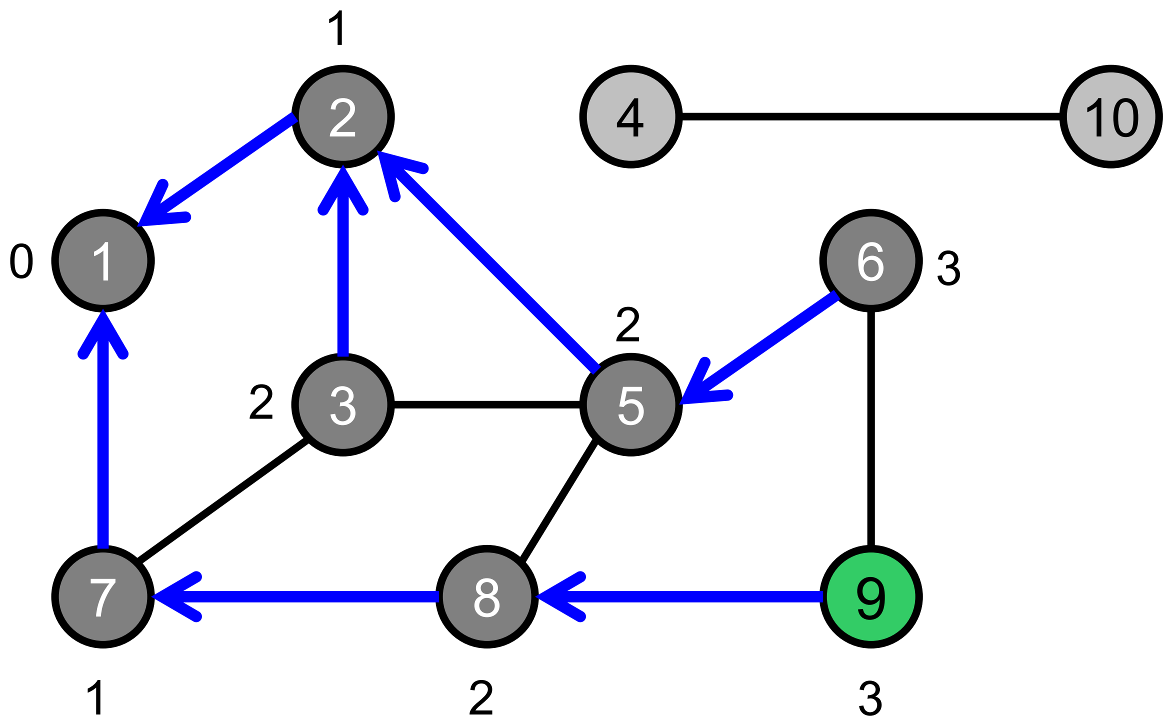
Coda = { 6, 9 }

Esempio



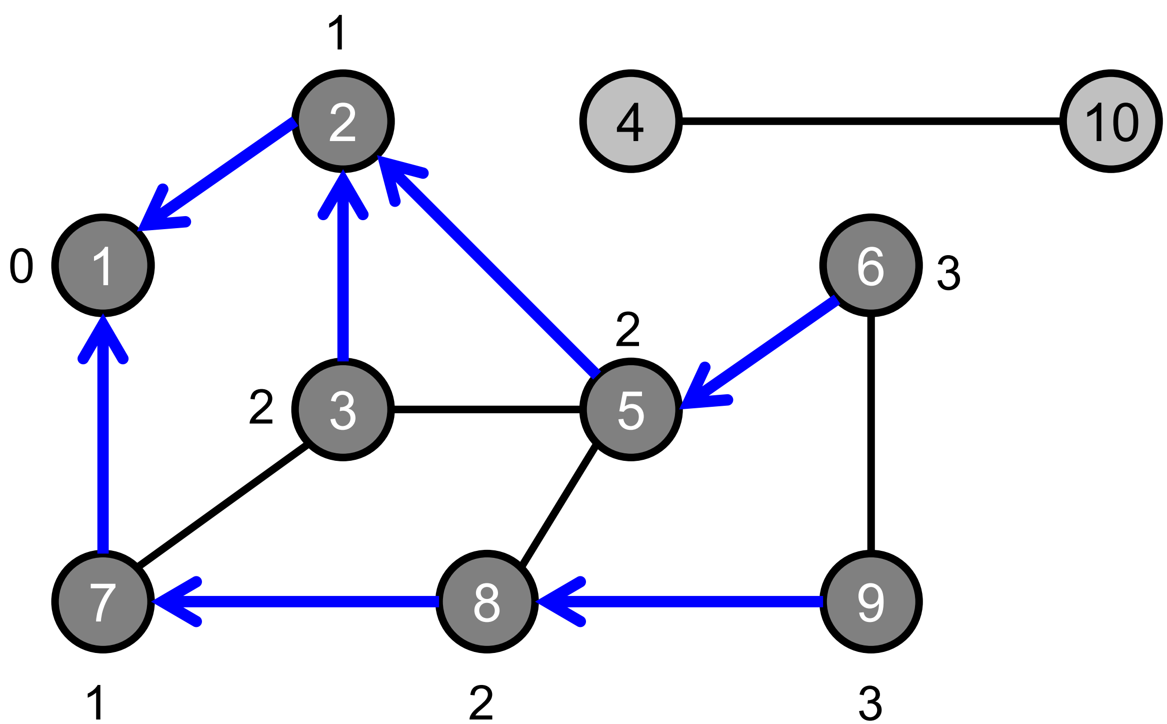
Coda = { 9 }

Esempio



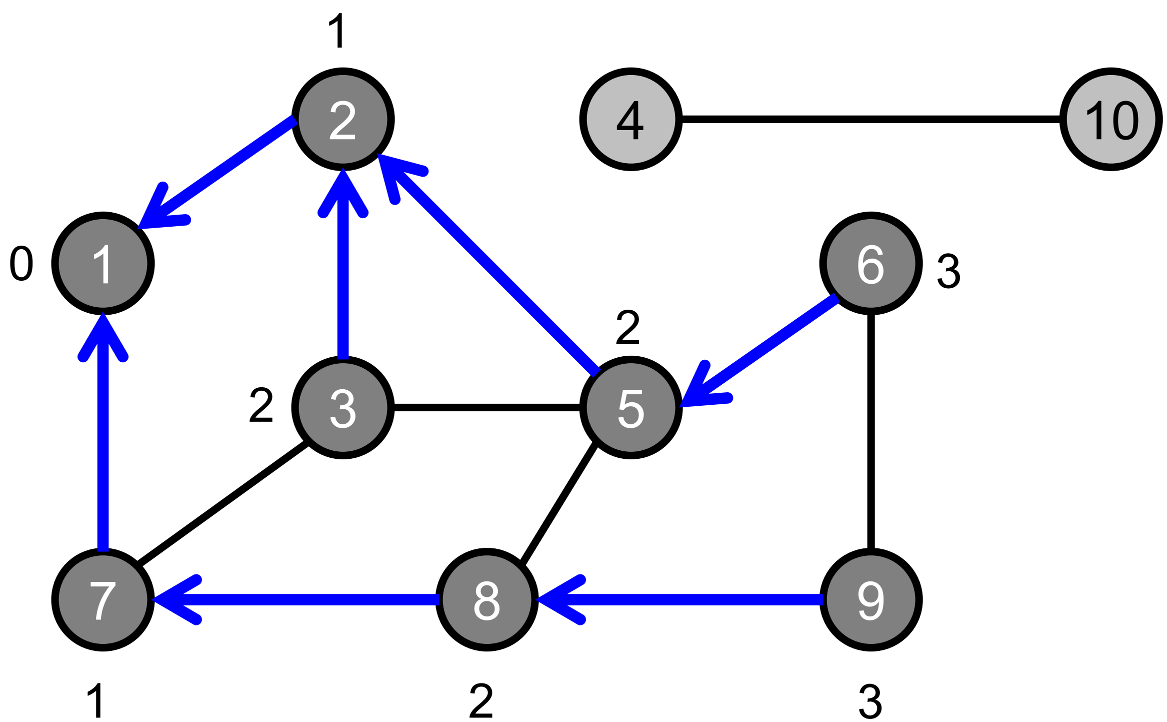
Coda = { }

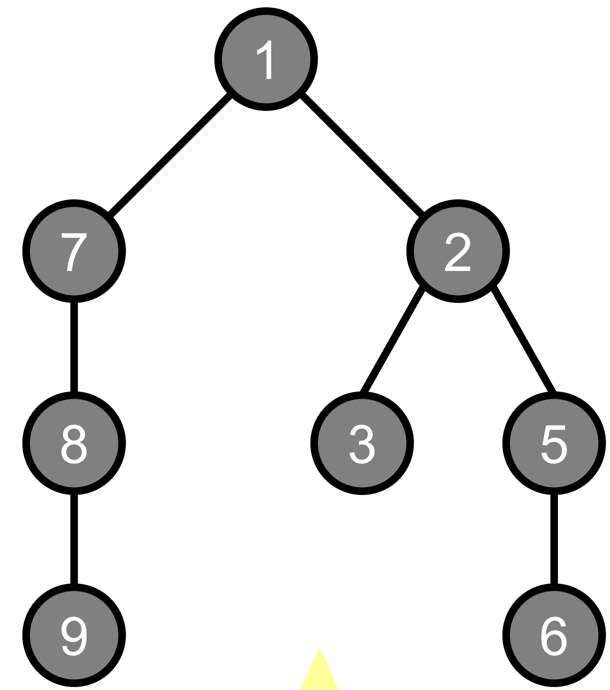
Esempio



Coda = { }

Fine esecuzione BFS





Albero prodotto dalla visita BFS

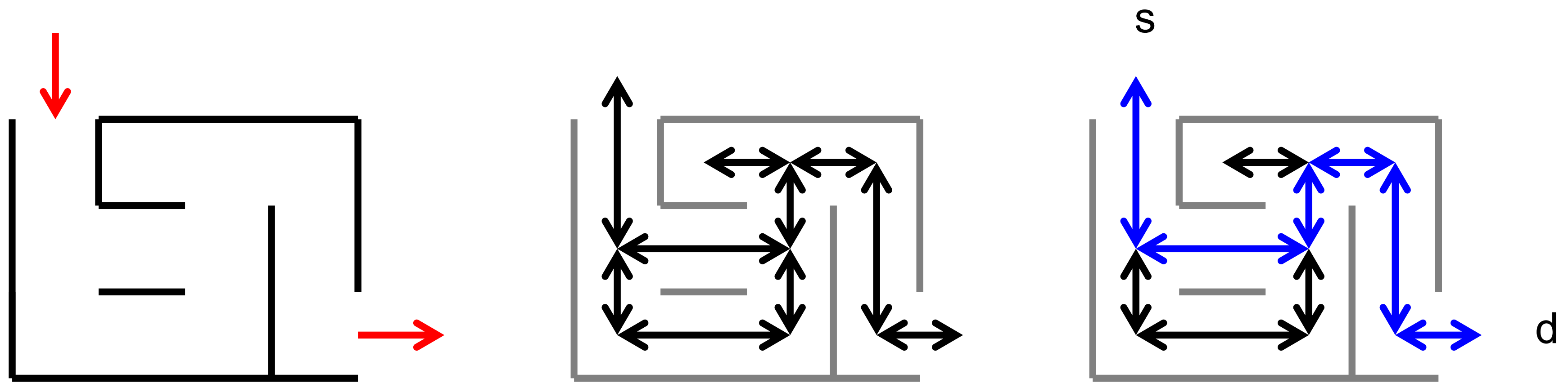
Applicazioni

- 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*, **`print-path(G, s, v)`** stampa il cammino più breve da *s* a *v*

```
print-path(Grafo G, nodo s, nodo v)
  if (v = s) then
    print s
  else if (v.parent = null) then
    print “no path from s to v”
  else
    print-path(G, s, v.parent)
    print v
  endif
```

Esempio

Percorso più breve per uscire dal labirinto?



Visita in Profondità

(*Depth-First Search, **DFS***)

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 <u>foresta DF</u> (depth-first) $G_\pi=(V, E_\pi)$ contenente un insieme di alberi DF
  - Informazioni addizionali sul tempo di visita
- Tempo di scoperta (discovery time) di un nodo
- Tempo di “terminazione” (finish time) di un nodo

Visita in profondità (depth first search, DFS)

```
time ← 0; // var.globale

algoritmo DFS(Grafo G)
  for each u in V do
    u.mark ← WHITE;
    u.parent ← null;
  endfor
  for each u in V do
    if (u.mark = WHITE) then
      DFS-visit(u);
    endif
  endfor
```

- *time* è una variabile globale che contiene il numero di “passi” dell'algoritmo
- *v.dt* (*discovery time*) è il tempo in cui il nodo è stato scoperto
- *v.ft* (*finish time*) è il tempo in cui termina la visita del nodo e di tutti quelli da lui raggiungibili

Visita in profondità (depth first search, DFS)

```
algoritmo DFS-visit(Grafo G, Nodo 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 nodo u"
  time ← time + 1;
  u.ft ← time;
  u.mark ← BLACK;
```

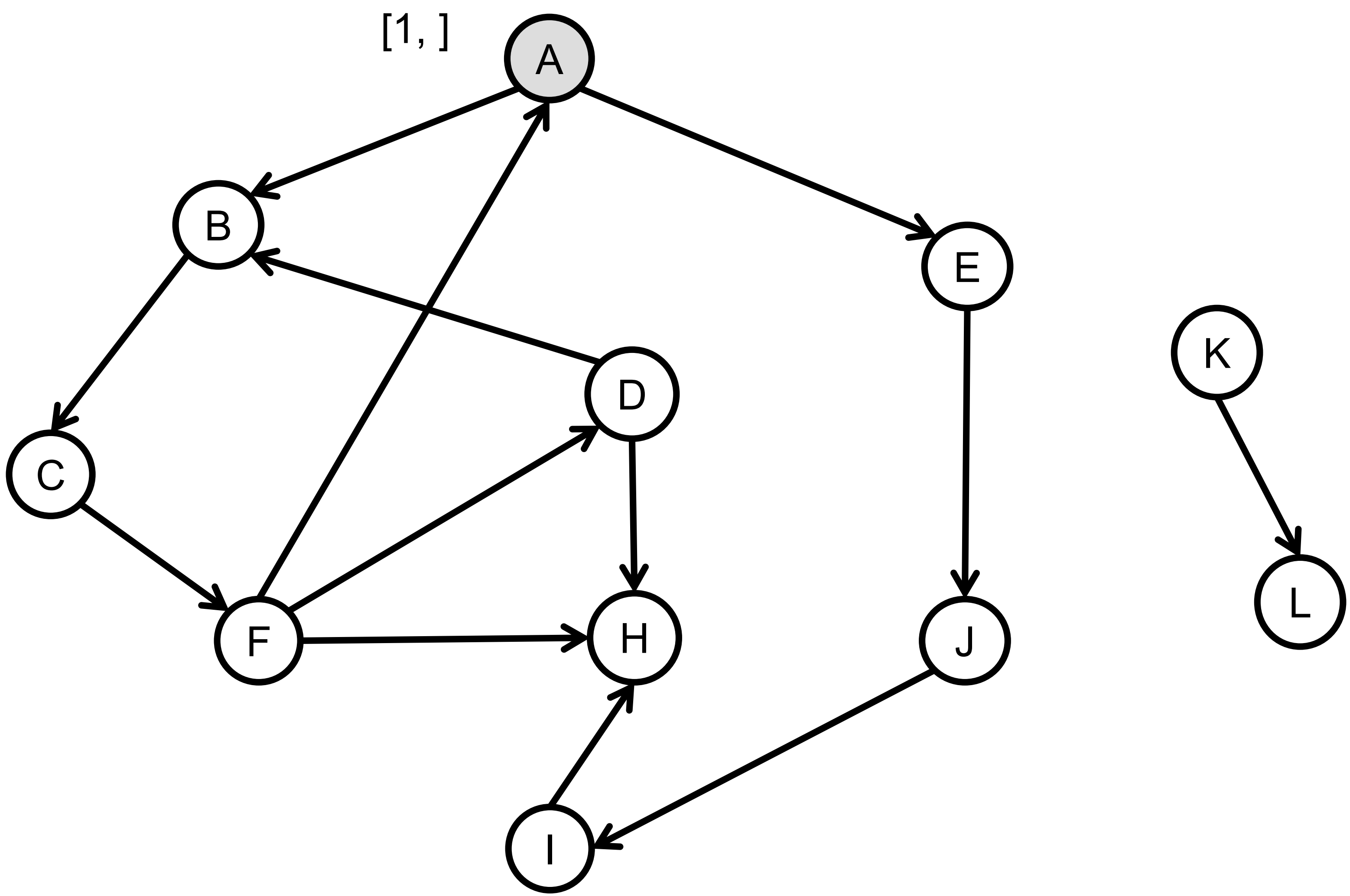
termina la visita del nodo e <u>di tutti quelli da lui raggiungibili</u>

Nodi bianchi = inesplorati

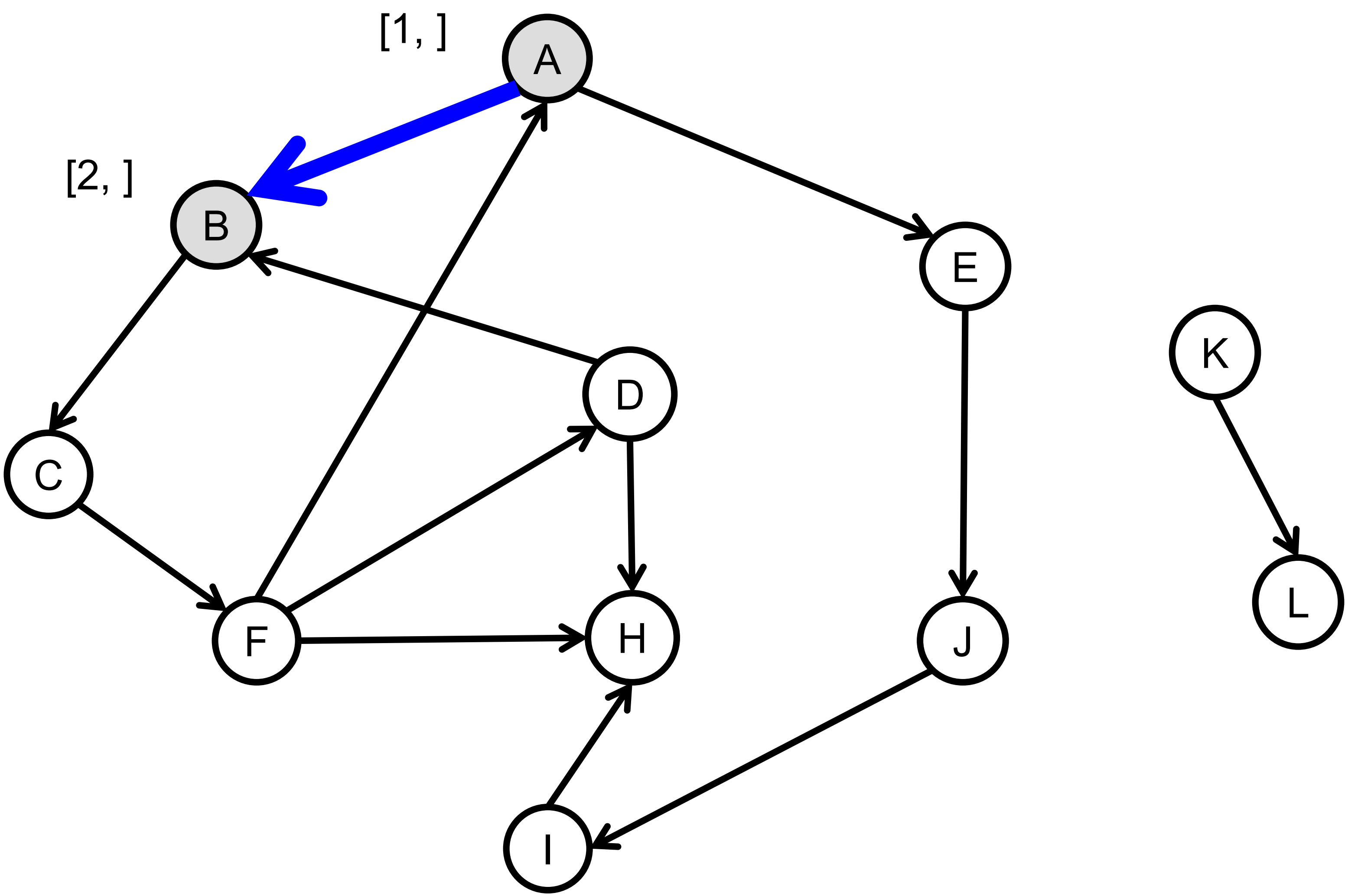
Nodi grigi = aperti

Nodi neri = chiusi

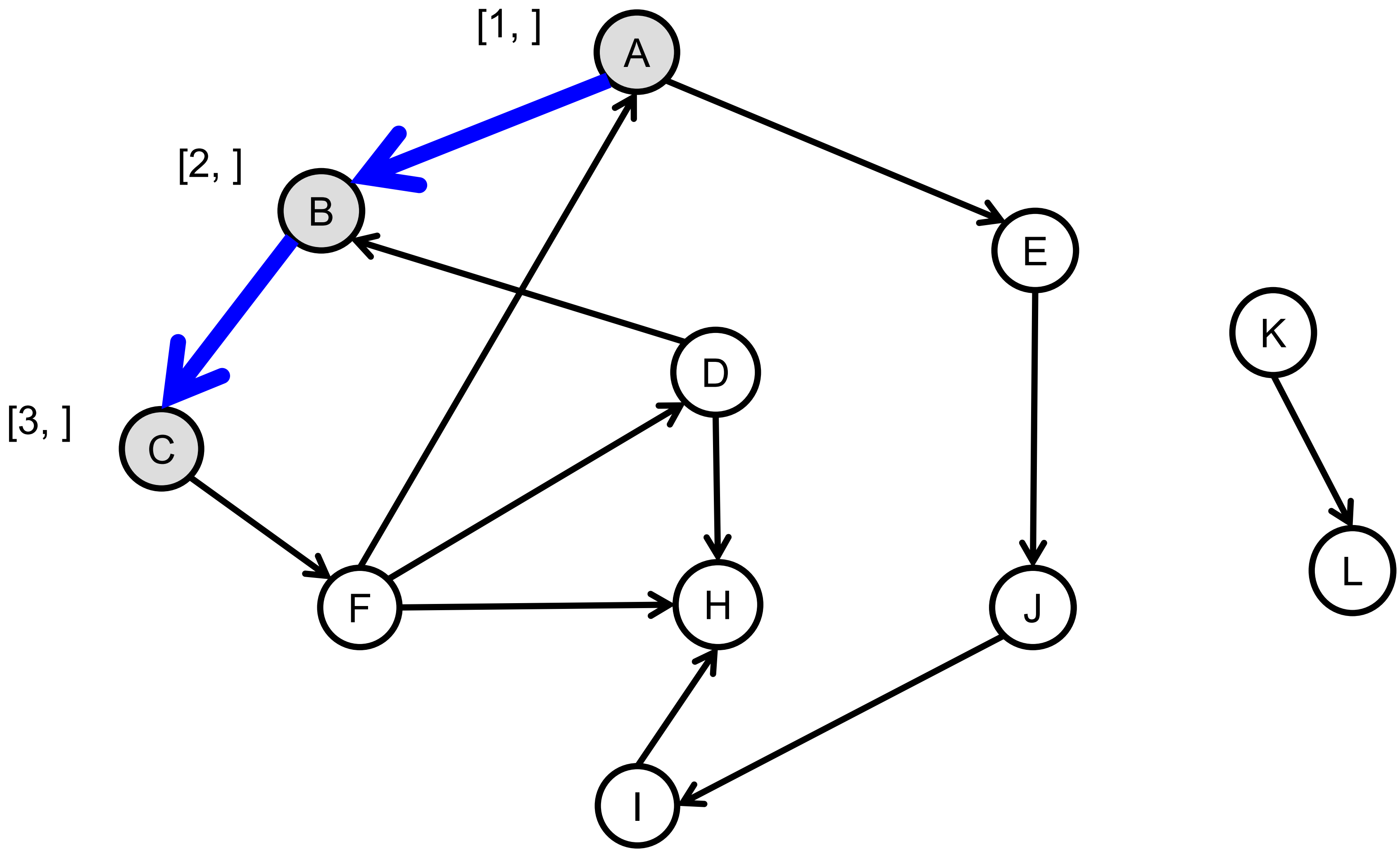
Esempio



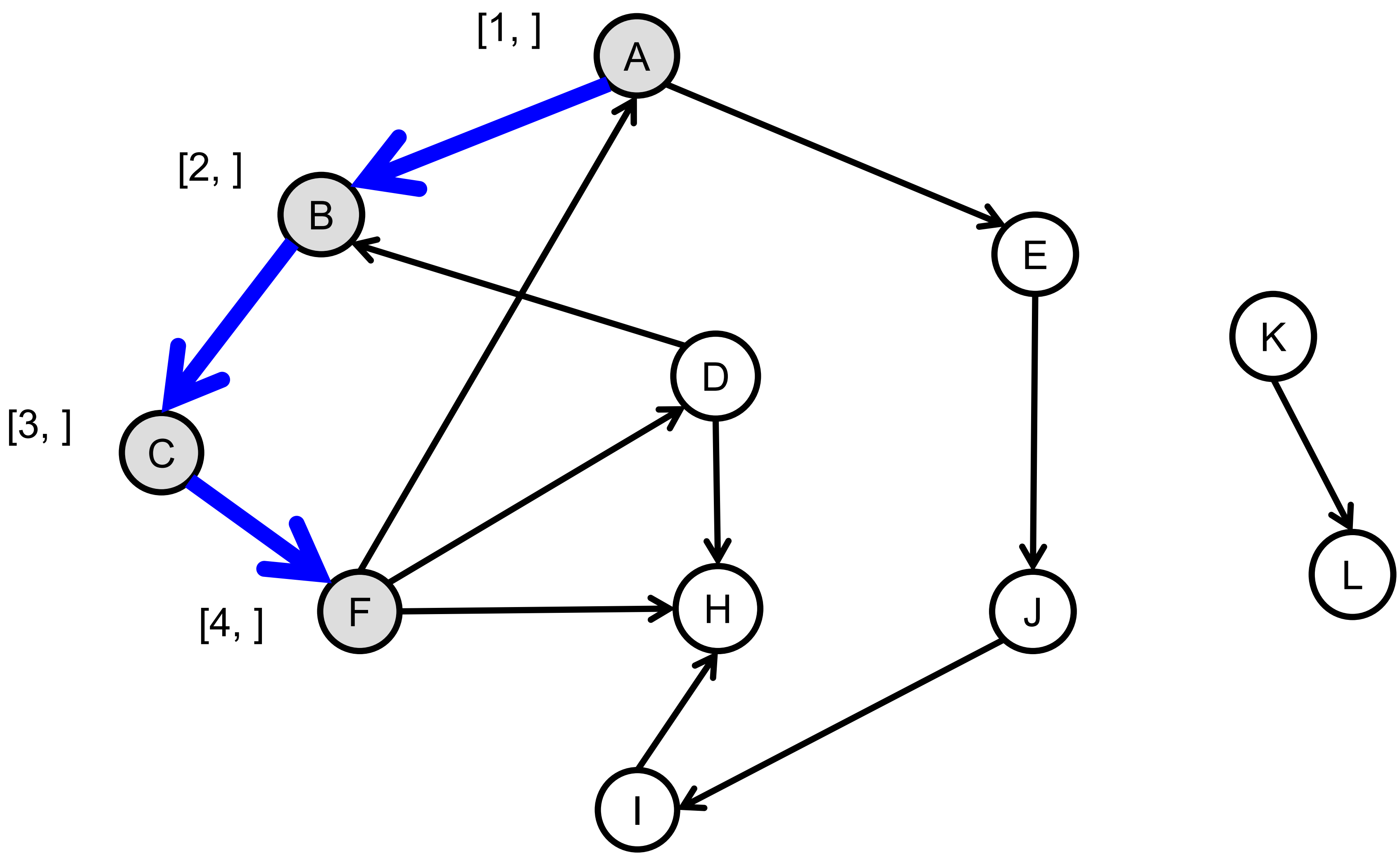
Esempio



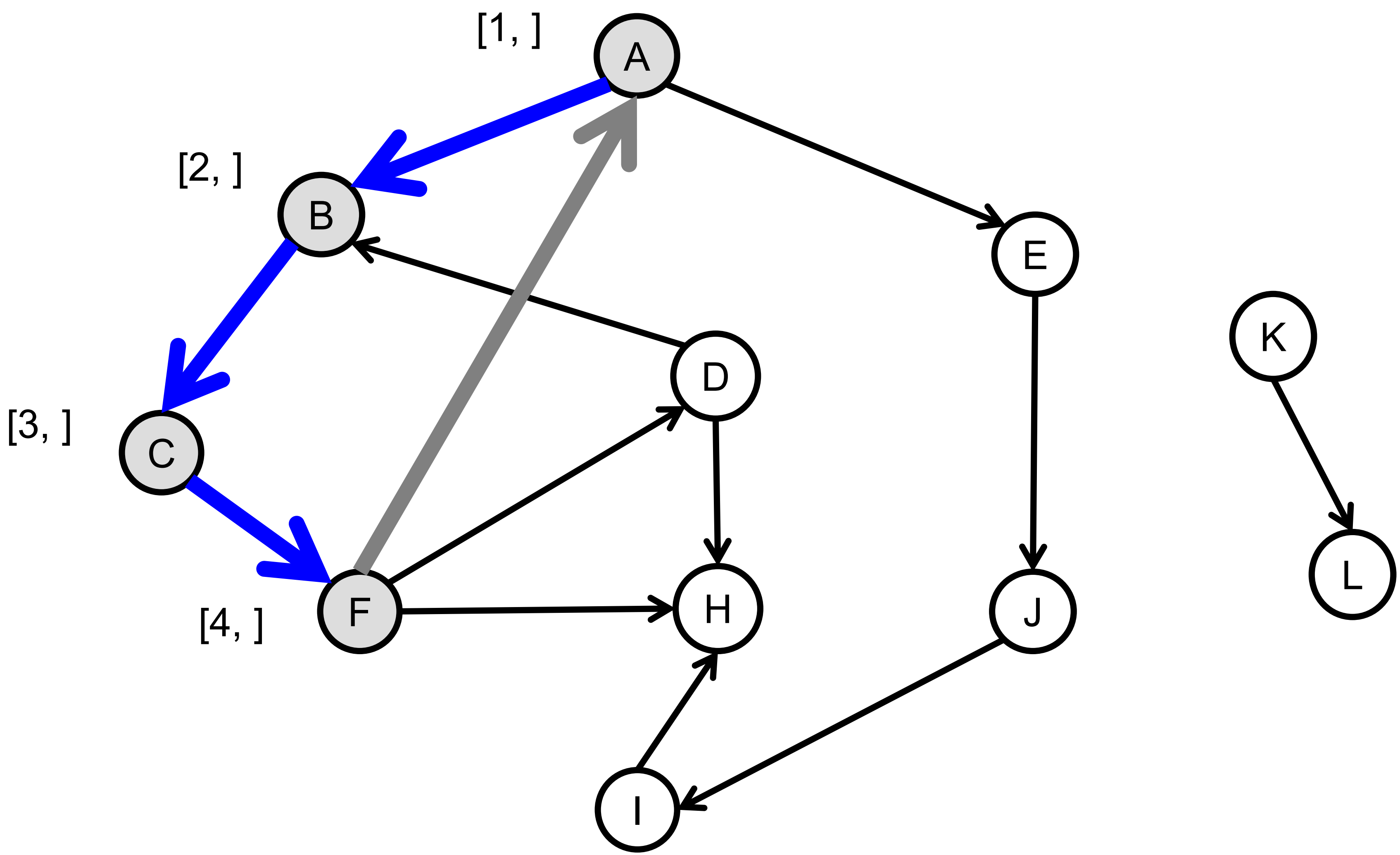
Esempio



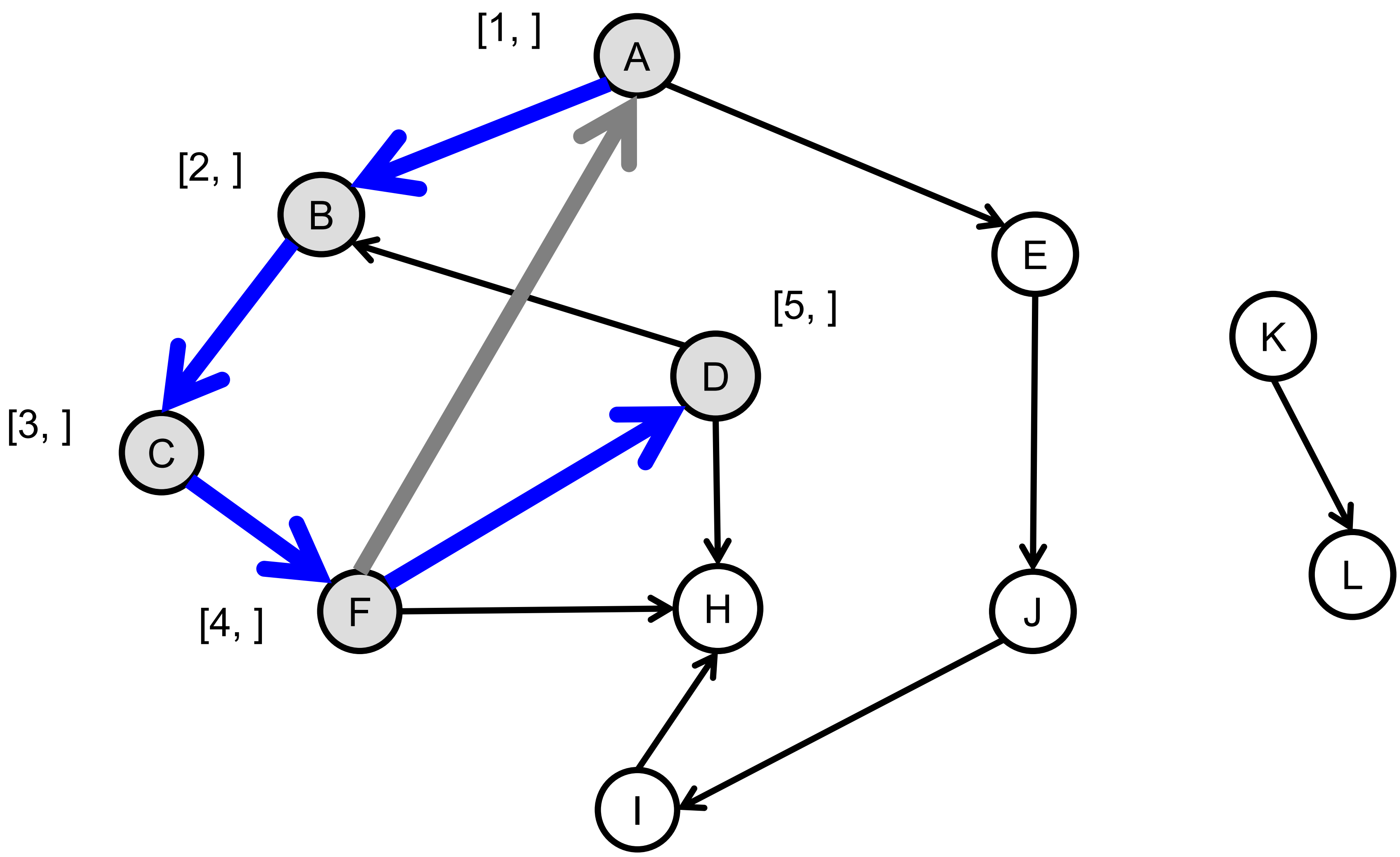
Esempio



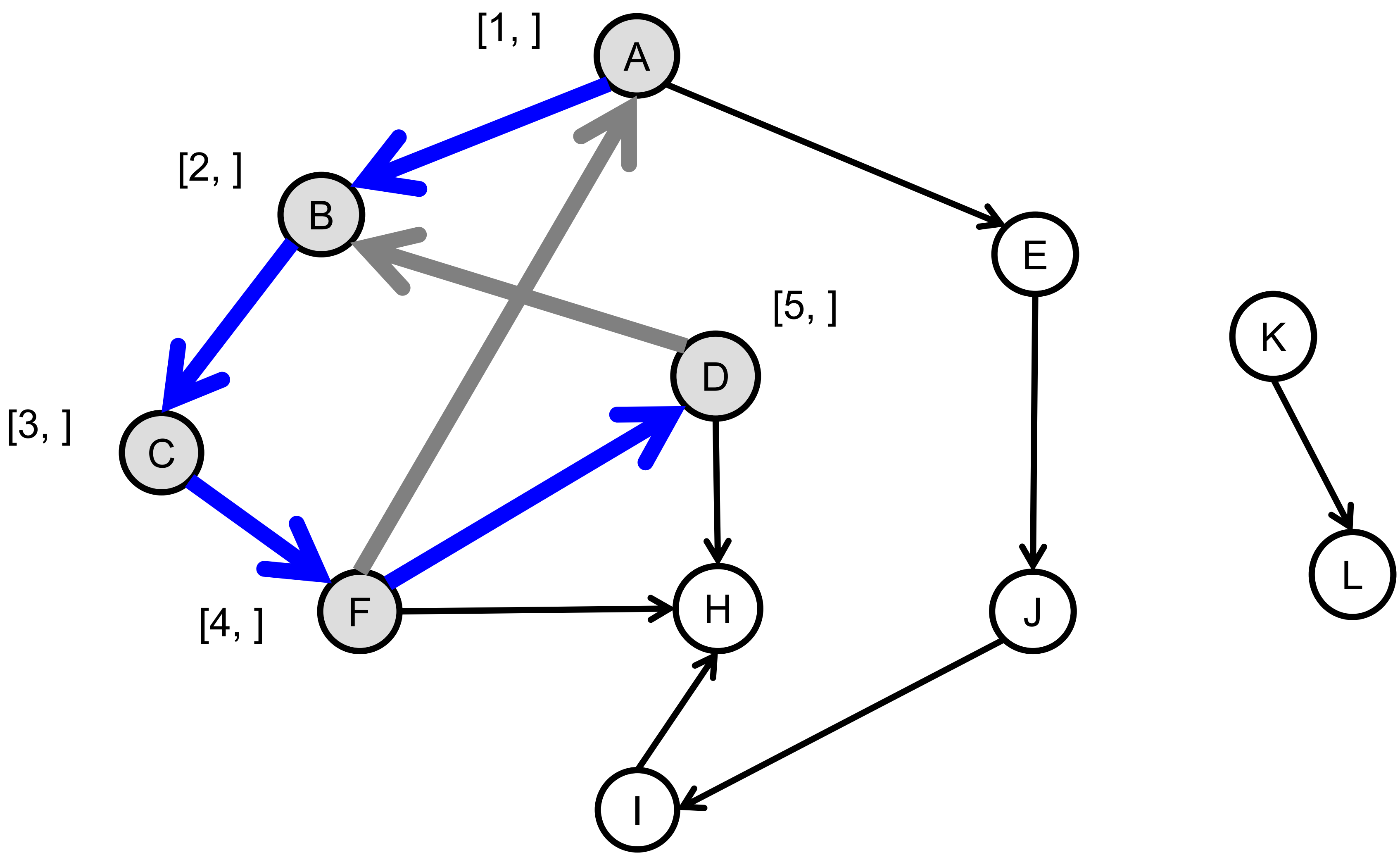
Esempio



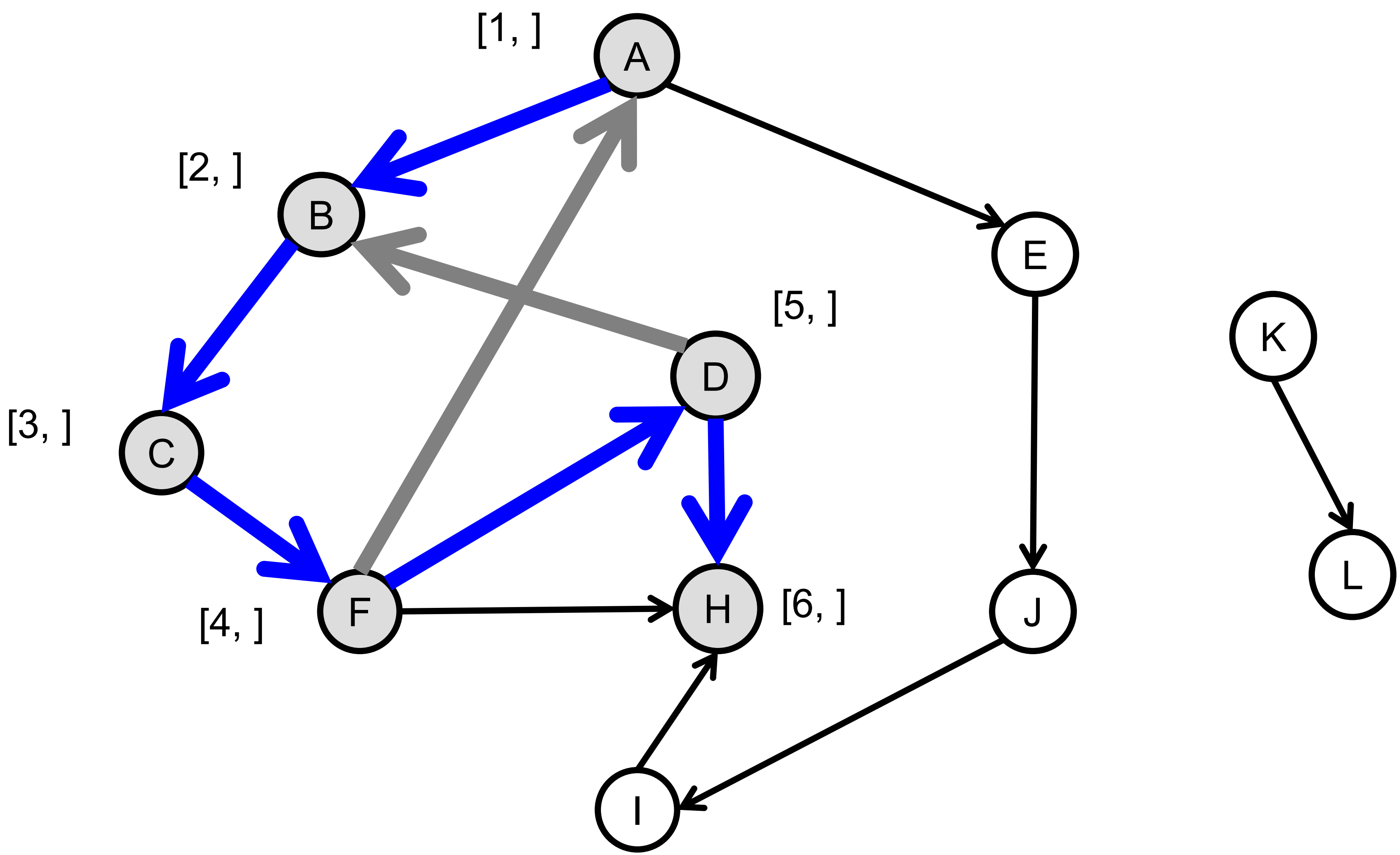
Esempio



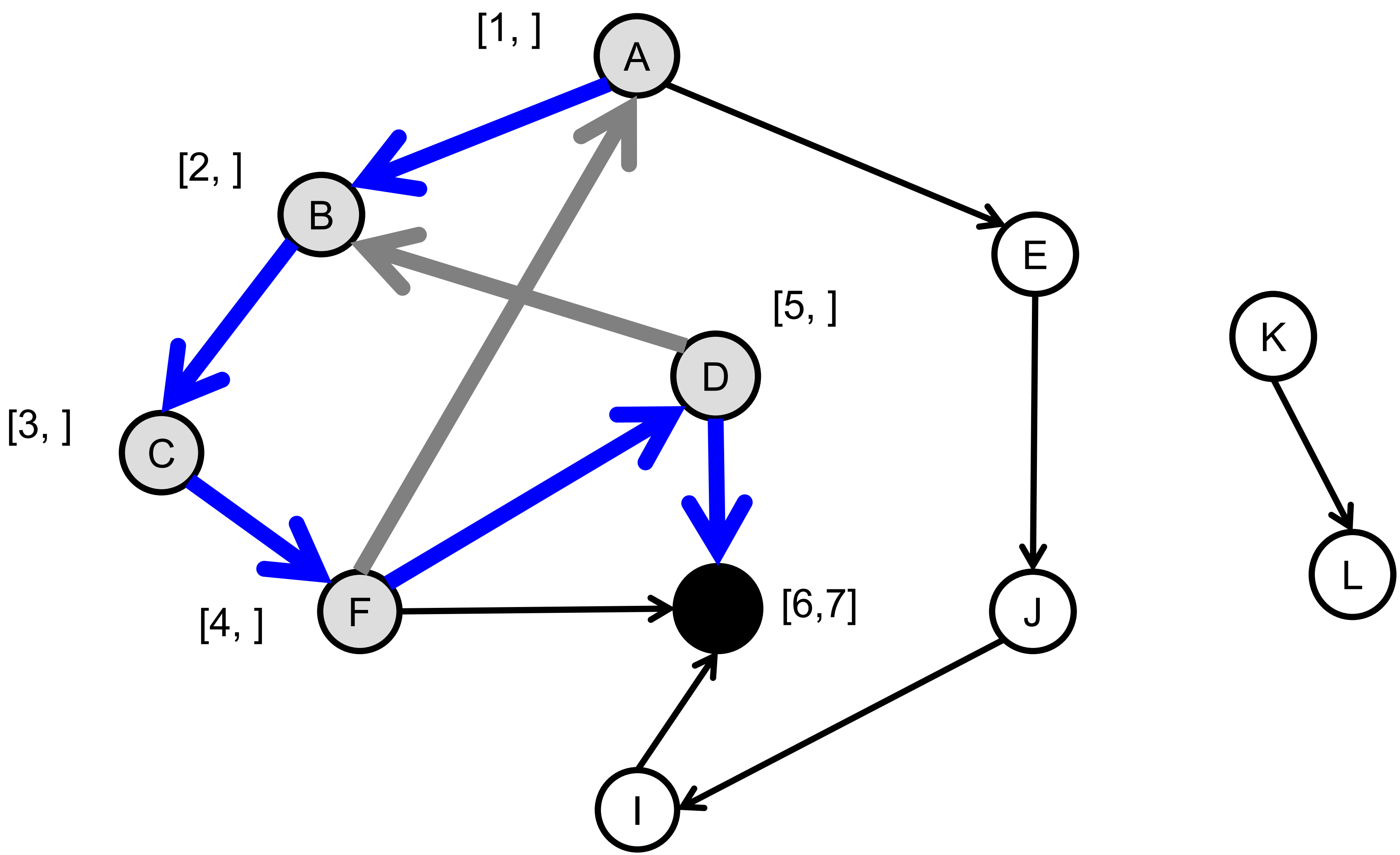
Esempio



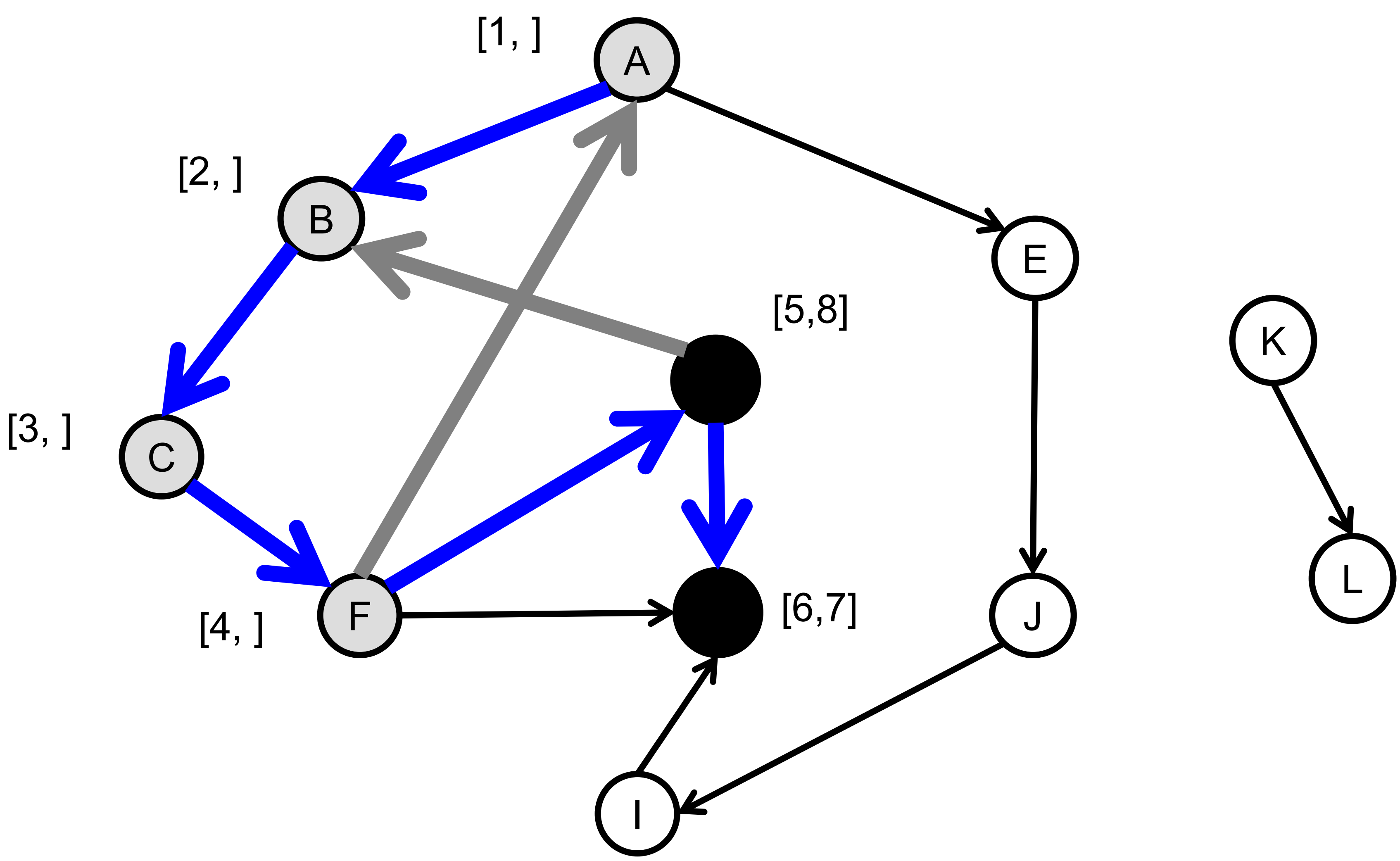
Esempio



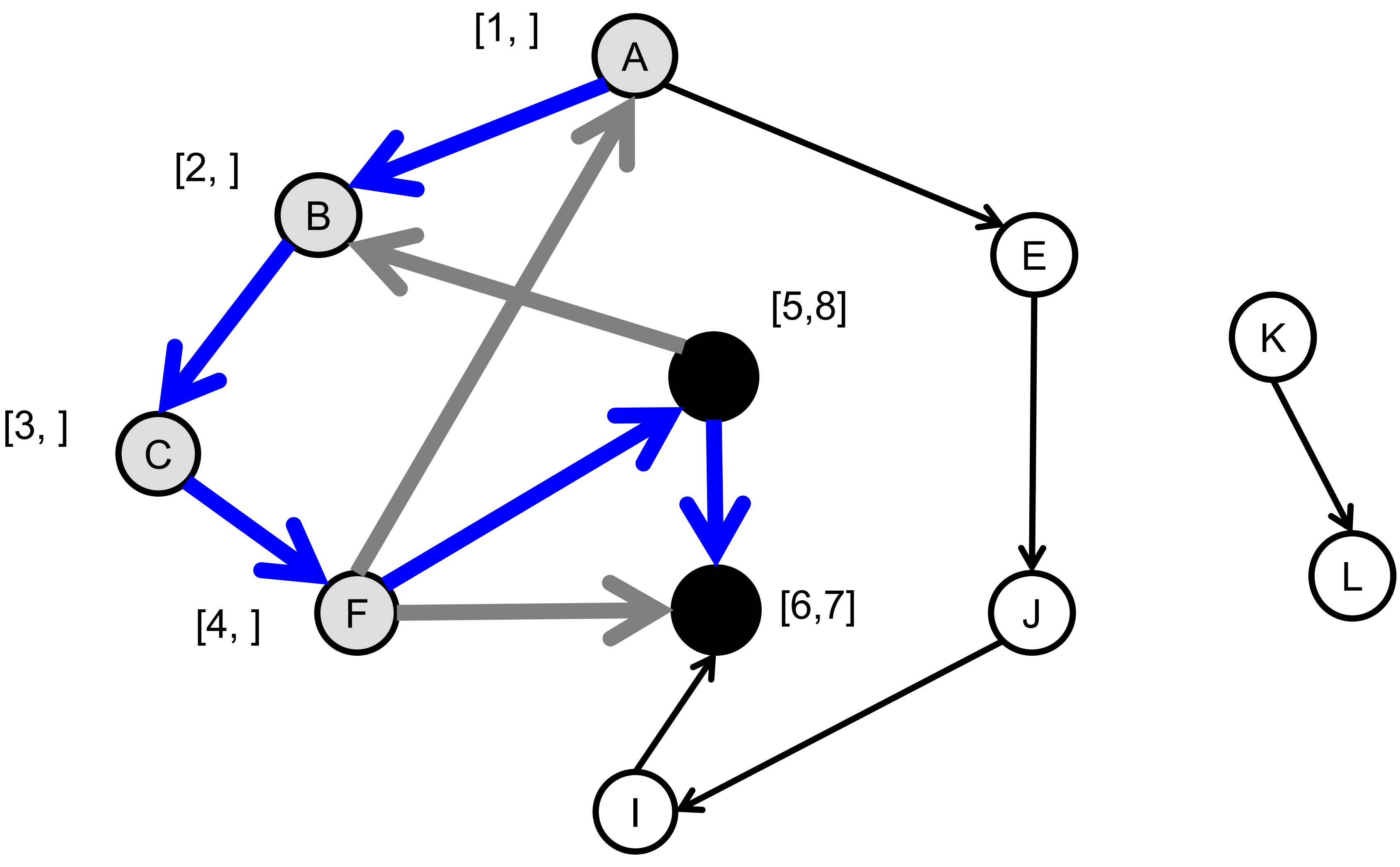
Esempio



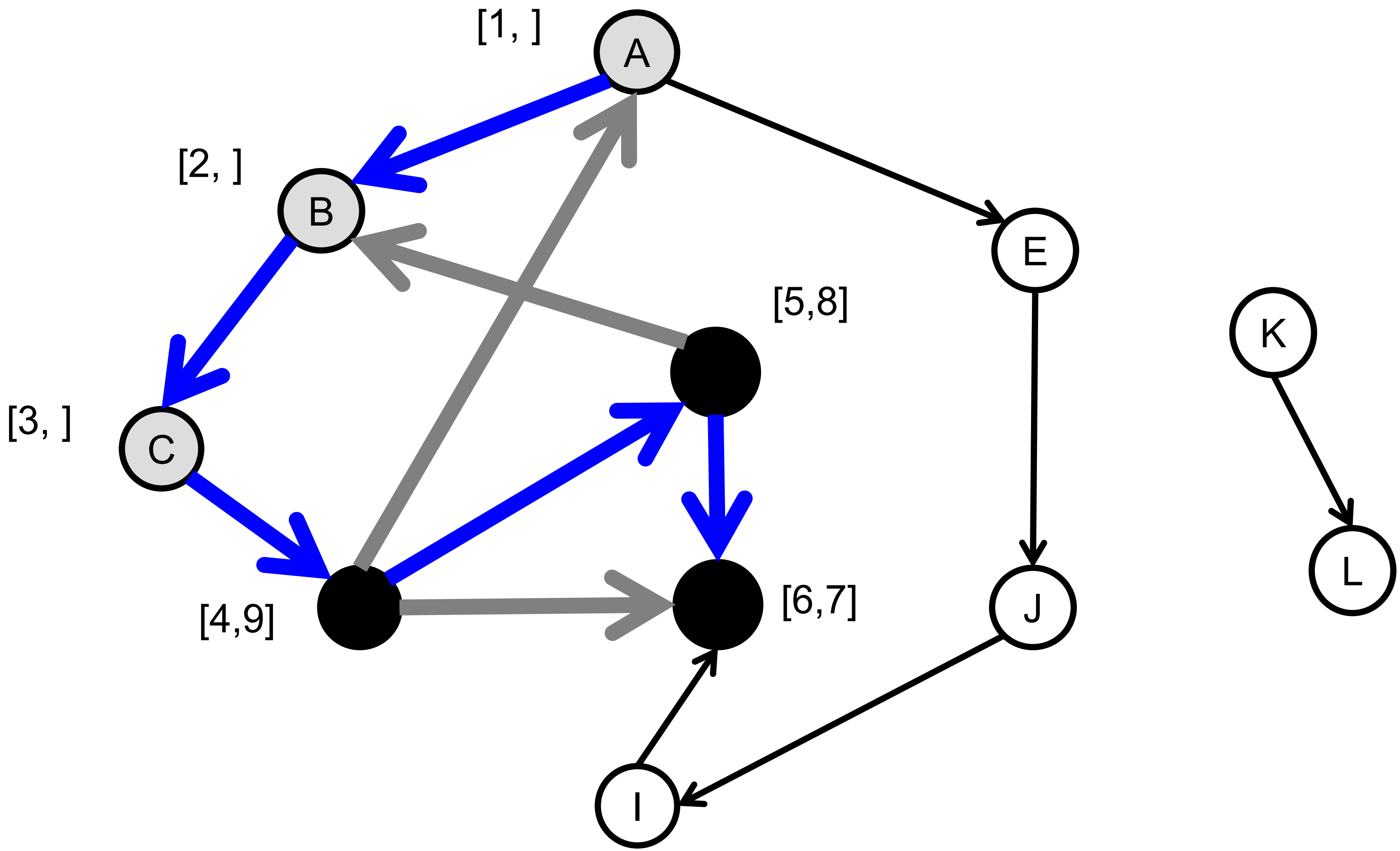
Esempio



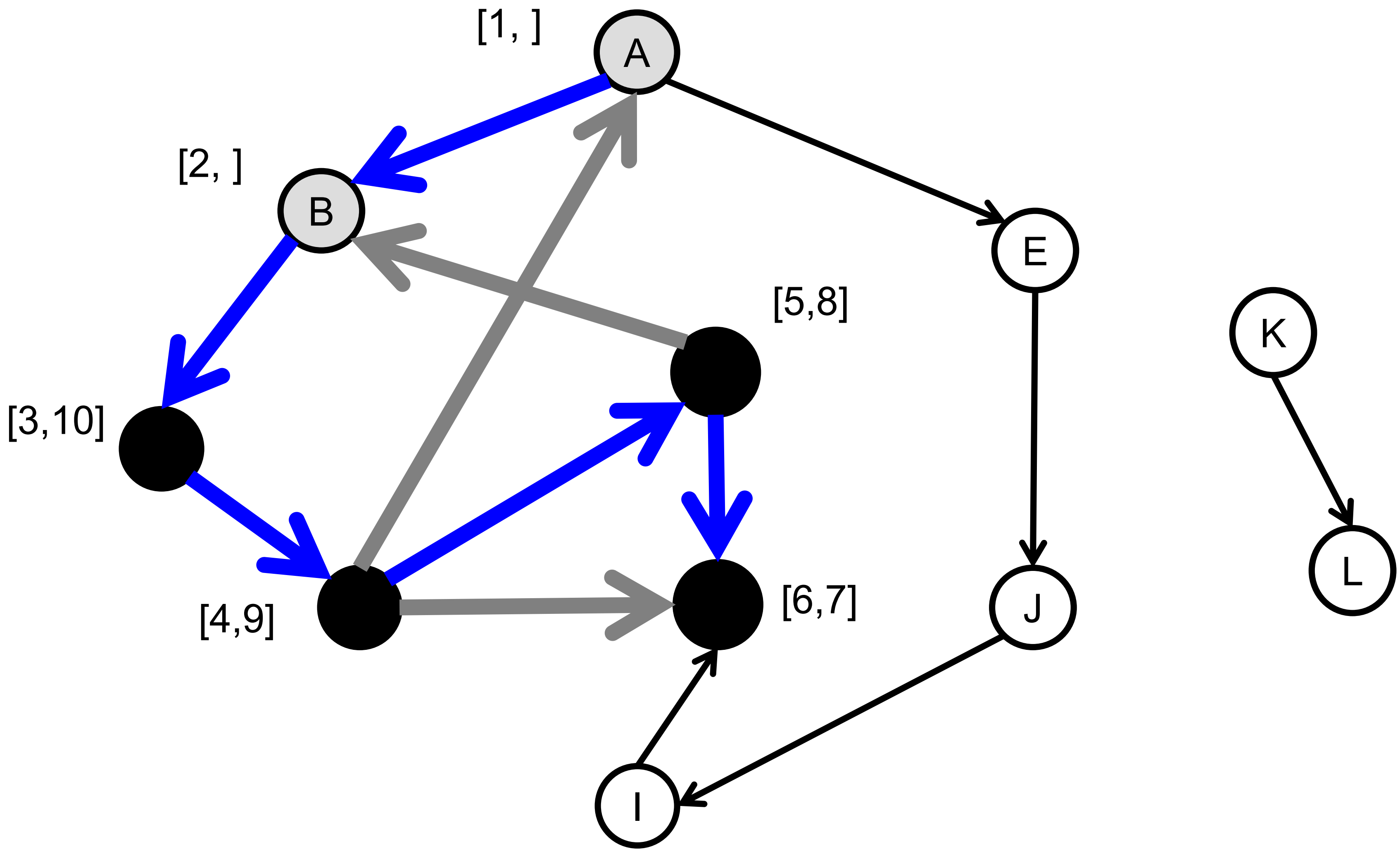
Esempio



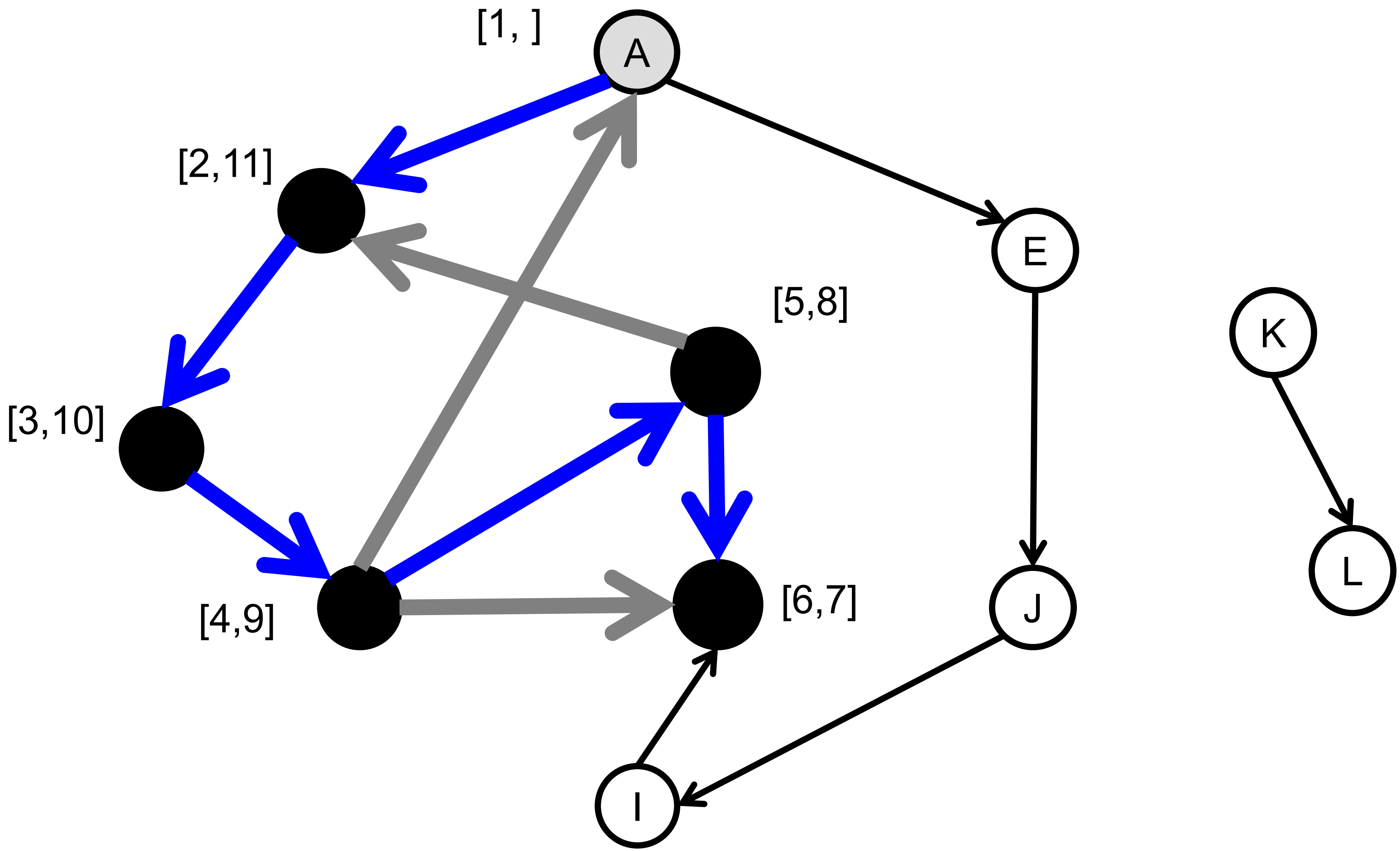
Esempio



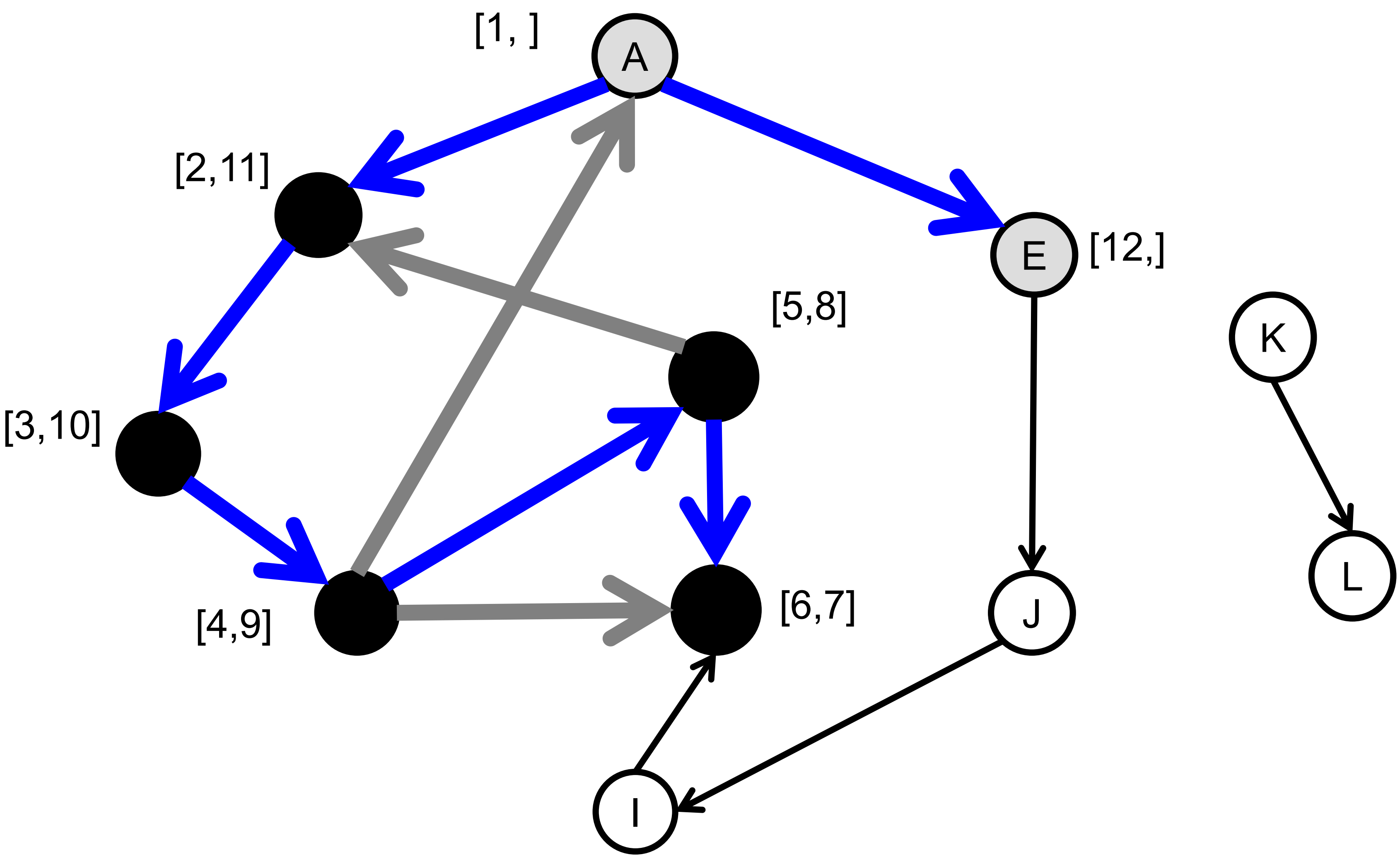
Esempio



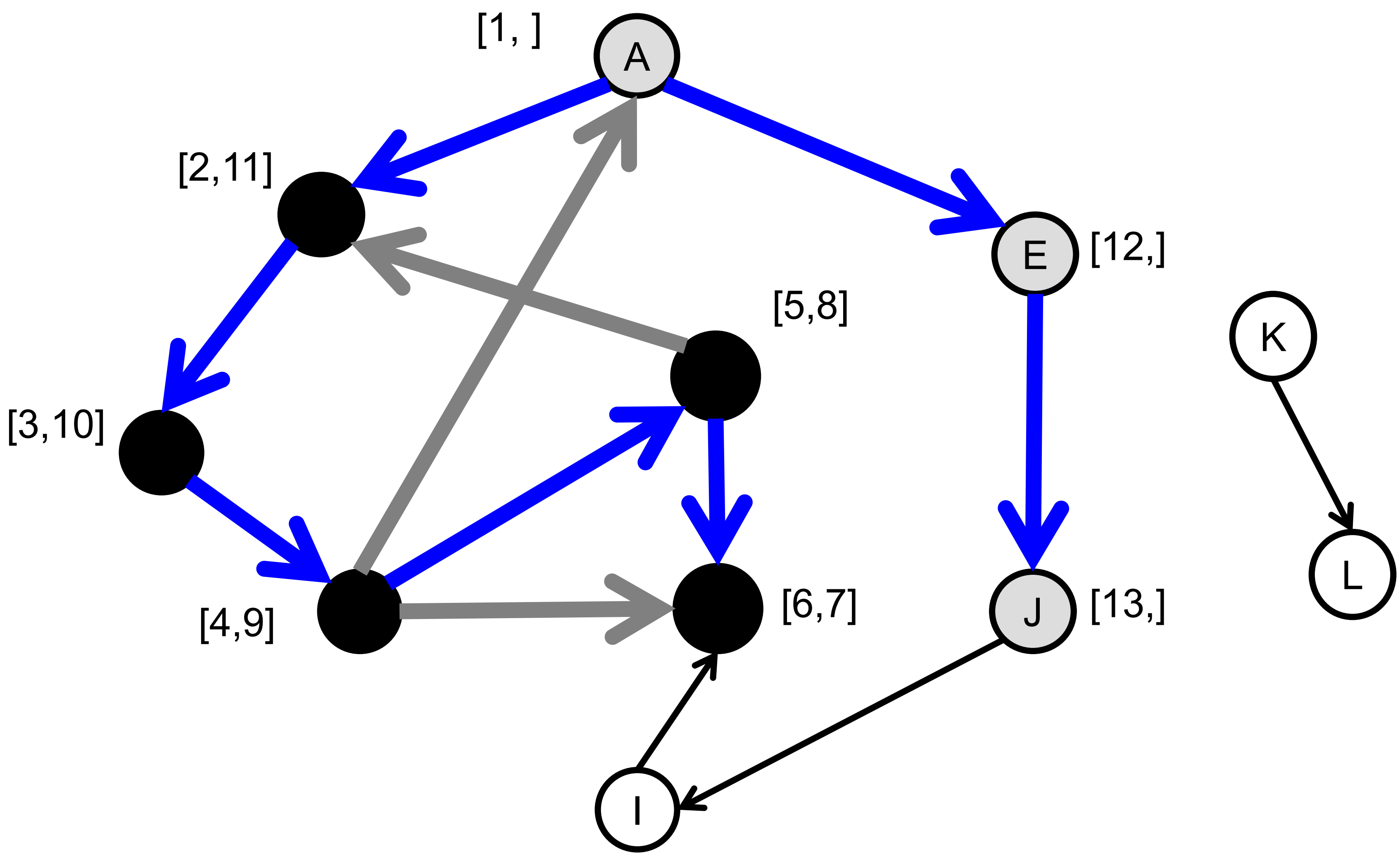
Esempio



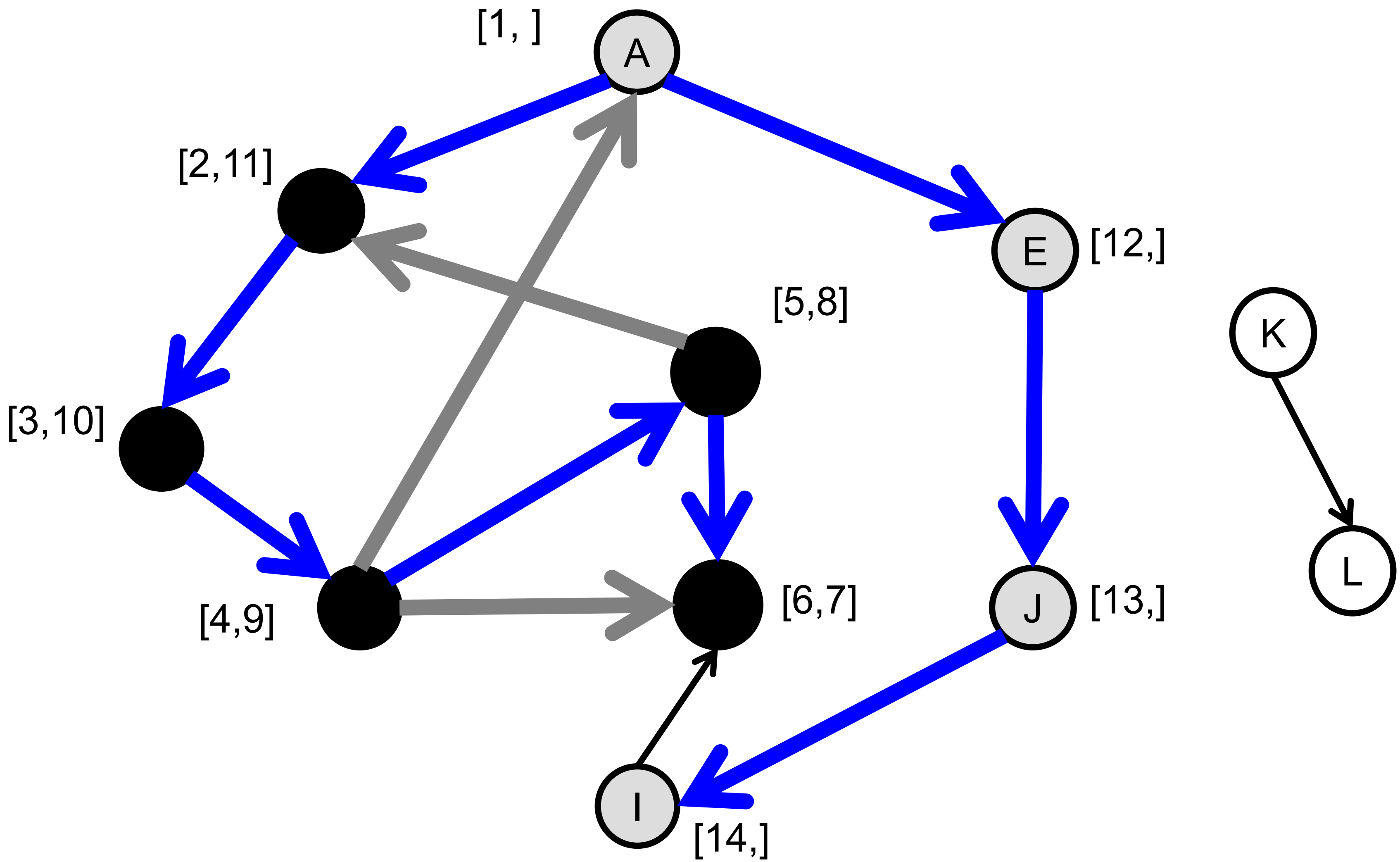
Esempio



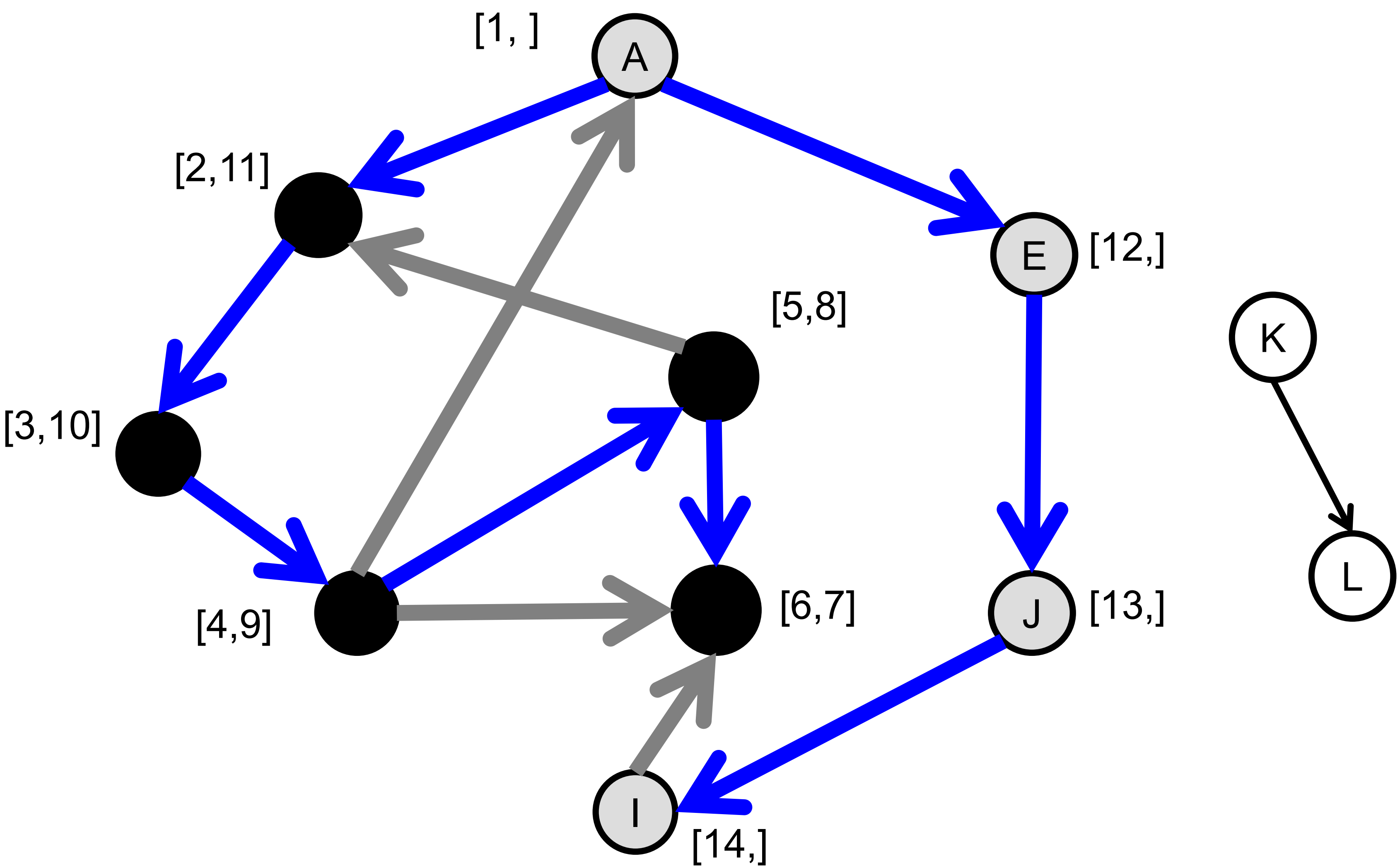
Esempio



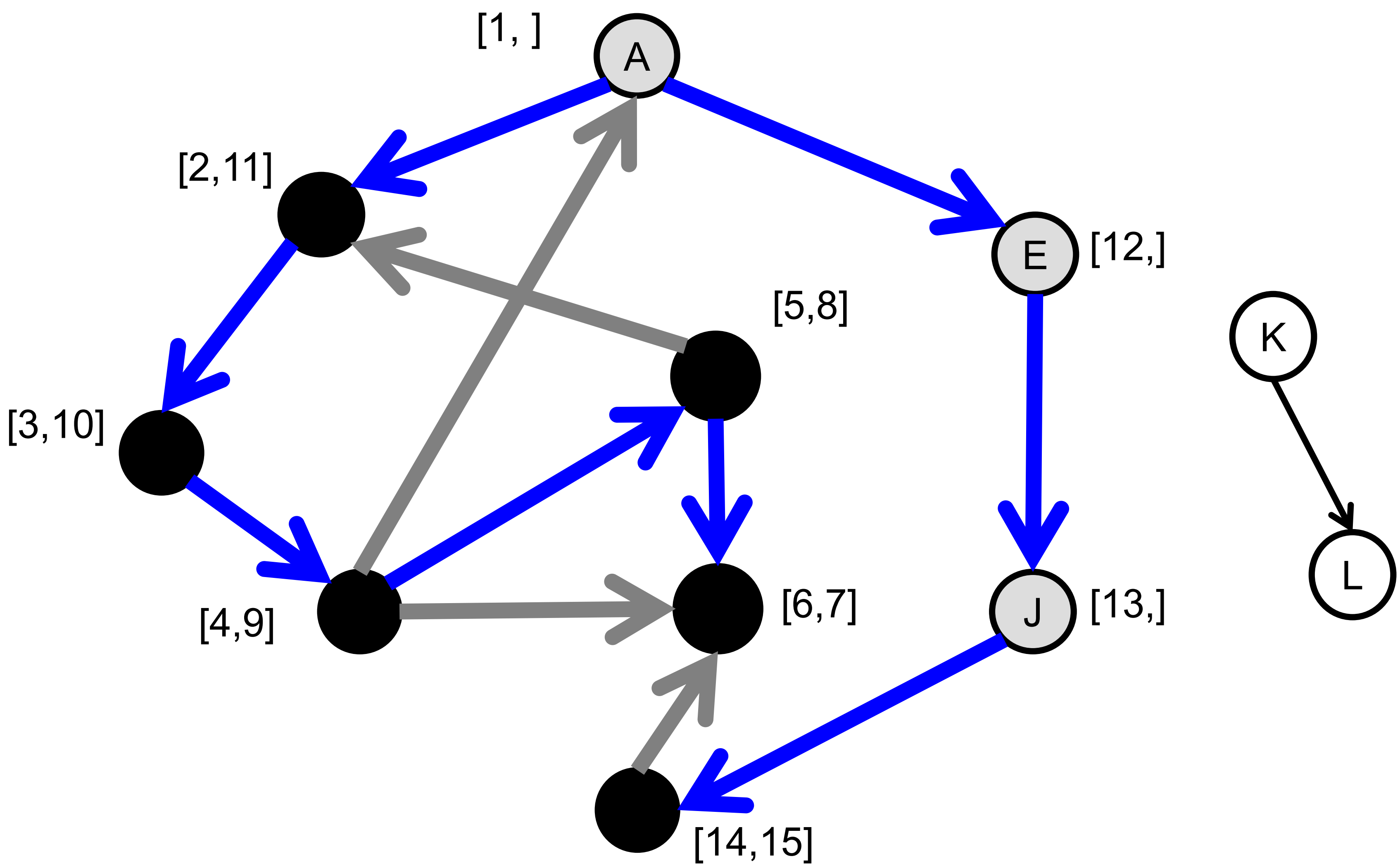
Esempio



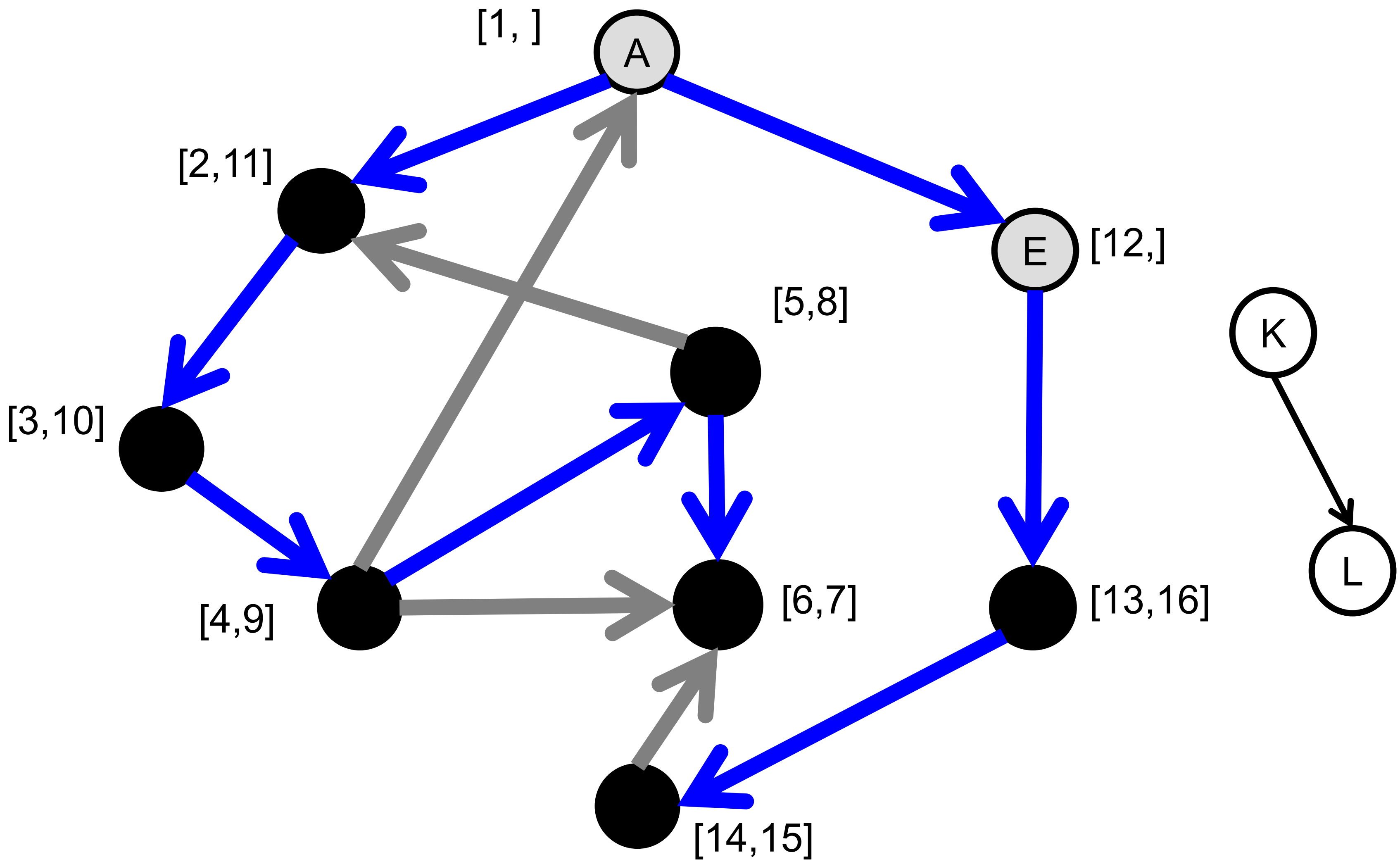
Esempio



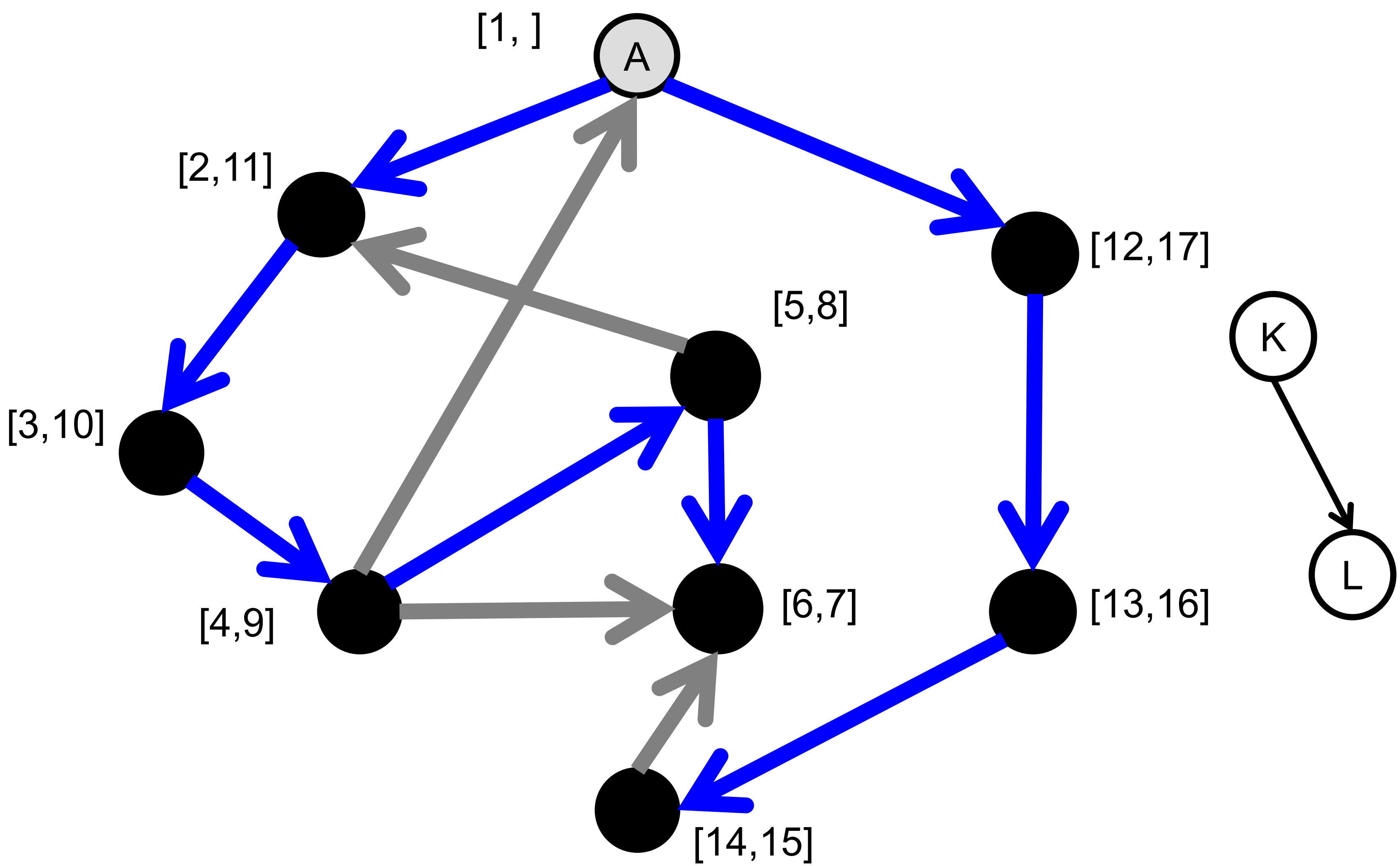
Esempio



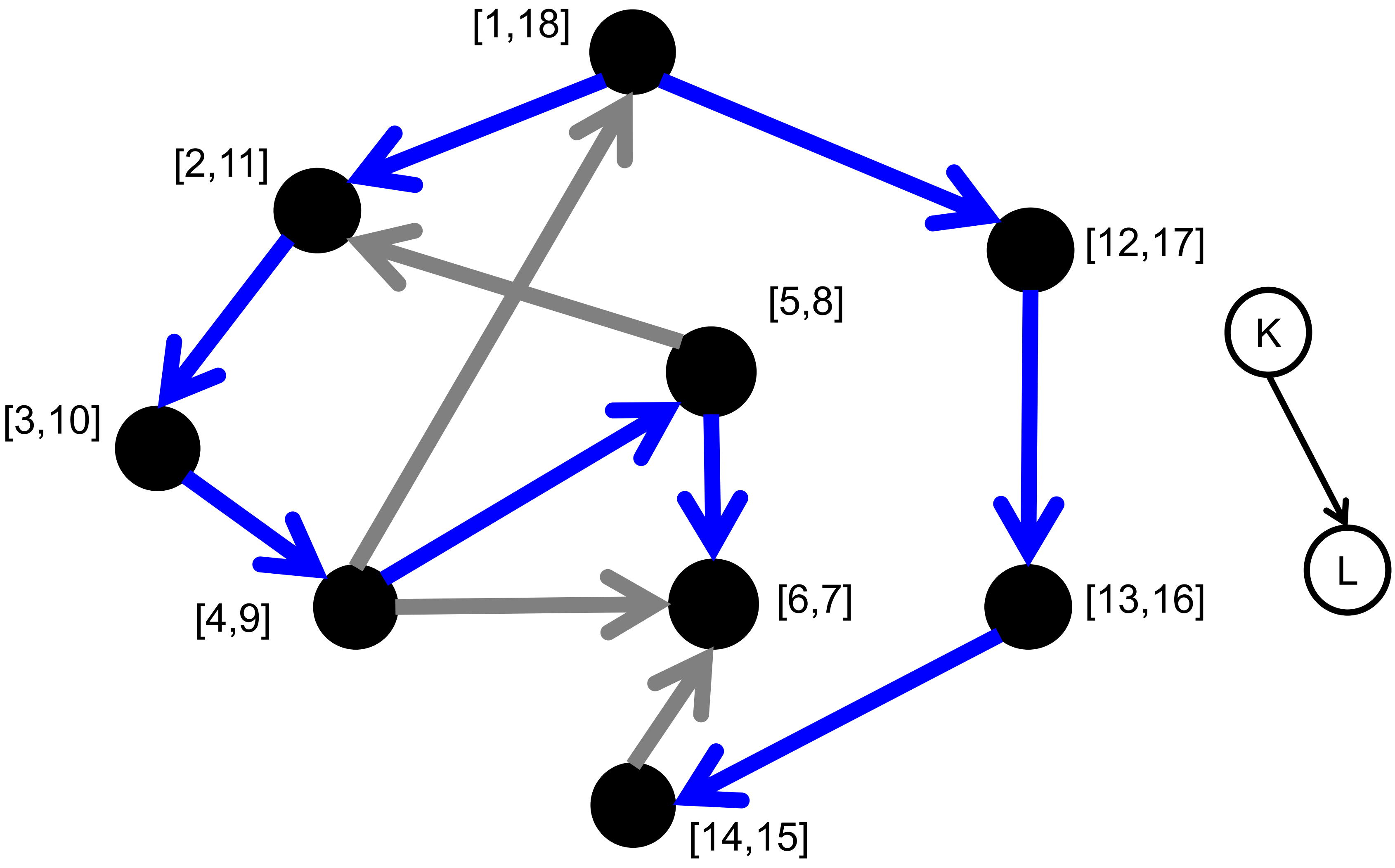
Esempio



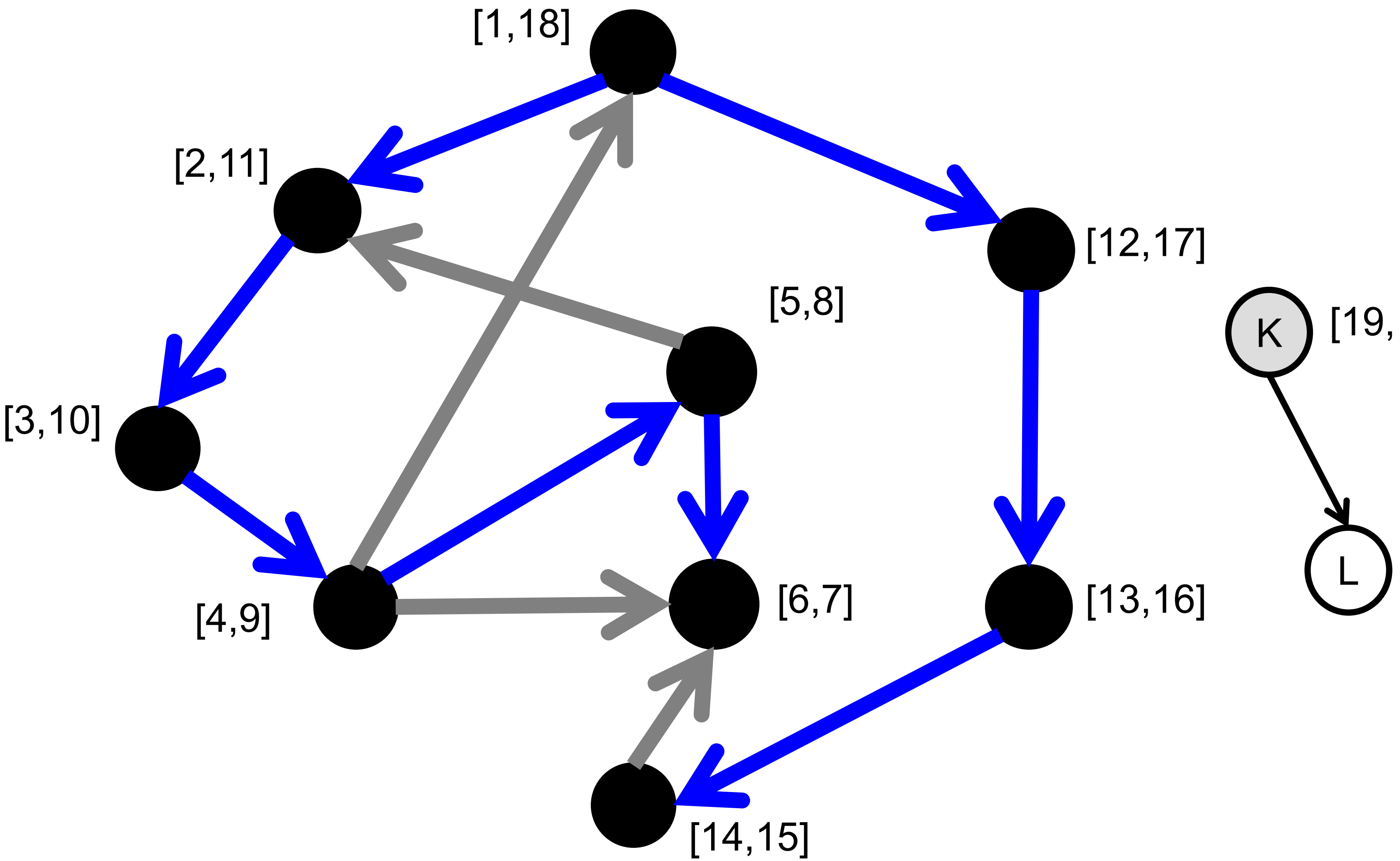
Esempio



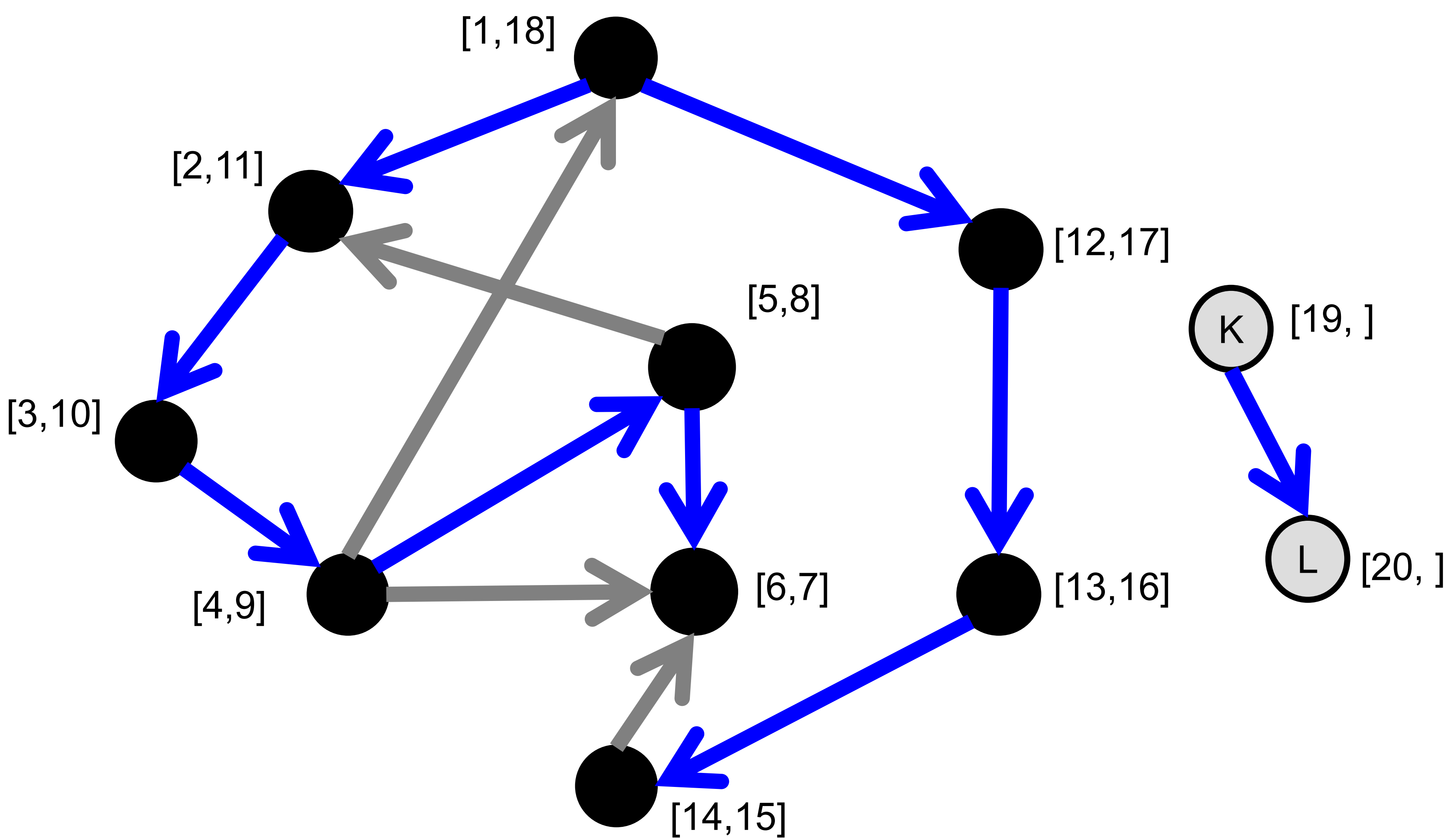
Esempio



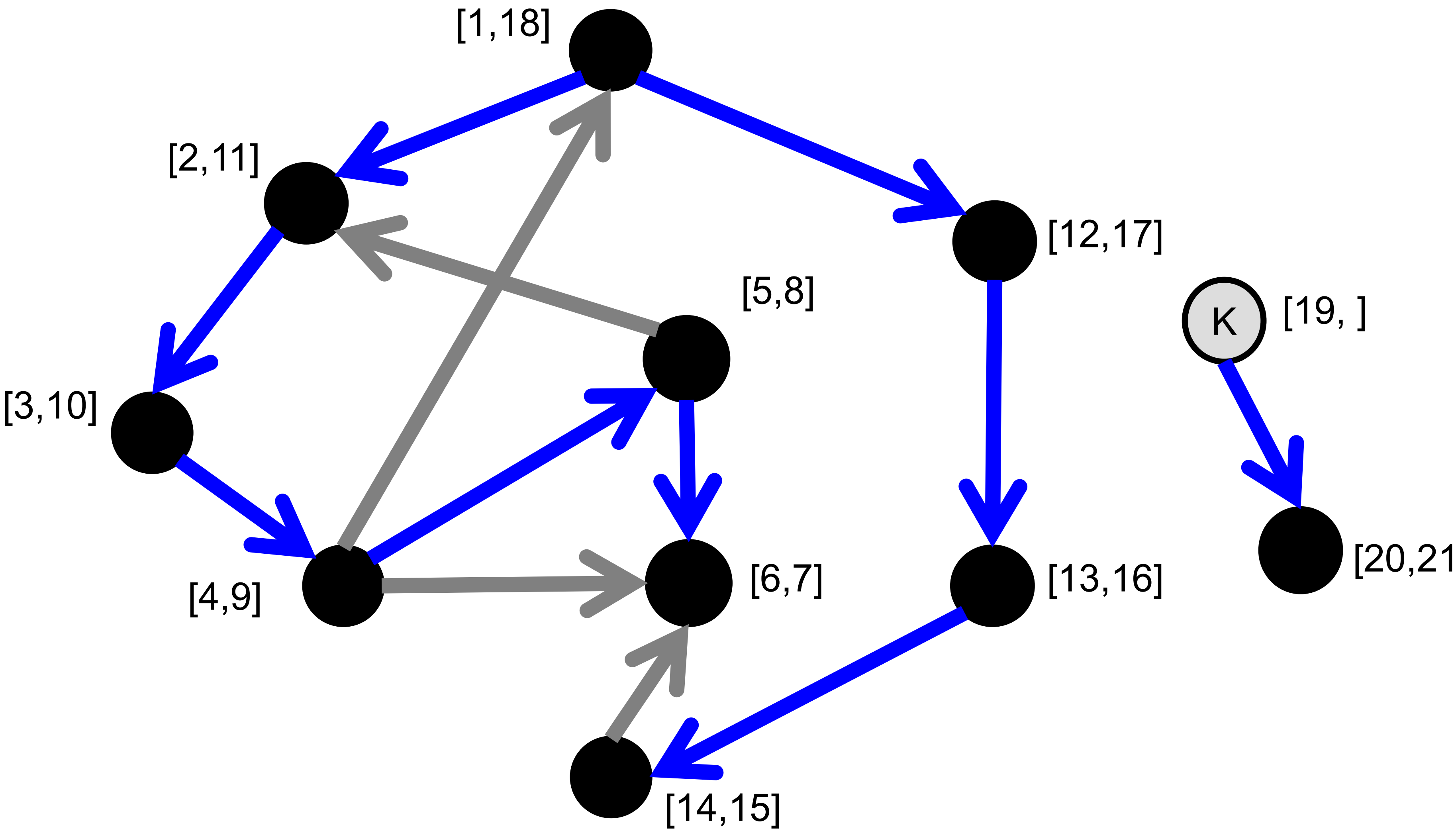
Esempio



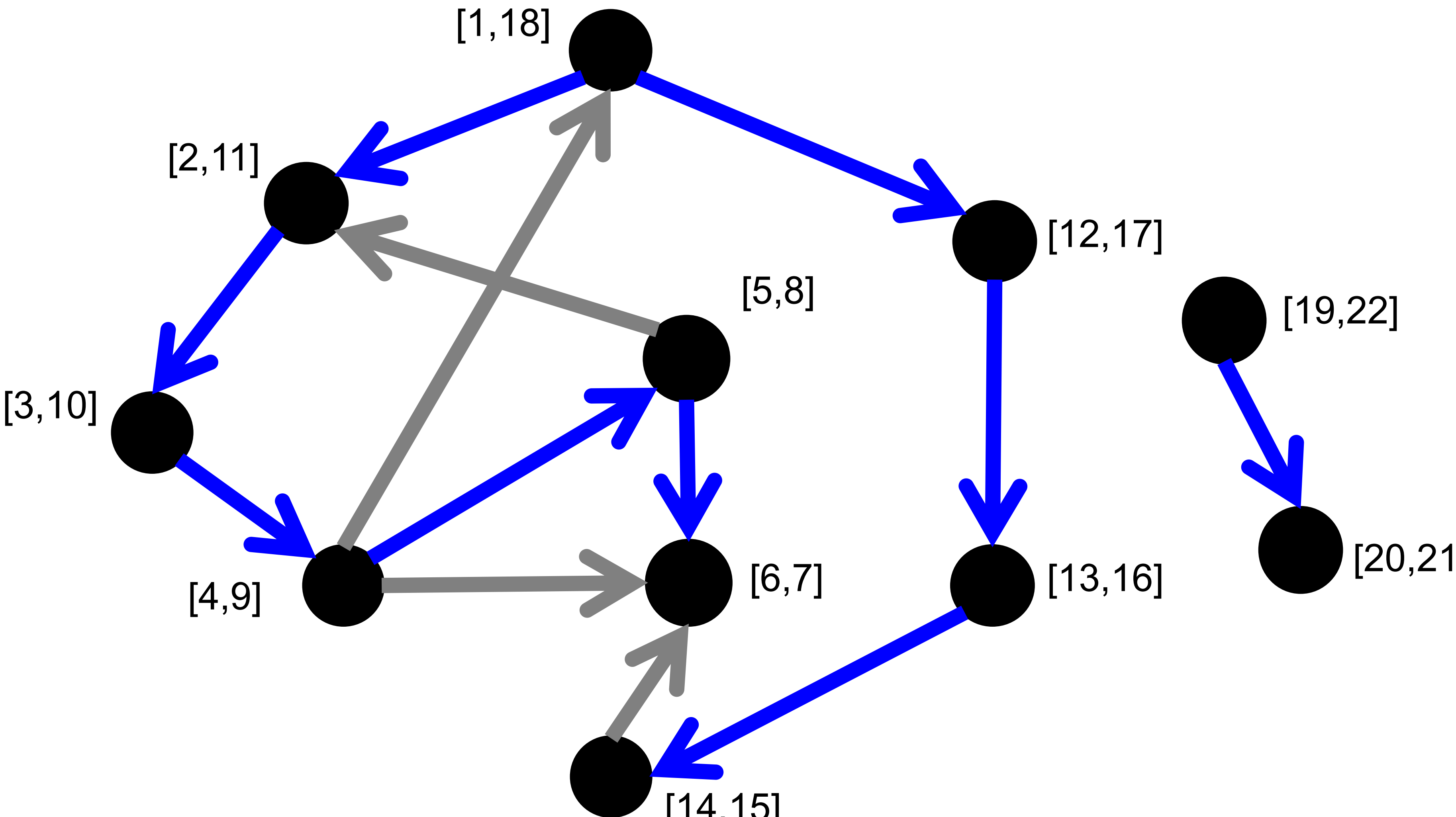
Esempio



Esempio



Esempio

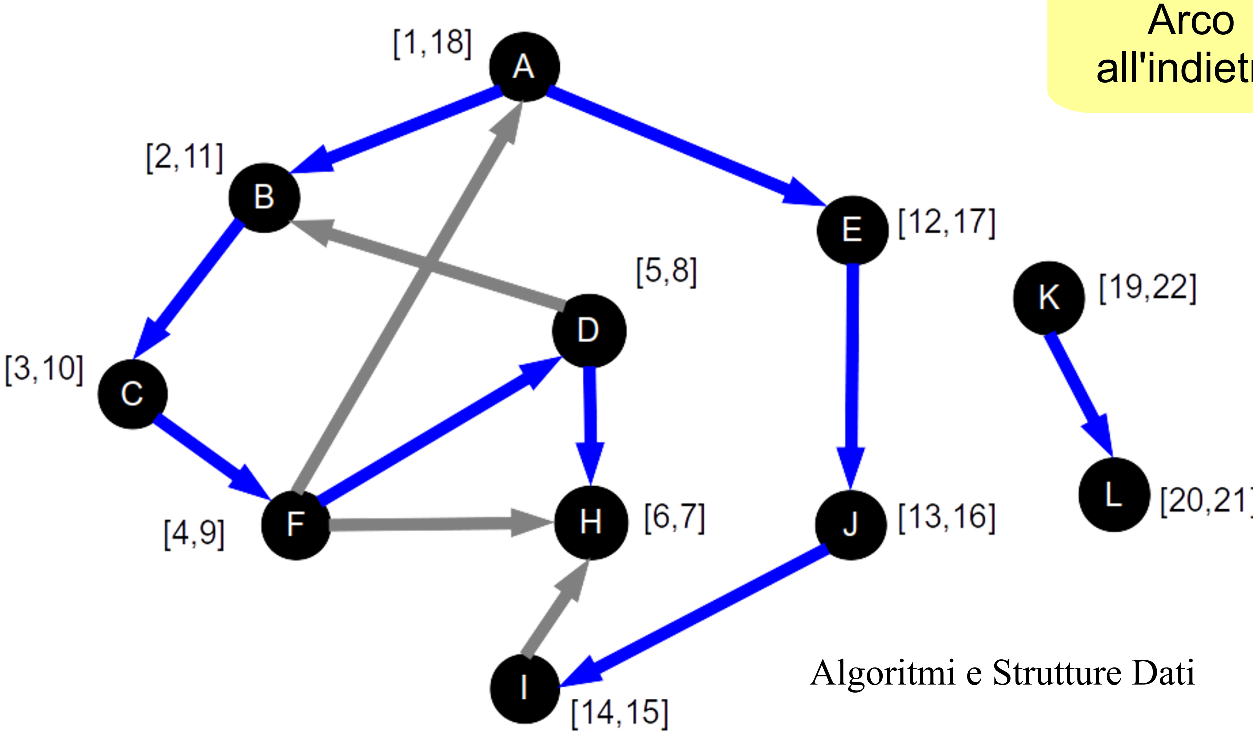


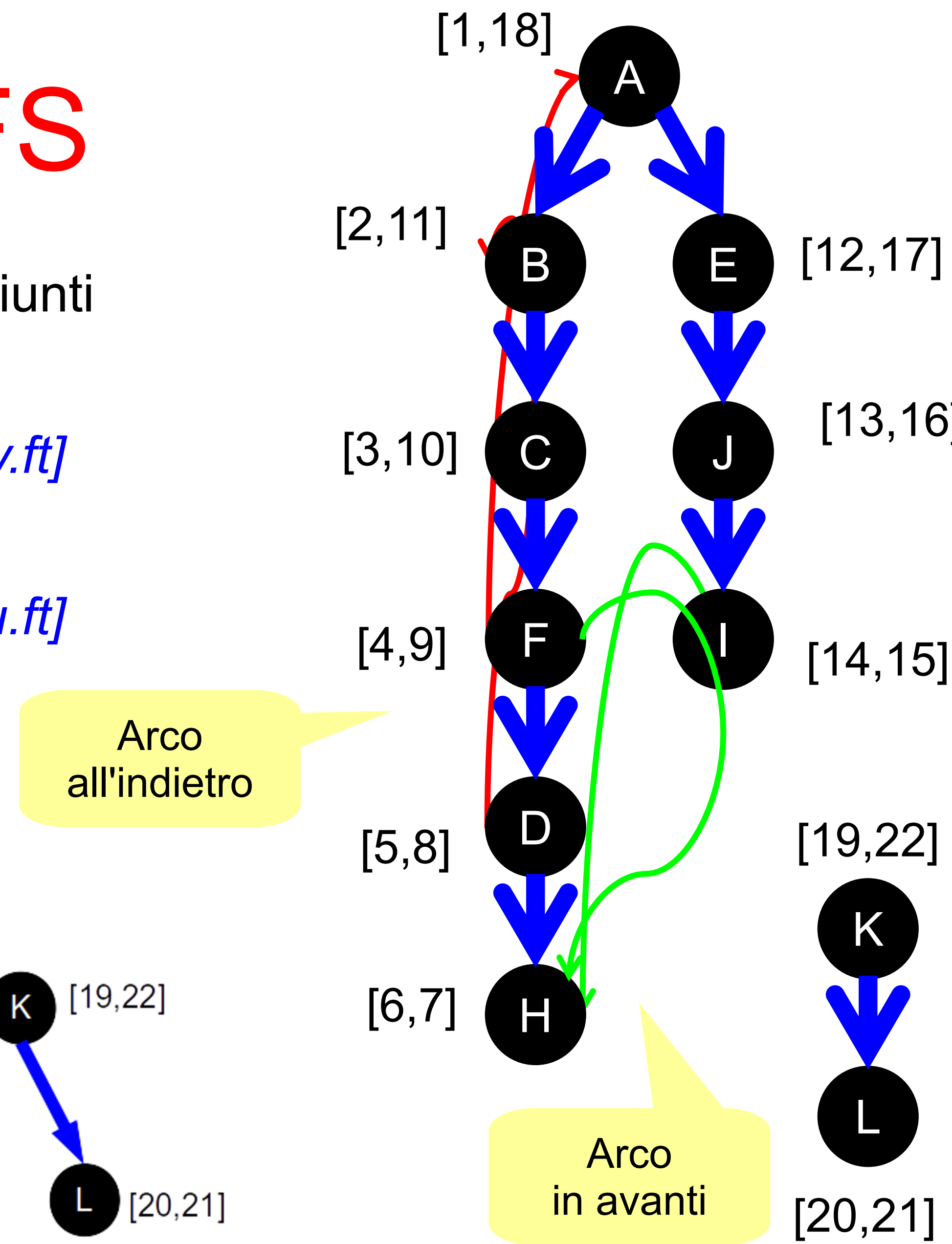
Proprietà della visita DFS
Teorema delle parentesi

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

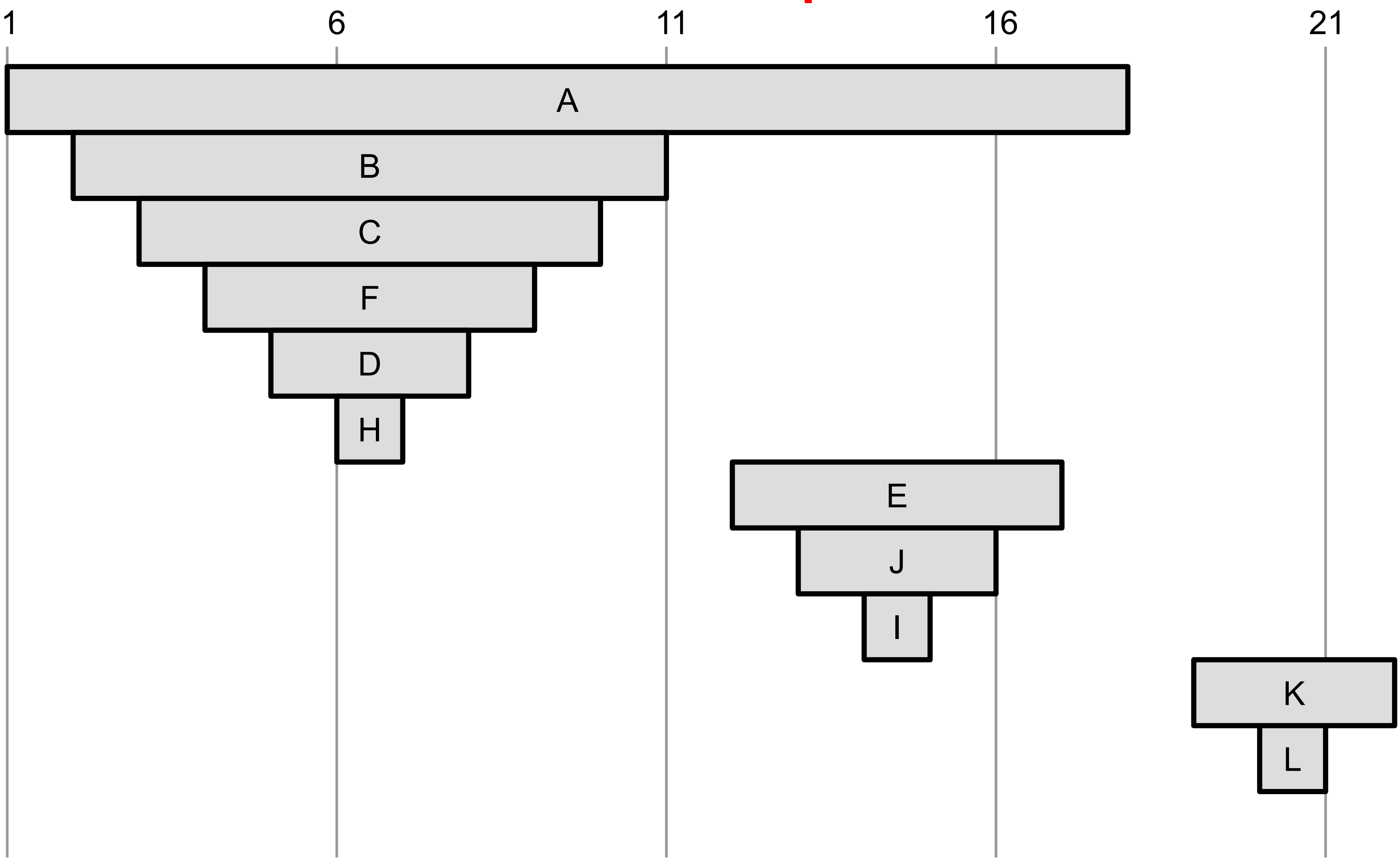
Foresta DFS

- Gli intervalli *[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

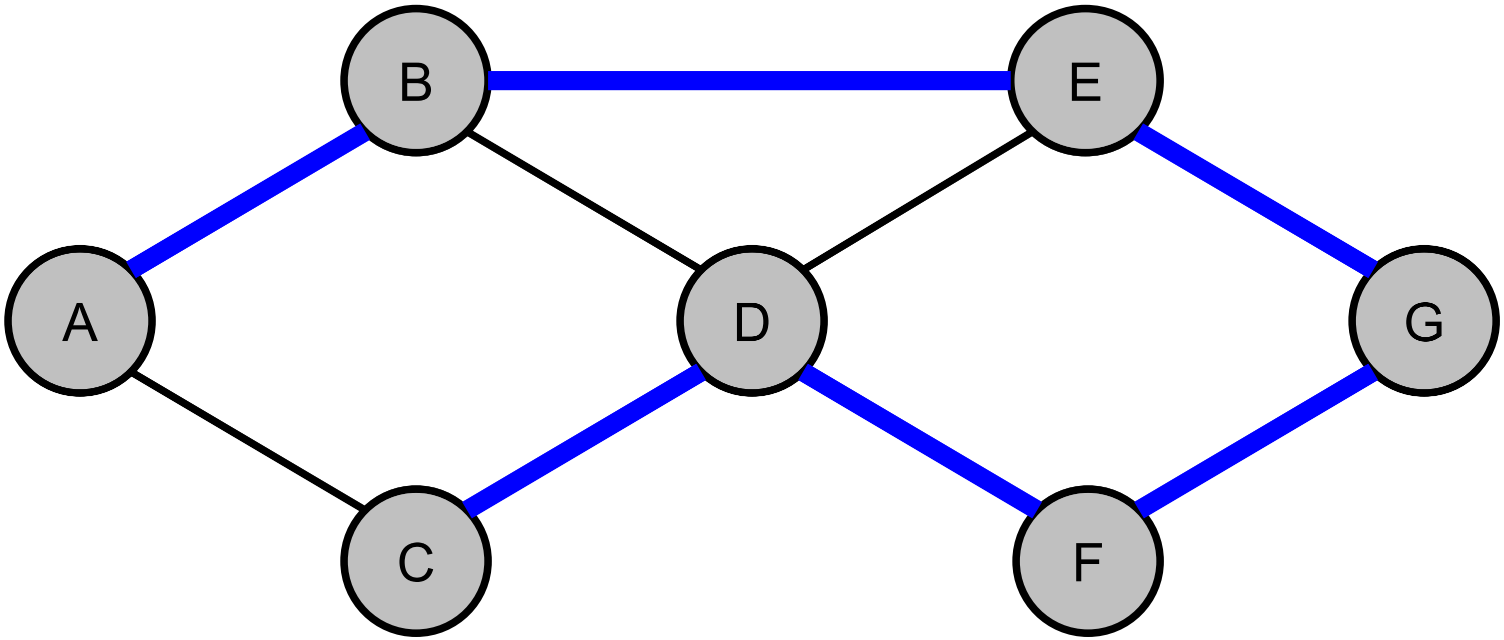




Teorema delle parentesi



Esercizio



- Il sottografo blu può essere prodotto da una visita in ampiezza? Se sì indicare un possibile nodo di inizio e come possono essere rappresentate le liste di adiacenza
- Ripetere l'esercizio con una visita in profondità

Applicazioni degli algoritmi di visita

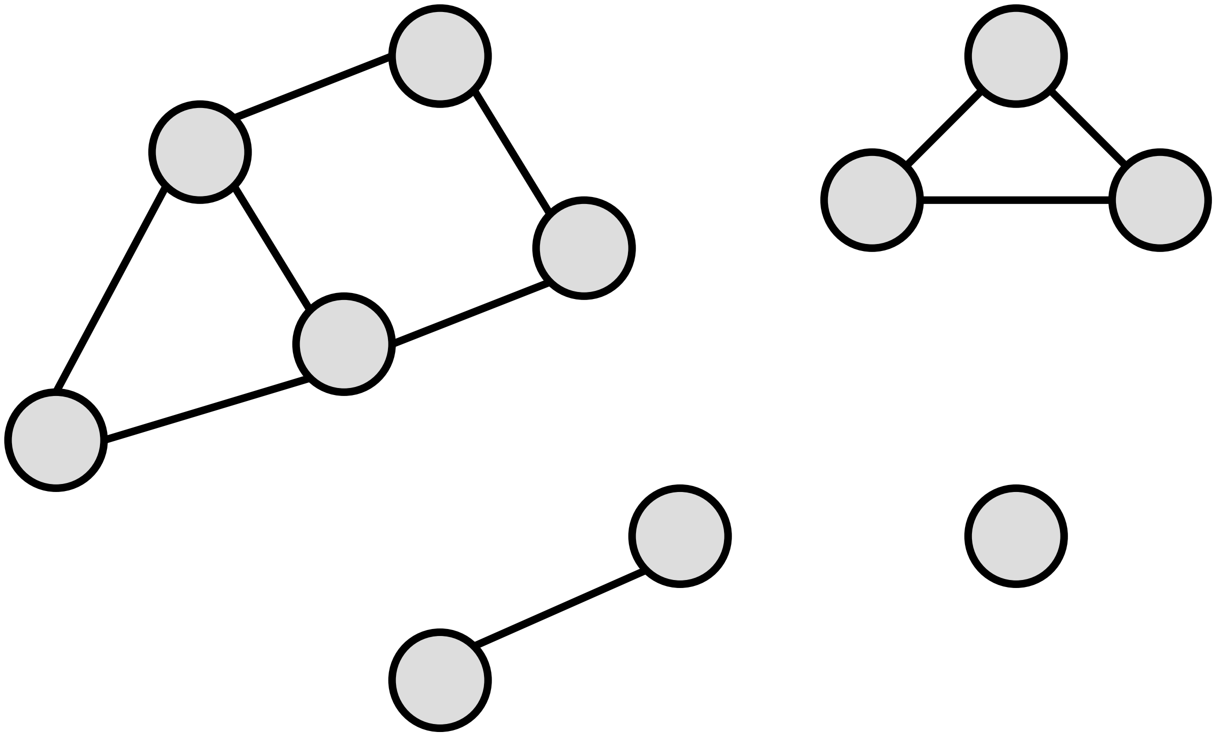
- Decidere se un grafo orientato ha un ciclo
  - Sì se e solo se l'albero DFS ha almeno un arco all'indietro
  - in altri termini, sì se durante la visita DFS incontro un nodo grigio

- Individuare le componenti connesse di un grafo non orientato

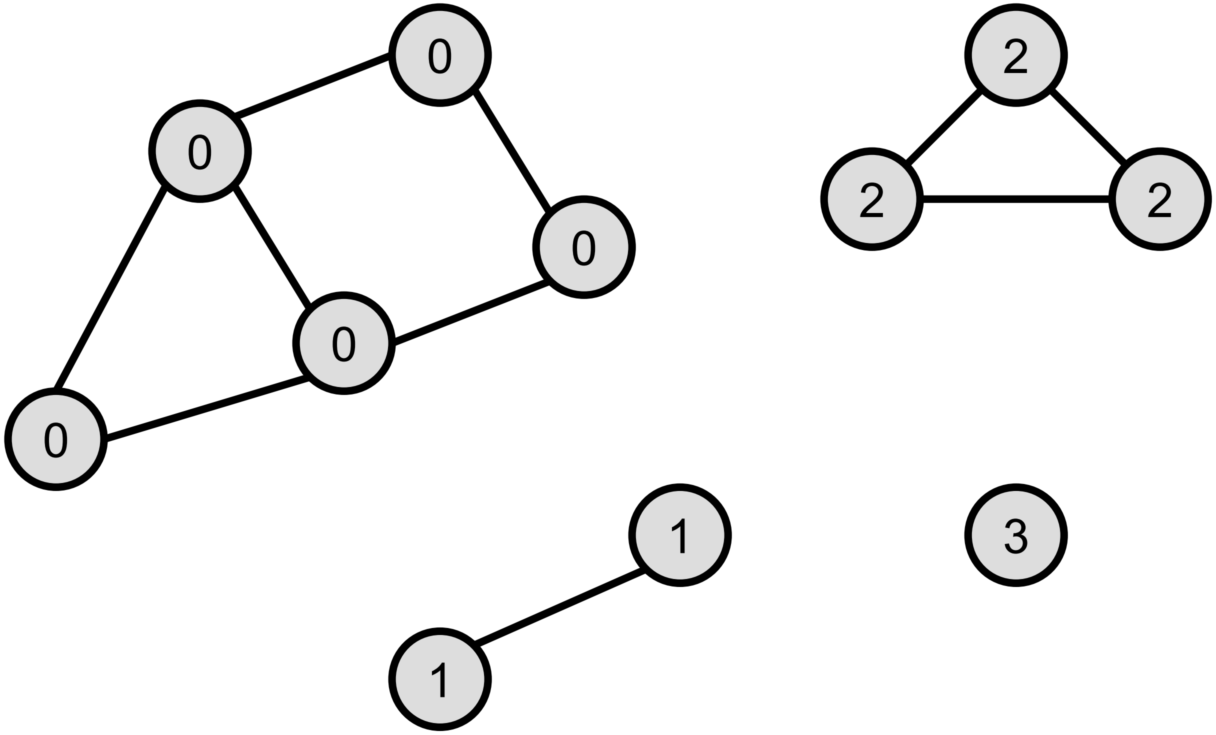
- Individuare le componenti fortemente connesse di un grafo orientato

Componenti Connesse

Componenti Connesse



Componenti Connesse



Componenti Connesse

- Dato un grafo non orientato $G = (V, E)$ composto da K componenti connesse, etichettare ogni nodo v con un intero v.cc scelto tra $\{0, 1, \ldots K - 1\}$ in modo tale che due nodi abbiano la stessa etichetta se e solo se appartengono alla stessa componente connessa.

Componenti Connesse

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

CC-visit(Grafo G=(V,E), nodo v, integer k)
  v.cc ← k;
  for each u adiacente a v do
    if (u.cc < 0) then
      CC-visit(G, u, k);
    endif
  endfor
```

Restituisce il numero di componenti connesse

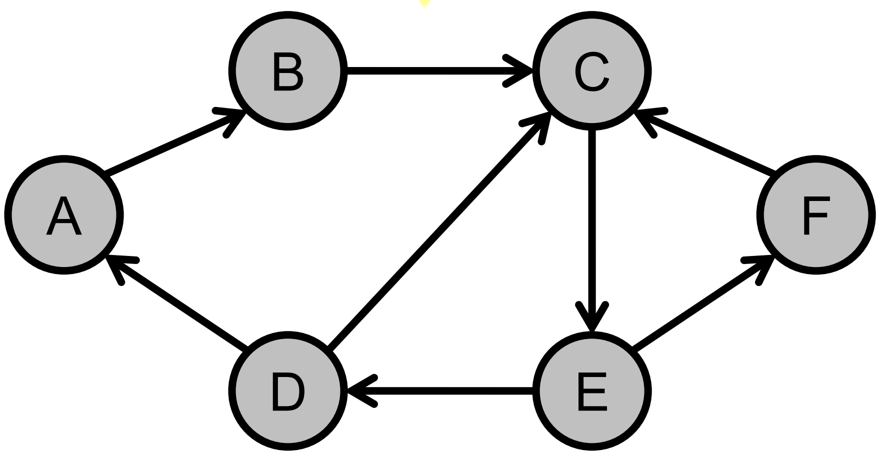
Etichetta con il valore *k* tutti i nodi della componente connessa cui appartiene *v*

Componenti Fortemente Connesse

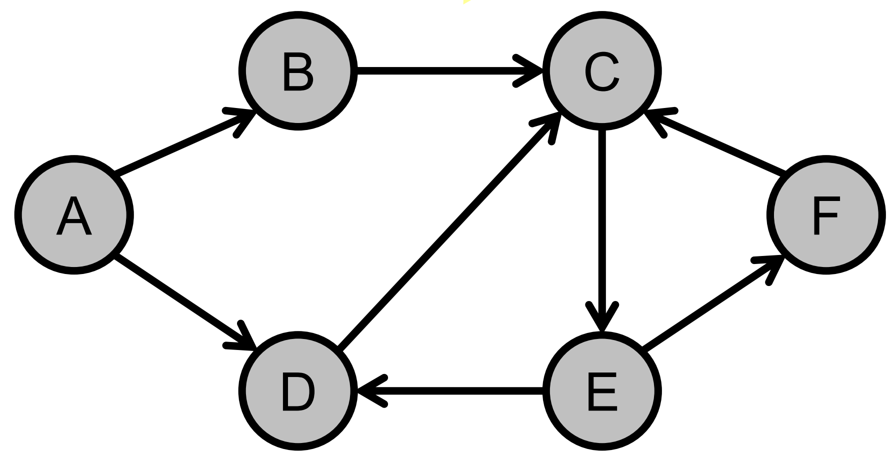
Componenti fortemente connesse
(*Strongly Connected Components*, SCC)

- Un grafo <u>orientato</u> *G* è fortemente connesso se e solo se ogni coppia di nodi è connessa da un cammino

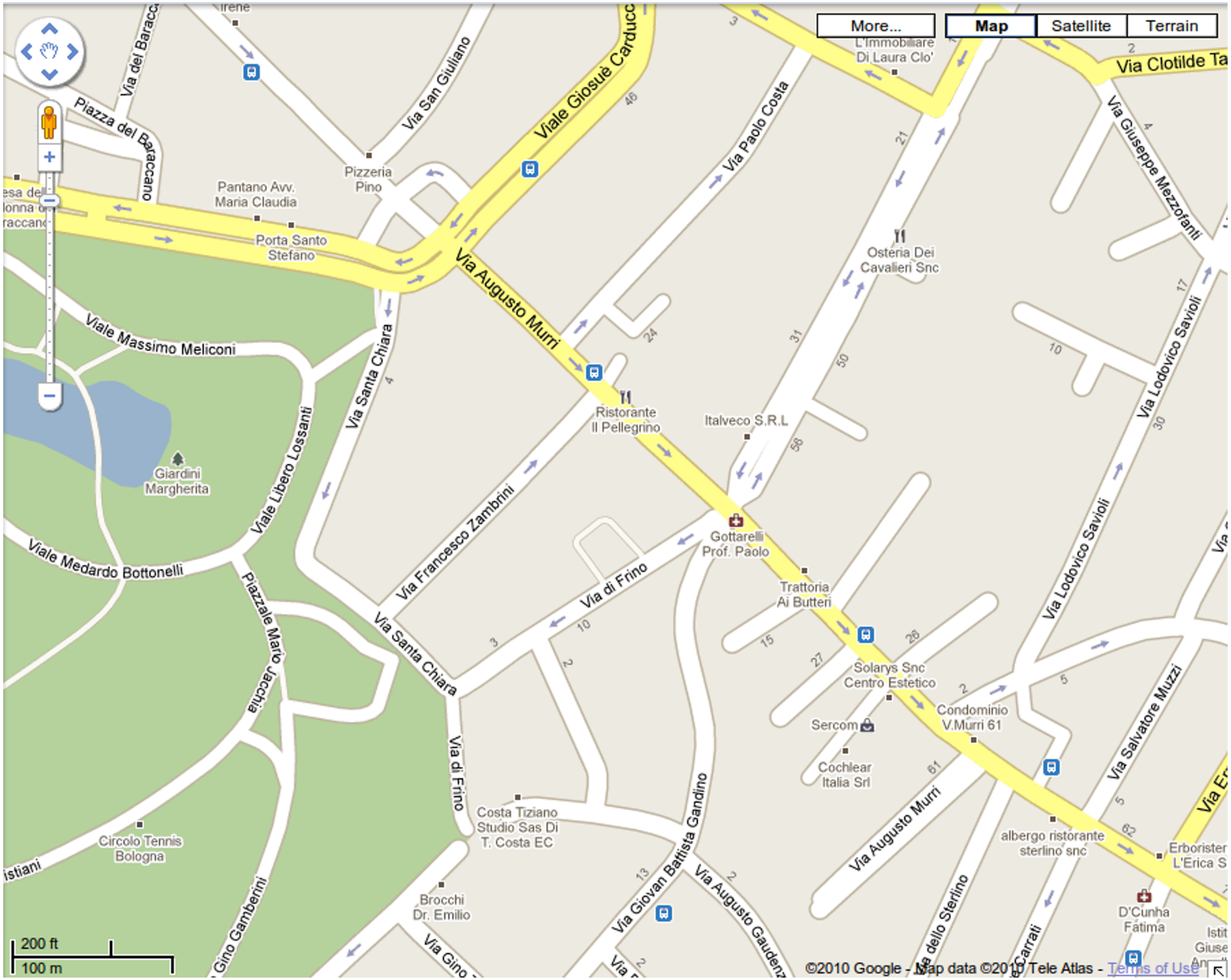
Questo grafo orientato è fortemente connesso.



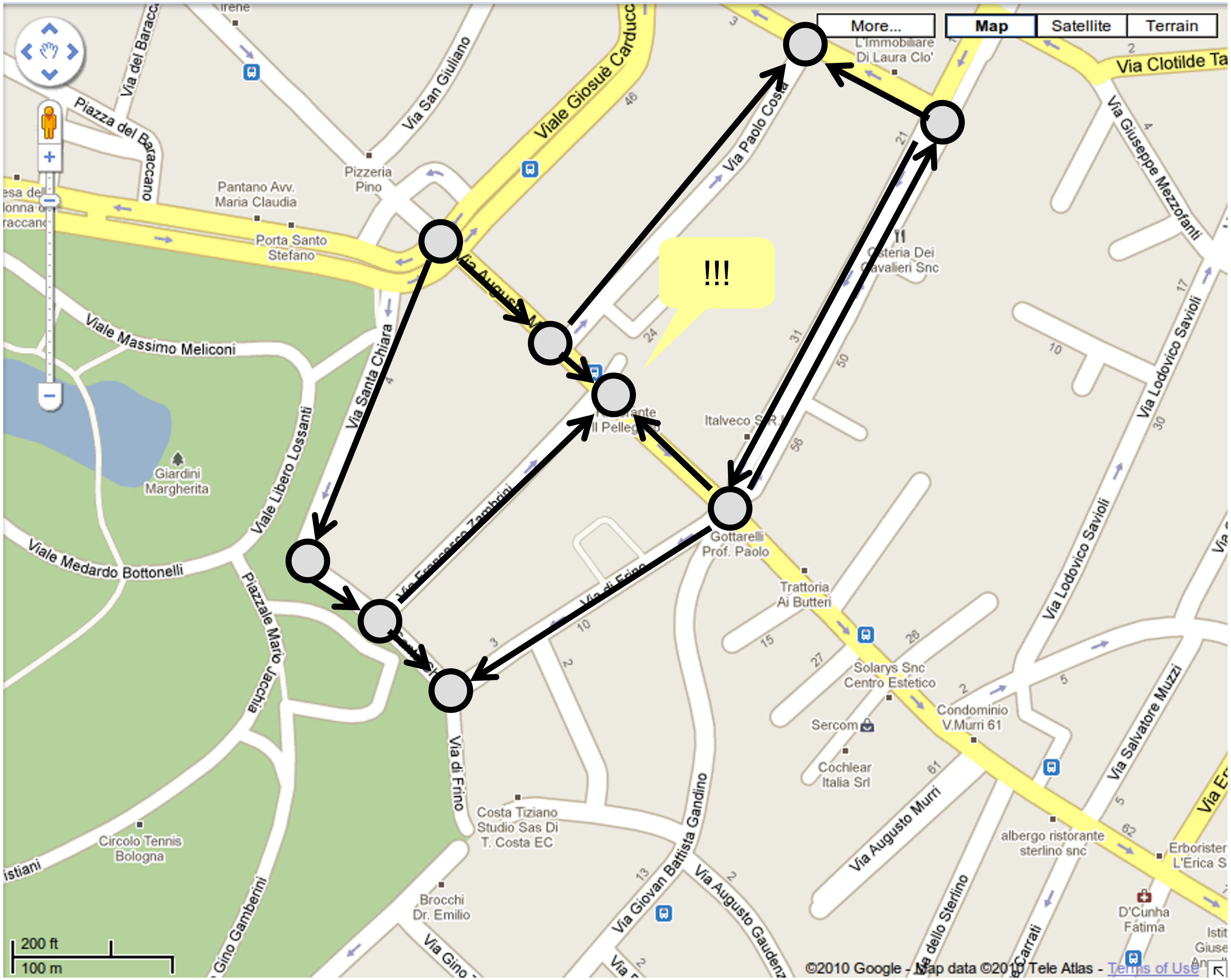
Questo grafo orientato **non è** fortemente connesso; ad es., non esiste cammino da D a A.



Nel mondo reale

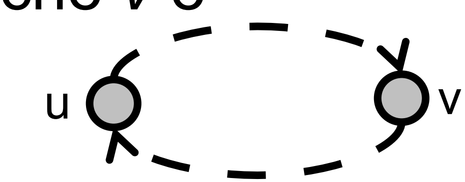


Nel mondo reale

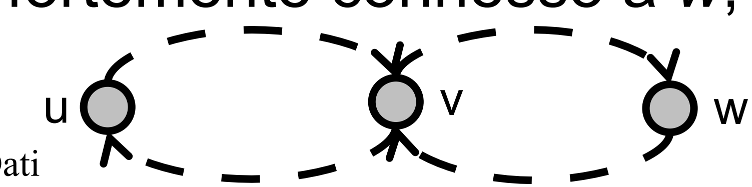


Componenti fortemente connesse

- *u* e *v* appartengono alla stessa componente fortemente connessa se e solo se esiste un cammino da *u* verso *v* e un cammino da *v* verso *u*.
- 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*.



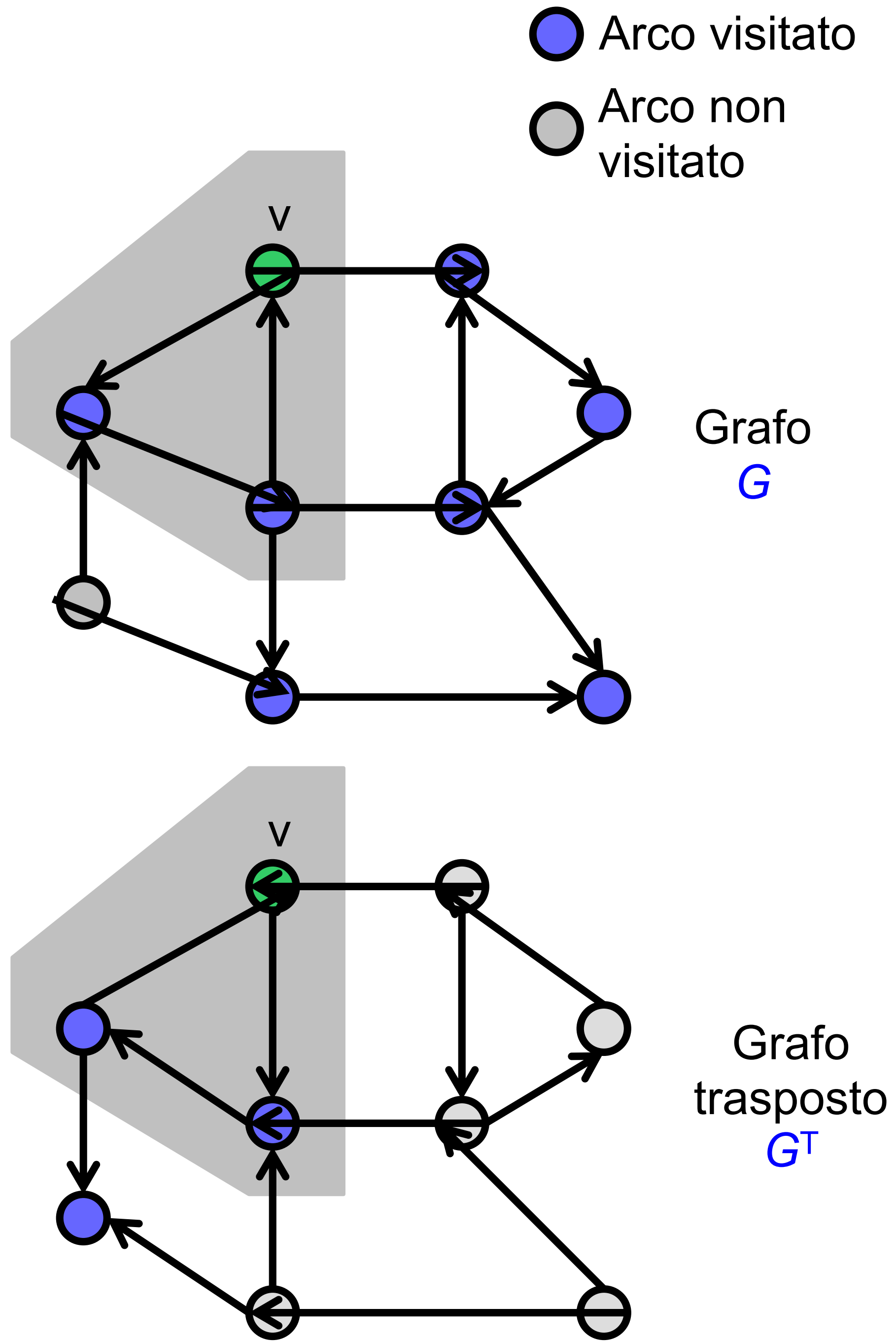
Idea

- 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 DFS che usa *u* come sorgente
  - Esiste un cammino *v*⟶⟶*u*
- cioè u è discendente di v in una visita DFS che usa *v* come sorgente

Idea

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

Idea



- Come calcolare $D(v)$?
  - $D(v)$ include i nodi visitati (DFS o BFS) usando v come sorgente
- Come calcolare $A(v)$?
  - Si inverte la direzione di tutti gli archi
  - Si effettua una nuova visita (DFS o BFS) usando v come sorgente
- Il calcolo di $A(v)$ e $D(v)$ richiede tempo $O(n + m)$

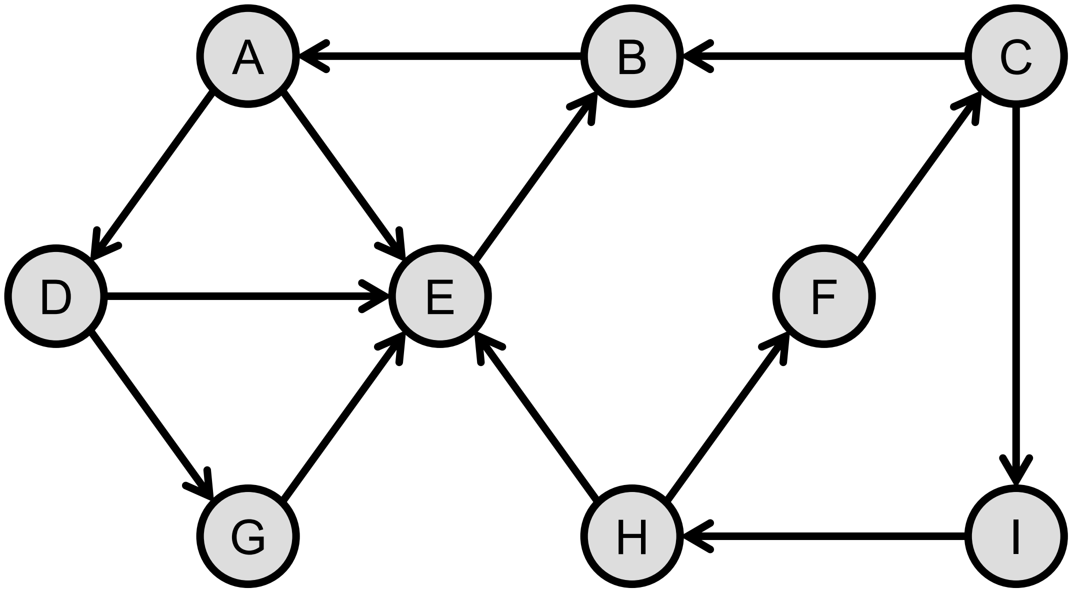
Schema di soluzione

```
Lista SCC(Grafo G, nodo x)
        Lista L ← lista vuota di nodi
(1)     Esegui DFS(G, x) marcando i nodi visitati
(2)     Calcola il grafo trasposto Gᵀ invertendo la direzione
degli archi di G
(3)     Esegui DFS(Gᵀ, 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 $O(n + m)$

Esercizio

- Calcolare tutte le componenti fortemente connesse del grafo orientato seguente

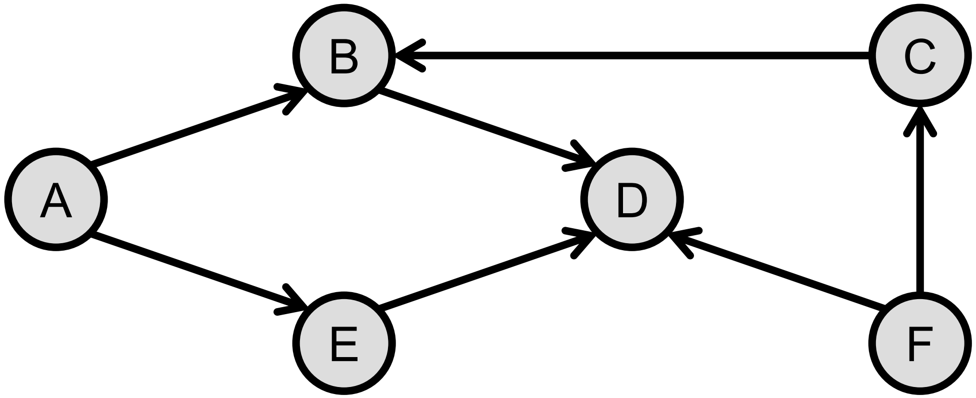


Ordinamento Topologico

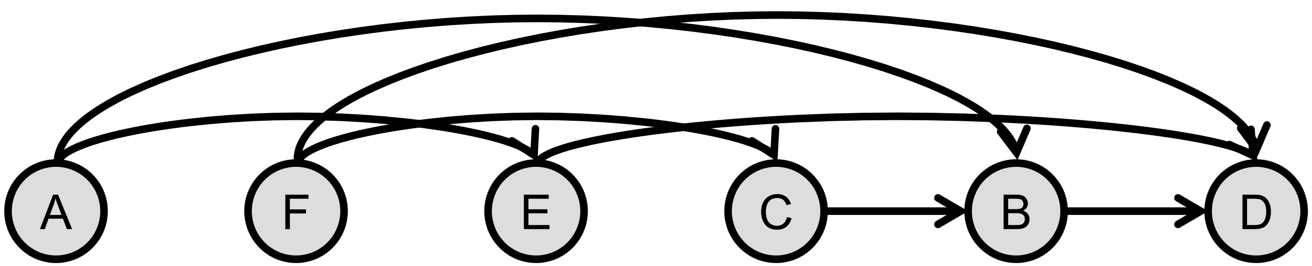
Ordinamento Topologico

- Dato un grafo orientato $G = (V, E)$, un ordinamento topologico di G è una sequenza ordinata dei suoi nodi tale che per ogni arco (x, y), allora il nodo x precede il nodo y nella sequenza

Ordinamento Topologico

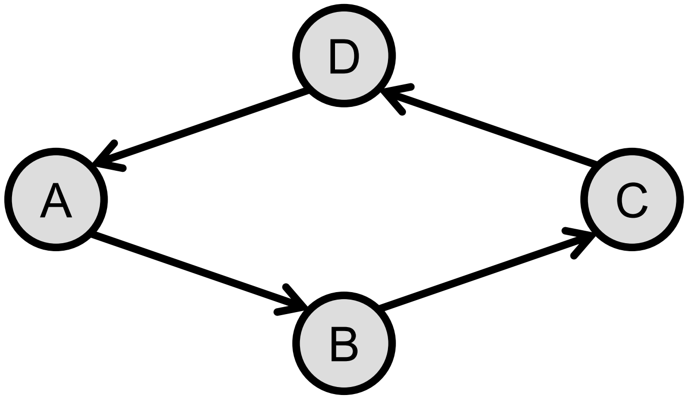


(A, F, E, C, B, D)



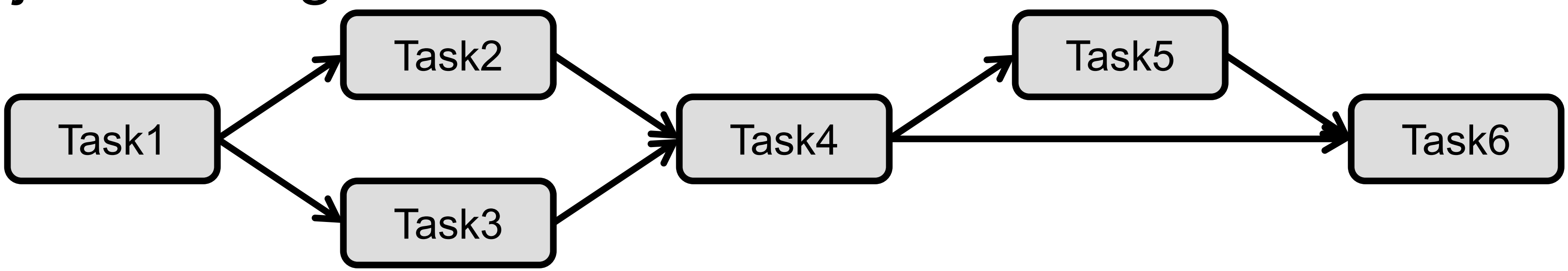
Ordinamento Topologico

- Possono esistere ordinamenti topologici diversi dello stesso grafo
- Solo i grafi orientati aciclici (DAG, *Directed Acyclic Graphs*) ammettono un ordinamento topologico



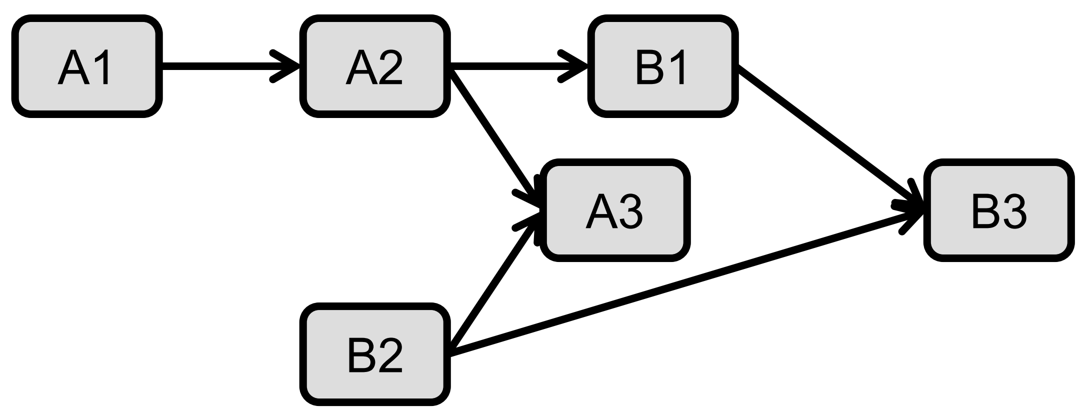
Possibili utilizzi

- Project Management



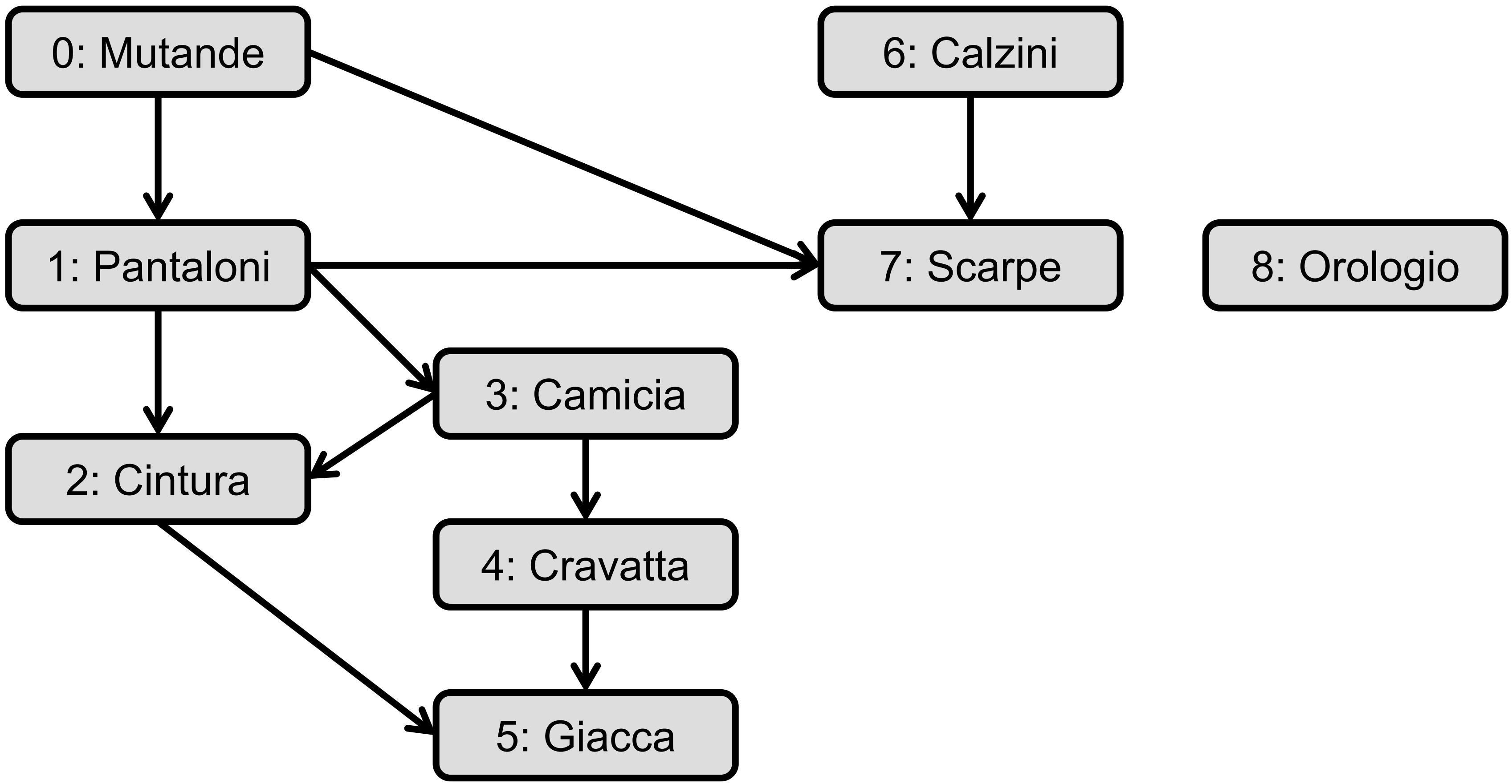
- Instruction scheduling

	A	B
1	1	A2 * 3
2	A1 + 1	3
3	A2 + B2	B1 + B2



Possibili utilizzi

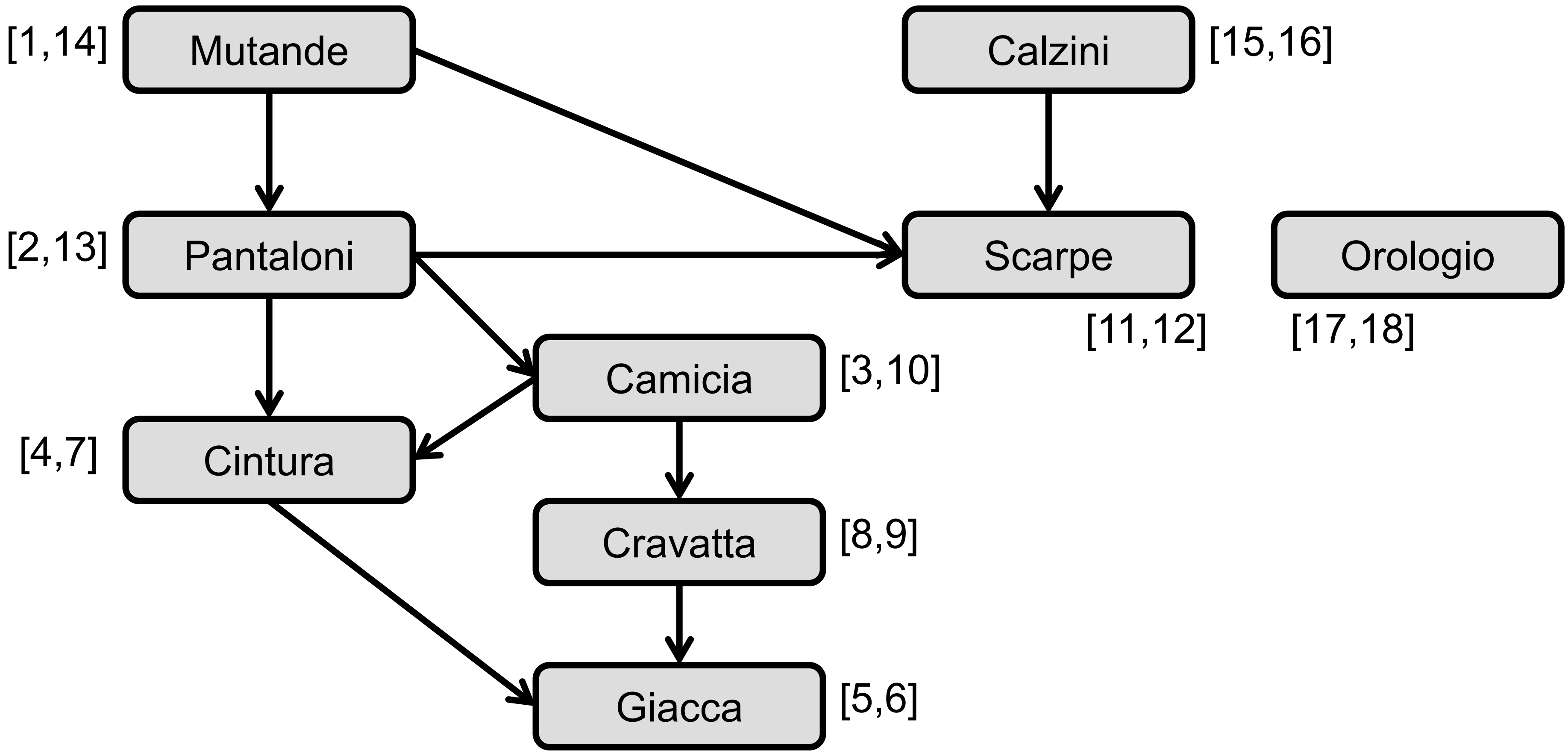
- Il professor De Funes deve rivestirsi dopo aver fatto la doccia



Calcolare un ordinamento topologico: algoritmo DFS modificato

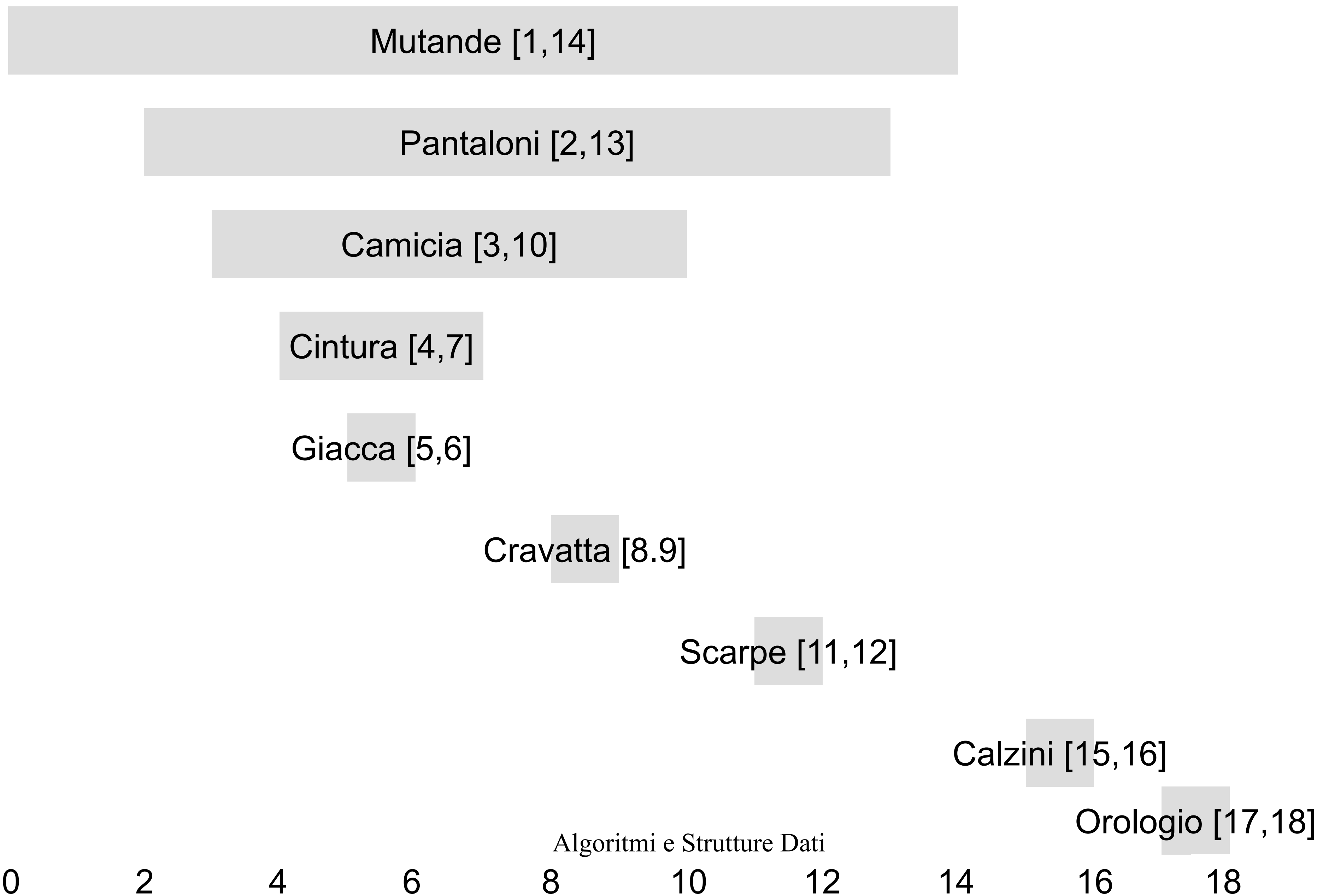
- Un possibile ordinamento è costituito dai nodi in ordine decrescente di tempo di chiusura (*finish time*)
- Algoritmo:
  - Sia *L* una lista vuota
  - Durante la visita DFS:
    - Quando si colora un nodo di nero, lo si inserisce all'inizio di L
    - Se si incontra un nodo grigio si interrompe la visita: esiste un ciclo quindi l'ordinamento topologico non esiste
  - Alla fine della visita DFS, *L* contiene i nodi in un possibile ordinamento topologico

Esempio



[1,14]
Mutande
Calzini
[15,16]
[2,13]
Pantaloni
Scarpe
Orologio
[11,12]
[17,18]
Camicia
[3,10]
[4,7]
Cintura
Cravatta
[8,9]
Giacca
[5,6]

Esempio


Mutande [1,14]
Pantaloni [2,13]
Camicia [3,10]
Cintura [4,7]
Giacca [5,6]
Cravatta [8.9]
Scarpe [11,12]
Calzini [15,16]
Orologio [17,18]
0
2
4
6
8
10
12
14
16
18

Calcolare un ordinamento topologico

- Soluzione diretta, basata sull'idea seguente:

```
L ← lista vuota
while ( non ho ordinato topologicamente tutti i nodi ) do
      if ( esiste un nodo v senza archi in ingresso ) then
            appendi v in coda alla lista L
            rimuovi v e tutti gli archi uscenti da v
      else
            errore: ordinamento topologico non esiste
      endif
endwhile
return L
```

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

```
Lista toposort( Grafo G=(V, E) )
        Lista L;
        Coda Q;
        for each u in V do
                if ( G.gradoEntrante(u) = 0 ) then
                        Q.enqueue( u );
                endif
        endfor

        while ( not Q.isEmpty() ) do
                u ← Q.dequeue();
                L.append( u );
                for each (u, v) in E do
                        G.rimuoviArco( (u, v) );
                        if ( G.gradoEntrante(v) = 0 ) then
                                Q.enqueue( v );
                        endif;
                endfor
        endwhile

        if ( L.length() = G.numNodi() ) then
                return L;
        else
                errore: ordinamento topologico non esiste
        endif
```

Costo:
O(*n* + *m*)

```
Lista TopoSort( Grafo G=(V, E) )
        Lista L;
        Coda Q;
        for each u in V do
                if ( G.gradoEntrante(u) = 0 ) then
                        Q.enqueue( u );
                endif
        endfor

        while ( not Q.isEmpty() ) do
                u ← Q.dequeue();
                L.append( u );
                for each (u, v) in E do
                        G.rimuoviArco( (u, v) );
                        if ( G.gradoEntrante(v) = 0 ) then
                                Q.enqueue( v );
                        endif;
                endfor
        endwhile

        if ( L.length() = G.numNodi() ) then
                return L;
        else
                errore: ordinamento topologico non esiste
        endif
```

La rimozione dell'arco (*u*, *v*) può avvenire contestualmente alla scansione della lista di adiacenza di *u*, quindi richiede tempo O(1)

Puo' essere implementato in tempo O(1)

Applichiamo l'algoritmo