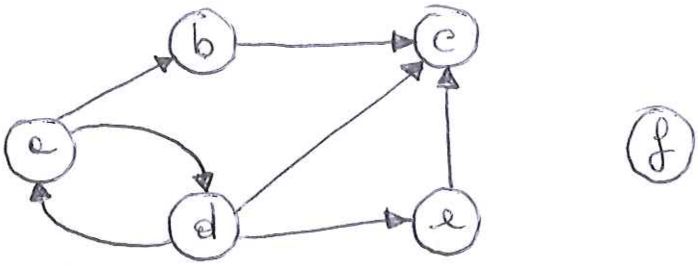
7. GRAFI

Un GRAFO è una struttura dati che permette di rappresentare delle RELAZIONI BINARIE, in particolare, permette di rappresentare qualsiasi tipo di relazione matematica.

DEFINIZIONI DI BASE

- GRAFO ORIENTATO: è una coppia di insiemi $G = (V, E)$, dove:
  - V = insieme dei NODI (o VERTICI)
  - E = insieme degli EDGES (archi), cioè coppie ORDINATE (u, v).



$V = \{a, b, c, d, e, f\}$; $E = \{(a,b), (a,d), (b,c), (d,a), (d,c), (d,e), (e,c)\}$

- GRAFO NON ORIENTATO:

  $G = (V, E)$ con V sempre insieme dei nodi ed E insieme degli archi.

  <u>PERò</u>, E stavolta è un insieme di COPPIE <u>NON</u> ORDINATE (u, v).

Riguardo all'esempio precedente, anziché avere (a,d) ed (d,a) si avrà solo uno dei 2.

- PSEUDOGRAFO: è un grafo in cui E contiene anche coppie (v, v), cioè un CAPPIO.

- CAMMINO (di lunghezza k): è una sequenza di NODI $v_1, v_2, \ldots, v_k$ tali che $(v_i, v_{i+1}) \in E$, cioè tali che c'è sempre un ARCO tra un nodo e il successivo.

- CIRCUITO: è un cammino con $v_1 = v_k$ (si possono avere nodi ripetuti)

- CICLO: è un CIRCUITO, ma SENZA NODI RIPETUTI.

- NODO ADIACENTE: un nodo V è ADIACENTE ad un nodo u, se $(u,v) \in E$.
- ARCO INCIDENTE: un ARCO (u,v) è detto INCIDENTE ~~da~~ u ~~e~~ v.
- GRADO: è il numero di ARCHI INCIDENTI di un nodo. Se il grafo è orientato, si ha un GRADO ENTRANTE e GRADO USCENTE.

- DIMENSIONI DI UN GRAFO:

$n = |V|$ ed $m = |E|$; GRAFO NON ORIENTATO $\longrightarrow m \leq \frac{n(n-1)}{2} = O(n^2)$

GRAFO ORIENTATO $\longrightarrow m \leq n^2 - n = O(n^2)$

$\Rightarrow$ La COMPLESSITÀ DEGLI ALGORITMI SUI GRAFI è spesso indicata in termini sia di n (numero di nodi) che di m (numero di archi): per esempio $O(n+m)$.

- GRAFO PESATO:

È un grafo in cui ad OGNI ARCO è associato un PESO, determinato da una funzione di peso $W: V \times V \longmapsto \mathbb{R}$.

Se $(u,v) \notin E$, il peso $w(u,v)$ è 0 oppure $+\infty$, a seconda del problema

OPERAZIONI FONDAMENTALI

- VERTICES(): restituisce l'insieme V di tutti i NODI.
- ADJ(v): restituisce l'insieme dei NODI ADIACENTI a v.
- INSERTNODE(v): inserisce nel grafo il nodo v.
- INSERTEDGE(u,v): aggiunge l'arco (u,v) all'insieme E del grafo.
- DELETENODE(v): rimuove dal grafo il NODO v e tutti gli ARCHI in cui esso è coinvolto.
- DELETEEDGE(u,v): rimuove l'arco (u,v) dal grafo, cioé dall'insieme E.

Rappresentazione dei Grafi

Ci sono 3 modalità differenti:

(1) <u>Liste di Adiacenza</u>: ad ogni NODO è associata la <u>LISTA</u> dei nodi ad esso adiacenti.

(2) <u>Matrici di Adiacenza</u>: matrice quadrata A, $n \times n$, tale che:

$$a_{ij} = \begin{cases} 1 & \text{se } (v_i, v_j) \in E \\ 0 & \text{altrimenti} \end{cases}$$

(3) <u>Matrici di Incidenza</u>: matrice rettangolare B, $n \times m$, tale che:

$$b_{ik} = \begin{cases} -1 & \text{se l'arco } k\text{-esimo esce dal nodo } i \\ 1 & \text{se l'arco } k\text{-esimo entra nel nodo } i \\ 0 & \text{altrimenti} \end{cases}$$

Confronto tra le Rappresentazioni

- <u>Liste di Adiacenza</u>: pro → individua i nodi adiacenti a v in $O'(\text{grado}(v))$
  contro → inserimenti e rimozioni in liste concatenate hanno costo $O'(\text{grado}(v))$
- <u>Matrici di Adiacenza</u>: pro → inserimento e rimozione in $O'(1)$
  contro → nodi adiacenti a v in $O'(n)$
- <u>Matrici di Incidenza</u>: pro → inserimento e rimozione in $O'(1)$
  contro → nodi adiacenti a v in $O'(m)$

Quindi, a seconda del grafo e del problema con cui si ha a che fare, si sceglie la RAPPRESENTAZIONE più efficiente.

IMPLEMENTAZIONE PYTHON (pesata, con dizionari)

```
class Graph:
  def __init__(self):
     self.nodes = { }        # dizionario

  def vertices(self):
     return self.nodes.keys()        # uso dei metodi dei dizionari

  def __len__(self):
     return len(self.nodes)

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

  def insertNode(self, u):
     if u not in self.nodes:
        self.nodes[u] = { }        # dizionario per contenere i nodi adiacenti

  def insertEdge(self, u, v, w=0):
     self.insertNode(u)
     self.insertNode(v)
     self.nodes[u][v] = w        # chiave → nodo adiacente, valore → peso arco
```

Visite di Grafi

Dato un grafo $G = (V, E)$ ed un nodo $r \in V$, detto RADICE o sorgente, si vuole visitare UNA ed UNA SOLA VOLTA TUTTI i NODI del grafo.

- Visita in Ampiezza (BFS): per livelli, prima la radice r, poi i nodi a distanza 1, poi a 2 e così via.
- Visita in Profondità (DFS): visita ricorsiva, per ogni nodo adiacente si visita RICORSIVAMENTE tale nodo.

Vediamo la BFS usata per gli alberi:

```
BFS (root):
  q ← Queue()
  node ← root
  while node ≠ ⊥ :
     <do something with node>
     for each child of node:
        q.enqueue(child)
     node ← q.dequeue()
```

* PROBLEMA!

① Nei grafi possono esserci CICLI, quindi occorre MARCARE I NODI GIA' VISITATI.

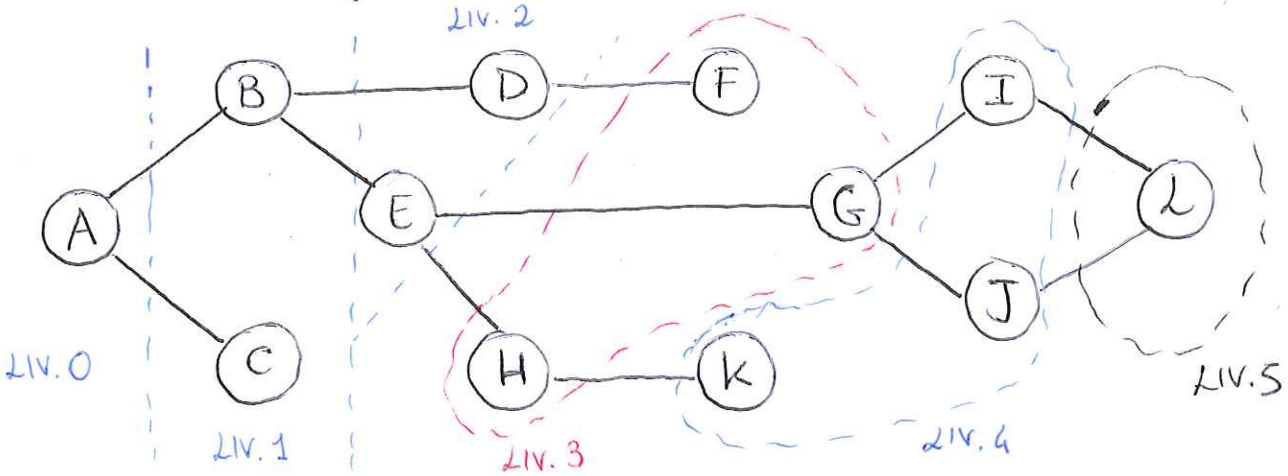
② Ci possono essere NODI ISOLATI o comunque FORESTE, ma il problema è risolvibile avendo a disposizione l'insieme di tutti i nodi.

- Vediamo ora prima uno schema generale di un ALGORITMO DI VISITA e poi in particolare quello della Visita in Ampiezza (BFS) ed in Profondità (DFS).

Algoritmo Generale di Attraversamento

```
GraphTraversal (G, r):
    Set S ← ∅                                  # insieme dei nodi scoperti ma non ancora visitati
    S.insert(r)
    < marca il nodo r come già scoperto >
    while S.size() > 0 :
        Node u ← S.remove()
        < visita il nodo u >
        for each v in G.adj(u):                 # Ci si muove tra i nodi adiacenti
            if v non è ancora stato marcato :
                < marca il nodo v come scoperto >
                S.insert(v)
```

- Visita in Ampiezza - BFS :
  - Fa uso di una **CODA** (Queue) per memorizzare i nodi adiacenti NON ANCORA SCOPERTI.
  - I nodi raggiungibili non marcati, vengono quindi marcati (ATTENZIONE: UN NODO VIENE MARCATO QUANDO SCOPERTO, NON QUANDO VISITATO!)
  - La visita procede estraendo il nodo successivo dalla coda. Essendo la struttura dati d'appoggio una coda, si ottiene una VISITA PER LIVELLI.



BFS (G, r) :

```
Queue Q ← ∅
Q.enqueue(r)
for each u in G.vertices() \ {r} :      # tutti i nodi eccetto la radice
    u.discovered ← false                #   vengono marcati come NON SCOPERTI

r.visited ← true                        # la radice è marcata come VISITATA

while not Q.isEmpty() :
    Node u ← Q.dequeue()
    <visita il nodo u>
    for each v in G.adj(u) :
        <visita l'arco (u,v)>
        if not v.discovered :           # se un nodo non è ancora stato scoperto
            v.discovered ← true         #   viene marcato come SCOPERTO e messo
            Q.enqueue(v)                #   in coda
```

<u>COMPLESSITA'</u> : $\boxed{O(n+m)}$ poiché visita una sola volta tutti i nodi e tutti gli archi.

• VISITA IN PROFONDITÀ – DFS:

Il principio utilizzato è lo stesso; la coda è sostituita da uno **STACK** (pila). Ciò permette una visita per nodi adiacenti e non per livelli:

```
DFS(G, r):
    Stack S ← ∅
    S.push(r)
    for each u in G.vertices():
        u.discovered ← false
    while not S.isEmpty():
        Node u ← S.pop()
        if not u.discovered:        # e' anticipata la condizione di NODO NON SCOPERTO!
            <visita il nodo u>
            u.discovered ← true
            for each v in G.adj(u):
                <visita l'arco (u,v)>
                S.push(v)
```

<u>COMPLESSITÀ</u>: $\boxed{O(n+m)}$

Cammini Minimi

Dato un <u>GRAFO PESATO</u> $G = (V, E)$, con pesi ≥ 0, ci sono diverse varianti:

(1) Trovare il cammino minimo tra un NODO SORGENTE $s \in V$ ed un NODO DESTINAZIONE $t \in V$;

(2) Trovare il cammino minimo tra un NODO SORGENTE $s \in V$ e TUTTI gli altri NODI del grafo G (<u>SINGLE SOURCE SHORTEST PATH – SSSP</u>);

(3) Trovare il cammino minimo tra TUTTE LE COPPIE di nodi del grafo (All Pairs Shortest Path – APSP).

• Archi con Peso Uguale (Erdös Number):

```
CAMMINI (G, r):
  Queue Q ← ∅                                  # è una BFS (uso di una CODA)
  Q.enqueue(r)
  for each u in G.vertices() \ {r}:
      u.distance ← ∞                           # sovrastima della distanze
  r.distance ← 0
  r.parent ← ⊥

  while not Q.isEmpty():
      Node u ← Q.dequeue()
      for each v in G.adj(u):
          if v.distance == ∞:                  # solo se il nodo non è ancora stato
                                               #   scoperto
              v.distance ← u.distance + 1
              v.parent ← u                     # i nodi visitati vengono fatti puntare
                                               #   AL GENITORE
              Q.enqueue(v)
```

* <u>ERDÖS NUMBER</u>: tutti gli archi hanno peso 1. Una volta creati tutti i riferimenti al genitore, se per ogni nodo li si percorre, si ottiene un albero, detto <u>ALBERO RICOPRENTE</u> del grafo.

* PROBLEMA: con l'algoritmo visto prima, viene preso come CAMMINO MINIMO il PRIMO PERCORSO per cui raggiungo la destinazione. MA, con archi che hanno PESI DIVERSI, il primo percorso potrebbe non essere quello a peso minimo!

ALGORITMO DI DIJKSTRA

È l'algoritmo SSSP. È un ALGORITMO GREEDY, poiché si basa sull'idea che ogni sottocammino di un cammino minimo è un cammino minimo.

- Si sovrastima la DISTANZA di ciascun vertice dalla radice $(+\infty)$;
- Si visita ogni nodo ed i suoi nodi adiacenti (DFS) per trovare il sottopercorso più breve verso i vicini.

Dalla **CODA DI PRIORITÀ** viene preso il nodo NON VISITATO con DISTANZA MINIMA; se per giungere ai nodi adiacenti si trova una distanza minore, essa viene aggiornata.
Un NODO viene MARCATO come VISITATO, solo DOPO aver VISITATO TUTTI GLI ADIACENTI.

```
SSSP(G, r):
  r.distance ← 0
  MinHeap PQ ← ∅                       # si usa una CODA DI PRIORITÀ (Min Heap)
  for each v in G:
    if v ≠ r:
      v.distance ← ∞                   # sovrastima della distanza
    v.parent ← NIL
    PQ.enqueue(v)                      # tutti i nodi di G sono in PQ
  while PQ is not empty:
    u ← PQ.getMin()                    # si inizia dalla radice r (distanza 0)
                                       # prendere i nodi dal Min Heap, vuol dire
                                       #   toglierli da PQ e quindi marcarli
    for each v in G.adj(u):
      if v is in PQ:
        newDist ← u.distance + w(u,v)  # si aggiunge il PESO dell'arco
        if newDist < v.distance:
          v.distance ← newDist
          v.parent ← u
          PQ.decreasePrio(v, newDist)  # si aggiorna la coda di priorità, in
                                       #   modo tale che getMin(), prendendo
                                       #   la RADICE del Min Heap, funzioni
                                       #   correttamente.
```

se $v \notin PQ$, vuol dire che è già stato visitato.

GRAFI ACICLICI

In un grafo NON ORIENTATO $G=(V,E)$ un ciclo C di lunghezza ($k>2$) è una sequenza di nodi $u_0, u_1, \dots, u_k$ tale che $\forall i : 0 \leq i \leq k-1 \ (u_i, u_{i+1}) \in E$ ed $u_0 = u_k$.
La CONDIZIONE $k>2$ ESCLUDE CICLI BANALI tra 2 elementi, sempre presenti in grafi NON ORIENTATI.

- GRAFI NON ORIENTATI ACICLICI:

Un grafo è ACICLICO se non contiene alcun ciclo (di lunghezza $k>2$).

```
HAS CYCLES(G):
  for each u in G.vertices():
    u.visited ← false
  for each u in G.vertices():          # serve per esaminare le FORESTE
    if not u.visited:
      if HAS CYCLE(G, u, ⊥):           # è ricorsiva e vede se c'è un ciclo
        return true
  return false

HAS CYCLE(G, u, p):                    # p = parent, il nodo da cui si proviene
  u.visited ← true                     # viene marcato u come VISITATO
  for each v in G.adj(u) \ {p}:        # si esclude il nodo da cui si proviene (k>2)
    if v.visited:
      return true
    else if HASCYCLE(G, v, u):         # chiamate ricorsive sugli adiacenti
      return true
  return false
```

Praticamente ritorna FALSE quando tutti i nodi sono stati visitati e per nessun nodo è stato trovato un ciclo.
È come se si esplorasse il grafo con una DFS partendo da u come RADICE, c'è però bisogno di tenere traccia del genitore p, per escluderlo dall'analisi dei nodi. È una DFS RICORSIVA!

Cicli in Grafi Orientati:

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

Un CICLO SEMPLICE è tale se tutti i suoi nodi sono distinti, tranne u_0 ed u_k. Cioè, se non è un circuito.

Un GRAFO ORIENTATO ACICLICO è detto DAG (Directed Acyclic Graph):

in ogni DAG c'è sempre un NODO POZZO, cioè un nodo che non ha archi uscenti.

Percorrendo un GRAFO, si tiene traccia del "tempo di visita": ogni nodo ha un TEMPO D'ENTRATA ed un TEMPO D'USCITA, aggiornati mediante un TIMESTAMP GLOBALE.

Così facendo, si possono classificare gli ARCHI (u,v):

- ARCO DELL'ALBERO: se $v.\text{enterT} == 0$, cioè se v non è ancora mai stato visitato;
- ARCO ALL'INDIETRO: se $u.\text{enterT} > v.\text{enterT}$ and $v.\text{exitT} == 0$, cioè se sono tornato in un nodo v di cui però non ho ancora finito di esaminare tutti i percorsi che partono da lui $\rightarrow$ È UN CICLO!
- ARCO IN AVANTI: se $u.\text{enterT} < v.\text{enterT}$ and $v.\text{exitT} \neq 0$, è semplicemente un arco per andare avanti nell'algoritmo.
- ARCO DI ATTRAVERSAMENTO: in tutti gli altri casi.

* Intuitivamente $\rightarrow$ se trovo un ARCO ALL'INDIETRO ho un CICLO.

Un DAG è un GRAFO SENZA ARCHI ALL'INDIETRO.

- **CLASSIFICAZIONE DEGLI ARCHI:**

```
global timestamp

CLASSIFYEDGES (G, u):
  for each v in G.adj(u):
    if v.enterT == 0:
      < arco dell'albero >
      timestamp ← timestamp + 1
      u.enterT ← timestamp                    # TEMPO D'ENTRATA
      CLASSIFYEDGES (G, v)                    # chiamata RICORSIVA sui nodi adiacenti

    else if u.enterT > v.enterT and v.exitT == 0:
      < ARCO ALL'INDIETRO >

    else if u.enterT < v.enterT and v.exitT ≠ 0:
      < arco in avanti >

    else:
      < arco di attraversamento >
  timestamp ← timestamp + 1
  u.exitT ← timestamp                         # TEMPO D'USCITA (alla fine del ciclo for)
```

- **INDIVIDUAZIONE DI CICLI IN UN GRAFO ORIENTATO:**

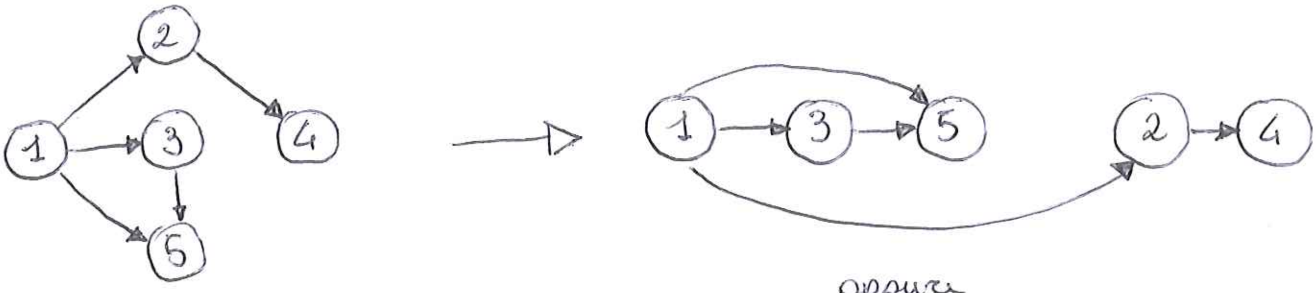
```
global timestamp
HASCYCLES (G, u):
  for each v in G.adj(u):
    if v.enterT == 0:                         # arco dell'albero
      timestamp ← timestamp + 1
      u.enterT ← timestamp
      if HASCYCLES (G, v):                    # chiamata RICORSIVA sui nodi adiacenti
        return true
    else if u.enterT > v.enterT and v.exitT == 0:   # ARCO ALL'INDIETRO → CICLO
      return true
  timestamp ← timestamp + 1
  u.exitT ← timestamp
  return false
```

- Non appena viene trovato un ARCO ALL'INDIETRO, si restituisce True!

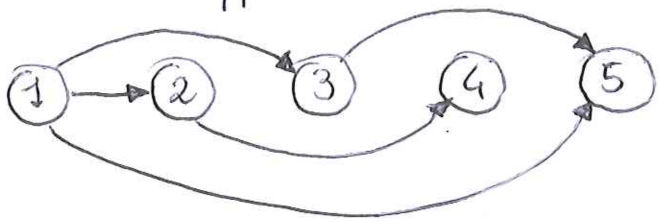
Ordinamento Topologico

È **POSSIBILE** SOLO per i **DAG**. Dato un DAG, un' **ORDINAMENTO TOPOLOGICO** è un ordinamento lineare dei nodi, tale per cui se $(u,v) \in E \Rightarrow u$ appare prima di v nell'ordinamento.

Dato un DAG, possono esserci più ordinamenti topologici.



oppure



- **Algoritmo Naïve:**

Sfrutta la proprietà che un SOTTOGRAFO di un DAG È un DAG.
Individua il NODO POZZO, lo mette in una **SEQUENZA** e lo elimina dal grafo, con i suoi archi incidenti. Ripete l'operazione per tutti i nodi del Grafo.

```
TopologicalSort(G):
    Sequence order ← ∅
    for i ← |G.vertices()| - 1 down to 0:
        v ← nodo pozzo in G.vertices()
        order.append(v)
        G.removeNode(v)          # rimuove anche gli archi incidenti a v.
```

- TOPOLOGICAL SORT con DFS e STACK:

```
TOPOLOGICAL SORT DFS (G, u, S):                 # S è uno STACK
   u.visited ← true                              # viene MARCATA la RADICE
   for each v in G.adj(u):                       # DFS RICORSIVA
      if not v.visited:
         TOPOLOGICAL SORT DFS (G, v, S)          # chiamate ricorsive
   S.push(u)

TOPOLOGICAL SORT (G):
                                                 # uso di STACK
   Stack S ← ∅
   for each u in G.vertices():
      u.visited ← false
   for each u in G.vertices():                   # per gestire le FORESTE
      if not u.visited:
         TOPOLOGICAL SORT DFS (G, u, S)          # DFS RICORSIVA
   return S                                      # ordine LIFO
```

ORDINAMENTO TOPOLOGICO IN GRAFI GENERICI

Si può estendere l'algoritmo visto sopra anche a grafi contenenti dei CICLI. Infatti, dall'esecuzione di TOPOLOGICAL SORT () siamo sicuri che:

- se un arco (u,v) non fa parte di un ciclo, allora u appare prima di v nell'ordinamento;
- gli archi (u,v) che appartengono ad un ciclo appaiono in un qualche ordine, MA tale ordine NON È INFLUENTE ai fini dell'ordinamento topologico.

Si può utilizzare l'algoritmo per la classificazione degli archi CLASSIFY EDGES (G, u), aggiungendo uno STACK globale e, alla fine dell'algoritmo, inserire S.push(u).

In tal modo, i nodi verranno ORDINATI PER TIMESTAMP DI USCITA.

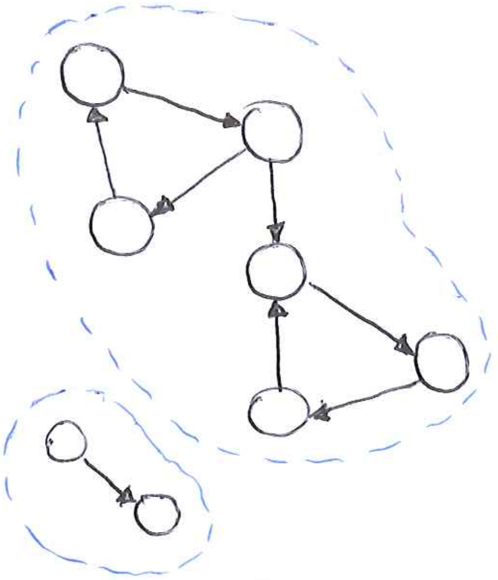
Componenti Connesse

Due nodi u, v sono <u>CONNESSI</u> in un GRAFO ORIENTATO se esiste un <u>CAMMINO</u> che collega u e v.

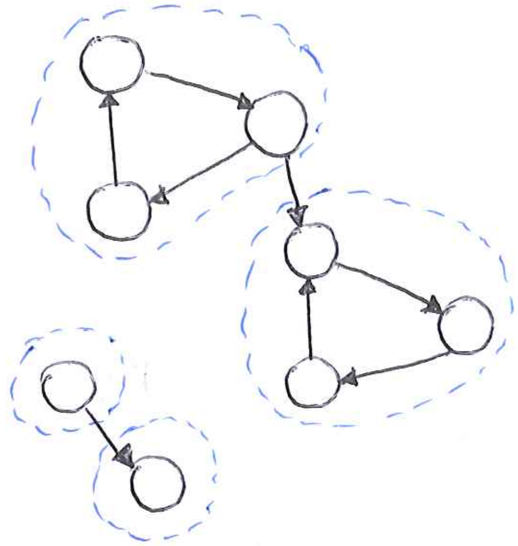
Un grafo diretto è FORTEMENTE CONNESSO se per ogni coppia (u, v) esiste un <u>CAMMINO</u> da u a v.

Una COMPONENTE CONNESSA di un grafo G è un sottografo G' <u>CONNESSO</u> e <u>MASSIMALE</u> di G.

Componenti Connesse



Componenti Fortemente Connesse



Individuazione delle Componenti Connesse

Serve per individuare i vari "GRUPPI" che compongono una <u>FORESTA</u>.

Si può fare TRAMITE $\boxed{\text{DFS}}$:

(1) Ogni NODO viene "AUMENTATO" con un intero che indica il gruppo cui appartiene (u.id);

(2) Si itera su tutti i nodi del grafo, con <u>DFS</u>;

(3) Se un nodo ha <u>u.id == 0</u> (cioè non appartiene ancora a nessuna componente connessa) <u>SI VISITA IL GRAFO A PARTIRE DA QUEL NODO</u>, associando tutti i nodi visitati alla stessa componente connessa.

```
ccDFS(G, id, u):
  u.id ← id                          # si assegna il nodo alla comp. connessa id
  for each v in G.adj(u):
    if v.id == 0:                    # se il nodo ancora non appartiene ad un "gruppo"
      ccDFS(G, id, v)                # DFS RICORSIVA
```

```
cc(G):
  for each u in G.vertices():
    u.id ← 0                         # imposta a 0 tutte le id
  id ← 0                             # id parte da 0
  for each u in G.vertices():
    if u.id == 0:
      id ← id + 1                    # incrementa id
      ccDFS(G, id, u)                # trova la componente connessa NUMERO id
```

Componenti Fortemente Connesse

Non è POSSIBILE applicare l'algoritmo precedente per l'individuazione delle componenti fortemente connesse, poiché la soluzione DIPENDEREBBE DAL NODO DI PARTENZA.

- **Grafo Trasposto**: Sia $G=(V,E)$ un grafo; il suo Grafo Trasposto è $G^T=(V,E^T)$, cioè un grafo con gli stessi nodi, ma ARCHI ORIENTATI AL CONTRARIO.

```
TRANSPOSE(G):
  Graph G^T ← ∅
  for each u in G.vertices():
    G^T.insertNode(u)
  for each u in G.vertices():        # per ogni nodo ...
    for each v in G.adj(u):          # ... si cicla sui suoi adiacenti
      G^T.insertEdge(v, u)           # inserire gli archi al contrario
  return G^T
```

Algoritmo di Kosaraju – Comp. Fortemente Connesse

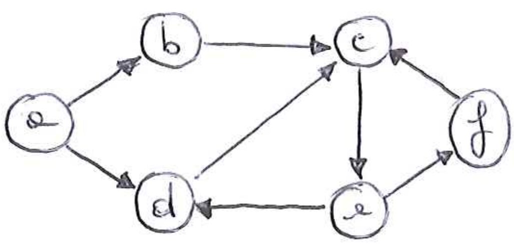
Individua le COMPONENTI FORTEMENTE CONNESSE in tempo lineare $\boxed{\Theta(n + m)}$.
L'idea di base è che un grafo G ed il TRAPOSTO G^T hanno le STESSE SCC.

(1) Si fa una PRIMA DFS su G, per calcolare i tempi d'uscita di ogni nodo;

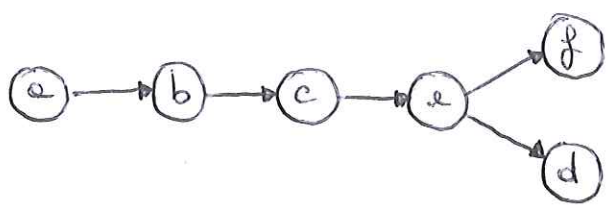
(2) Si calcola il GRAFO TRASPOSTO G^T;

(3) Si fa una seconda DFS su G^T, partendo dai nodi con TEMPO D'USCITA MAGGIORE.

```
KOSARAJU(G):
    Stack S ← TOPOLOGICAL SORT(G)      # il topologicalSort() usa la DFS
    G^T ← TRANSPOSE(G)
    return CC(G^T, S)                  # viene usato l'ordine nello Stack, cioè
                                       # per exitTime decrescente
```

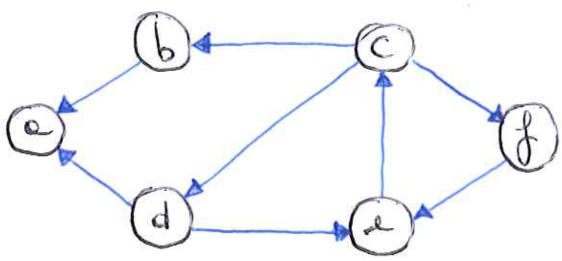


(1) G

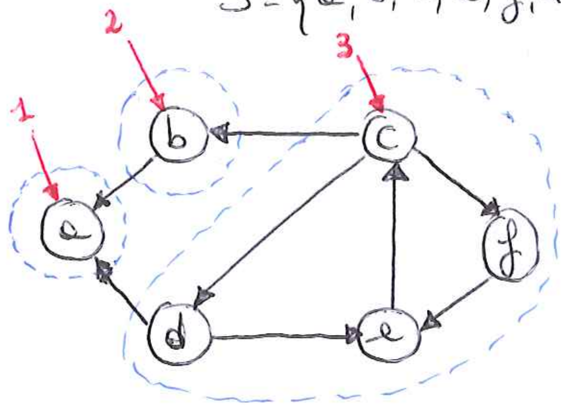


(2) TOPOLOGICAL SORT(G)

$S = \{a, b, c, e, f, d\}$



(3) G^T



(4) SCC() – Strongly Connected Components

MINIMUM SPANNING TREE – MST

L'obiettivo è individuare un SOTTOGRAFO di un grafo che mantenga la CONNETTIVITÀ TRA TUTTI I NODI AL MINOR COSTO POSSIBILE.

La soluzione è sempre un ALBERO, in quanto è un grafo aciclico.

La ricerca di un MST si basa su **ALGORITHMI GREEDY**, che cercano di costruire l'MST aggiungendo un arco alla volta.
L'MST costruito incrementalmente contiene archi che sono sempre un sottoinsieme degli archi che compongono il MST.

- EDGE SAFETY: un arco è SICURO se può essere aggiunto al MST in costruzione senza violare l'ottimalità globale del MST.

- EDGE SAFETY:

Un TAGLIO è una partizione di V in due insiemi $\rightarrow (S, V \setminus S)$;

Un arco (u,v) ATTRAVERSA IL TAGLIO se $u \in S$ e $v \in V \setminus S$, o viceversa;

Un taglio RISPETTA un insieme di archi $A \subset E$ se nessun arco di A attraversa il taglio;

Un ARCO LEGGERO è l'arco a PESO MINIMO tra quelli che ATTRAVERSANO un taglio.

- $\boxed{\text{TEOREMA}}$: Sia $G = (V, E)$ un GRAFO NON ORIENTATO e CONNESSO, con pesi ≥ 0 per gli archi. Sia $A \subset E$, incluso in qualche MST di G.
Sia $(S, V \setminus S)$ un TAGLIO QUALSIASI che RISPETTA A; sia (u,v) un ARCO LEGGERO (attraversa il taglio con peso minimo) di tale taglio.

$\Rightarrow$ Allora: (u,v) è un ARCO SICURO per A!

Quindi (u,v) può essere aggiunto ad A per la COSTRUZIONE dell'MST.

* Scegliere ogni volta l'arco a peso minimo che attraversa il taglio porta alla costruzione del MST!

TROVA MST (G):

```
A ← ∅
while A non è un MST di G:
    trova un arco (u,v) sicuro per A
    A ← A ∪ {(u,v)}
return A
```

Algoritmo generale di COSTRUZIONE di un MST

ALGORITMO DI BORŮVKA

- L'insieme A forma un insieme di COMPONENTI CONNESSE: all'inizio, c'è una componente connessa per ciascun nodo;
- Per ogni nodo, viene determinato l'ARCO di PESO MINORE che connette 2 componenti connesse in A;
- Viene selezionata una componente connessa da A;
- Si seleziona l'ARCO a PESO MINIMO che connette quella componente connessa con un'altra componente connessa di A;
- Si rimuovono gli archi che connettono 2 nodi appartenenti alla stessa componente connessa ma non sono in A.

COMPLESSITA': $O(|V| \log |E|) = \boxed{O(m \log m)}$

ALGORITMO DI KRUSKAL

È simile a Borůvka, ma sfrutta strutture dati differenti. Gli ARCHI vengono ORDINATI PER PESO, e si procede prendendo l'ARCO a PESO MINIMO che collega 2 componenti connesse. COMPLESSITA': $\boxed{O(m \log m)}$

KRUSKAL (G):

```
A ← ∅
for each v in G.vertices():
    MAKESET(v)
for each(u,v) in G.edges() ordinati per w(u,v) crescente:
    if FINDSET(u) ≠ FINDSET(v):        # se u e v appartengono a 2 componenti diverse
        A ← A ∪ {(u,v)}
        UNION(FINDSET(u), FINDSET(v))  # unione delle 2 componenti connesse in 1 unica
return A
```

ALGORITMO DI PRIM

- L'insieme A costituisce SEMPRE UNA ed UNA SOLA COMPONENTE CONNESSA;
- A viene inizializzato con un nodo scelto a caso;
- Una **CODA DI PRIORITÀ** viene riempita con i nodi del grafo e le distanze dal nodo scelto a caso ed inserito in A (se non c'è un arco che collega il nodo scelto ad un nodo v del grafo, allora v.distance = +∞);
- Ad ogni passo, si aggiunge ad A il nodo raggiunto dall'ARCO LEGGERO che attraversa il taglio (A, V\A).

* Intuitivamente: ad ogni passo si aggiunge il nodo raggiungibile dalla componente connessa (A) a DISTANZA MINIMA.

```
PRIM (G, r):
  A = {r}
  Min Heap PQ ← ∅                        # PRIORITY QUEUE
  for each v in G.vertices():
      v.distance ← w(r,v)                # vale ∞ se r e v non sono adiacenti
      v.parent ← NIL                     # occorre tenere traccia del percorso, in quanto
                                           stavolta A contiene NODI, NON ARCHI
      PQ.enqueue(v)
  while PQ is not empty:
      u ← PQ.getMin()
      (u,v) ← arco a minima distanza, con v∈A      # ARCO LEGGERO
      A ← A ∪ {u}
      u.parent ← v
      for each u in PQ tale che (u,v) ∈ E:
          if u.distance > w(v,u):
              PQ.updatePriority(u, w(v,u))
```

COSTO: $\boxed{O(m \log n)}$