SAPIENZA
UNIVERSITÀ DI ROMA

UNIVERSITÀ "SAPIENZA" DI ROMA
FACOLTÀ DI INFORMATICA

Progettazione di Algoritmi

Appunti integrati con il libro "Introduzione agli algoritmi e strutture dati", T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein

Author
Simone Bianco

5 marzo 2023

Indice

Capitolo 0

Introduzione

Capitolo 1

Teoria dei grafi

1.1 Grafi, vertici e archi

Definition 1. Grafo

Un **grafo** $G = (V, E)$ è una struttura matematica composta da un insieme V di **vertici (o nodi)** ed un insieme E di **archi (o spigoli)** che collegano due vertici, dove:

$$E = \{(v_1, v_2) \mid v_1, v_2 \in V, v_1 \neq v_2\}$$

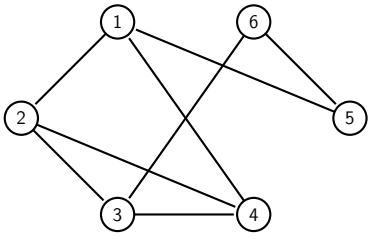
Di conseguenza, in un grafo non sono presenti né **archi ripetuti tra due vertici**, né **cappi**, ossia archi da un vertice in se stesso.

Definition 2. Multigrafo

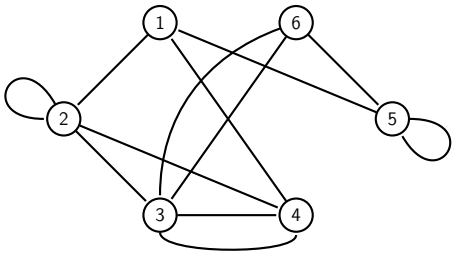
Un **multigrafo** $G = (V, E)$ è un particolare tipo di grafo dove **sono concessi archi ripetuti e cappi** nell'insieme degli archi E

Esempio:

Grafo



Multigrafo



Definition 3. Incidenza e adiacenza

Sia $G = (V, E)$ un grafo o un multigrafo. Se $(v_1, v_2) \in E(G)$, allora definiamo l'arco (v_1, v_2) come **incidente in** v_1 **e** v_2, mentre definiamo v_1 e v_2 come **adiacenti**

Definition 4. Grafo diretto e non diretto

Sia $G = (E, V)$ un grafo. Definiamo G come **grafo diretto**, o **digrafo**, se i suoi archi possiedono un **orientamento**, ossia se

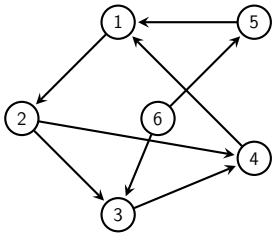
$$(v_1, v_2) \in E(G) \implies (v_2, v_1) \notin E(G)$$

Viceversa, definiamo G come **grafo non diretto**, o semplicemente **grafo**, se i suoi archi non possiedono orientamento, ossia se:

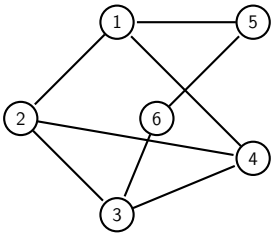
$$(v_1, v_2) \in E(G) \implies (v_2, v_1) \in E(G)$$

Esempio:

Grafo diretto



Grafo non diretto



Definition 5. Grado di un vertice

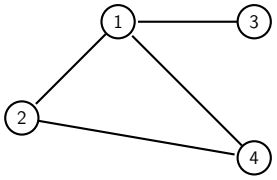
Sia $G = (V, E)$ un grafo o un multigrafo e sia $v \in V(G)$.

Se G è un grafo **non diretto**, definiamo come **grado** di v, indicato come $deg(v)$, il numero di archi incidenti a v_1

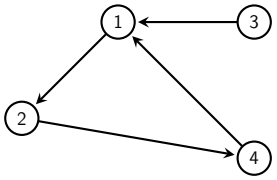
Se invece G è un grafo **diretto**, definiamo come **grado entrante** il numero di archi $(x, v), \forall x \in V(G)$ e come **grado uscente** il numero di archi $(v, x), \forall x \in V(G)$

Esempio:

- Nel seguente grafo non diretto, si ha che $deg(4) = 2$



- Nel seguente grafo diretto, il grado uscente e il grado entrante di 1 sono ripettivamente pari a 1 e 2, dunque si ha che $deg(1) = 3$



Theorem 1. Somma dei gradi di un grafo

Dato un grafo $G = (V, E)$ avente n vertici, dunque $|V(G)| = n$, e m archi, dunque $|E(G)| = m$, si ha che:

$$\sum_{v \in V(G)} deg(v) = 2m$$

Dimostrazione:

- Poiché ogni arco $e \in E(G)$ è incidente a due vertici $v_i, v_j \in V(G) \mid v_i \neq v_j$, incrementando di 1 il grado di entrambi i vertici. Di conseguenza, si vede facilmente che:

$$\sum_{v \in V(G)} deg(v) = 2m$$

□

Definition 6. Matrice di adiacenza

Sia $G = (V, E)$ un grafo avente n vertici, dunque $|V(G)| = n$. Definiamo come **matrice di adiacenza** una matrice $M \in Mat_{n \times n}(\{0, 1\})$ tale che:

$$m_{i,j} = \begin{cases} 1 & \text{se } (v_i, v_j) \in E(G) \\ 0 & \text{se } (v_i, v_j) \notin E(G) \end{cases}$$

Proposition 2. Costi della matrice di adiacenza

Sia $G = (V, E)$ dove $|V(G)| = n$ e sia $M \in Mat_{n \times n}(\{0, 1\})$ la sua matrice di adiacenza.

Il **costo spaziale** di tale matrice è $O(n^2)$, mentre il **costo computazionale** delle sue operazioni risulta essere:

- Verificare se $(v_i, v_j) \in E(G)$: $O(1)$
- Trovare tutti gli adiacenti a v_i: $O(n)$
- Aggiungere o rimuovere $(v_i, v_j) \in E(G)$: $O(1)$

Dimostrazione:

- Poiché $M \in Mat_{n\times n}(\{0,1\})$, si vede facilmente che il suo costo spaziale sia $O(n^2)$
- Inoltre, poiché $(v_i, v_j) \in E(G) \iff m_{i,j} = 1$, è sufficiente leggere il valore dell'entrata $m_{i,j}$ per verificare se $(v_i, v_j) \in E(G)$, rendendo quindi il costo pari a $O(1)$.

  Per trovare tutti gli adiacenti di un vertice v_i, dunque, è sufficiente leggere il valore delle entrate $m_{i,k}, \forall k \in [0, n]$, rendendo il costo pari a $O(n)$.
- Nel caso in cui si voglia aggiungere o rimuovere un arco $(v_i, v_j) \in E(G)$, se il grafo è diretto sarà necessario modificare l'entrata $m_{i,j}$, rendendo il costo pari a $O(1)$, mentre se il grafo non è diretto sarà necessario modificare l'entrata $m_{i,j}$ e $m_{j,i}$, rendendo il costo pari a $2 \cdot O(1) = O(1)$

□

Definition 7. Liste di adiacenza

Sia $G = (V, E)$ un grafo avente n vertici, dunque $|V(G)| = n$. Definiamo come **liste di adiacenza** l'insieme di liste $L_0, \dots, L_n$ dove $\forall x \in V(G)$ si ha che:

$$L_x := [v \in V(G) \mid (x, v), (v, x) \in E(G)]$$

Se G é un **grafo diretto**, definiamo come **liste di entrata** l'insieme di liste $L_0^{in}, \dots, L_n^{in}$ e come **liste di uscita** l'insieme di liste $L_0^{out}, \dots, L_n^{out}$ dove $\forall x \in V(G)$ si ha che:

$$L_x^{in} := [v \in V(G) \mid (v, x) \in E(G)]$$

$$L_x^{out} := [v \in V(G) \mid (x, v) \in E(G)]$$

Proposition 3. Costi delle liste di adiacenza

Sia $G = (V, E)$ dove $|V(G)| = n$ e siano $L_0, \dots, L_n$ le sue liste di adiacenza.

Il **costo spaziale** necessario per tutte le liste è $O(n + m)$, dove $|E(G)| = m$, mentre il **costo computazionale** delle sue operazioni risulta essere:

- Verificare se $(v_i, v_j) \in E(G)$: $O(deg(v_i))$
- Trovare tutti gli adiacenti a v_i: $O(deg(v_i))$
- Aggiungere o rimuovere $(v_i, v_j) \in E(G)$: $O(deg(v_i))$

Dimostrazione:

- Nel caso in cui G sia un grafo, poiché $(v_i, v_j) \in E(G) \implies (vj, v_i) \in E(G)$, si ha che $|L_i| = deg(v_i), \forall v_i \in V(G)$. Di conseguenza, il costo spaziale per tutte le liste corrisponderà a:

$$O\left(\sum_{v \in V(G)} deg(v)\right) = O(2m) = O(m)$$

Inoltre, poiché sono necessari n puntatori ognuno facente riferimento alla testa di una lista di adiacenza, il costo spaziale finale pari a $O(n+m)$

- Nel caso in cui G sia un grafo diretto, si ha che $|L_i^{in}| = deg_{in}(v_i) \leq deg(v_i)$ e $|L_i^{out}| = deg_{out}(v_i) \leq deg(v_i)$, dunque il costo spaziale di entrambe le liste corrisponde $O(deg(v_i))$. Di conseguenza, il costo spaziale per tutte le liste corrisponderà a:

$$O\left(\sum_{v \in V(G)} 2deg(v)\right) = O(4m) = O(m)$$

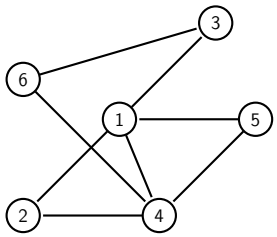
Inoltre, poiché sono necessari $2n$ puntatori ognuno facente riferimento alla testa di una lista di entrata o di uscita, il costo spaziale finale pari a $O(2n+2m) = O(n+m)$

- Poiché ognuna delle tre operazioni nel caso peggiore richiede di scorrere l'intera lista di adiacenza, di entrata o di uscita, il costo computazionale di ognuna di esse sarà $O(deg(v_i))$

□

Esempi:

1. - Consideriamo il seguente grafo

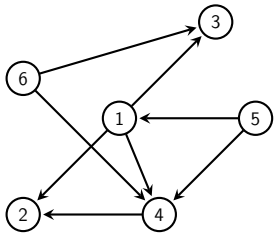


   - La sua rappresentazione tramite matrice di adiacenza e liste di adiacenza corrisponderà a

	1	2	3	4	5	6
1	0	1	1	1	1	0
2	1	0	0	1	0	0
3	1	0	0	0	0	1
4	1	1	0	0	1	1
5	1	0	0	1	0	0
6	0	0	1	1	0	0

$1 \to [2, 4, 3, 5]$
$2 \to [1, 4]$
$3 \to [6, 1]$
$4 \to [2, 1, 5]$
$5 \to [1, 4]$
$6 \to [4, 3]$

2. - Consideriamo il seguente grafo



- La sua rappresentazione tramite matrice di adiacenza e liste di adiacenza corrisponderà a

	1	2	3	4	5	6
1	0	1	1	1	0	0
2	0	0	0	0	0	0
3	0	0	0	0	0	0
4	0	0	0	0	0	0
5	1	0	0	1	0	0
6	0	0	1	1	0	0

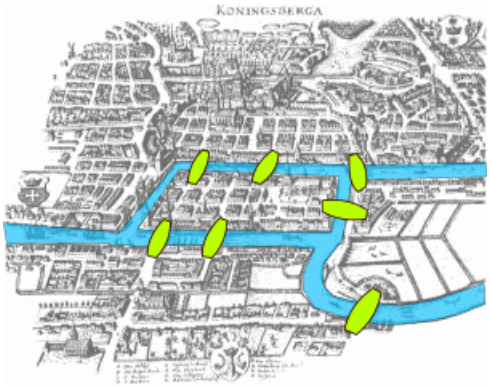
Entrata
$1 \to [5]$
$2 \to [1, 4]$
$3 \to [6, 1]$
$4 \to [2, 1, 5]$
$5 \to [\,]$
$6 \to [\,]$

Uscita
$1 \to [2, 4, 3]$
$2 \to [\,]$
$3 \to [\,]$
$4 \to [\,]$
$5 \to [1, 4]$
$6 \to [4, 3]$

1.2 Passeggiate, cicli e cammini

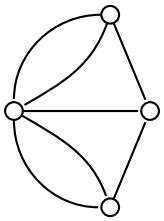
Lo studio della teoria dei grafi deriva da un problema all'apparenza semplice, seppur richiedente uno studio dettagliato. Tale problema corrisponde al **problema dei sette ponti di Königsberg**:

- Nella città di Königsberg ci sono sette ponti posizionati nel seguente modo:



  Vogliamo sapere se sia possibile effettuare una passeggiata per la città passando per tutti i ponti tornando al punto di partenza senza mai passare due volte sullo stesso ponte.

- A risolvere il problema fu Eulero nel 1736, provando che non sia possibile effettuare un tale tipo di passeggiata. Nella sua dimostrazione, Eulero modellò il problema come un multigrafo, dando origine alla teoria dei grafi:



- In seguito, vedremo la dimostrazione data da Eulero tramite il suo teorema generale

Definition 8. Passeggiata

Dato un grafo $G = (V, E)$, definiamo come **passeggiata** una sequenza alternata di vertici $v_1, \dots, v_k \in V(G)$ ed archi $e_1, \dots, e_k \in E(G)$, dove $e_i = (v_{i-1}, v_i)$.

In altre parole, definiamo la seguente sequenza come passeggiata:

$$v_0 e_1 v_1 \dots v_{i-1} e_i v_i \dots v_{k-1} e_k v_k$$

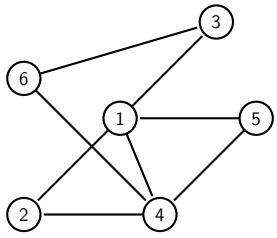
Definition 9. Traccia e Cammino

Sia $G = (V, E)$ un grafo. Definiamo una passeggiata in G come:

- **Traccia** se tale passeggiata **non contiene archi ripetuti**
- **Cammino** se tale passeggiata **non contiene vertici ripetuti** (e di conseguenza neanche archi ripetuti)

Esempi:

- Consideriamo il seguente grafo



- La seguente sequenza è una passeggiata su tale grafo

$$1 - (1,2) - 2 - (2,4) - 4 - (4,5) - 5 - (5,4) - 4 - (4,1) - 1$$

- La sequente sequenza è una traccia su tale grafo

$$4 - (4,5) - 5 - (5,1) - 1 - (1,2) - 2 - (2,4) - 4 - (4,1) - 1$$

- La sequente sequenza è un cammino su tale grafo

$$4 - (4,5) - 5 - (5,1) - 1 - (1,2) - 2 - (2,4) - 4$$

Definition 10. Visita di un vertice

Sia $G = (V, E)$ un grafo. Dato $v \in V(G)$, definiamo un vertice $v' \in V(G)$ **visitabile** da v, indicato come $v_1 \to v_2$, se esiste una passeggiata da v_1 a v_2

Observation 1

Dato un grafo $G = (V, E)$ si ha che:

$$\exists \text{ una passeggiata } \mid x \to y \text{ in } G \implies \exists \text{ un cammino } \mid x \to y \text{ in } G$$

Definition 11. Grafo connesso e fortemente connesso

Sia $G = (V, E)$ un grafo. Definiamo G come **connesso** se

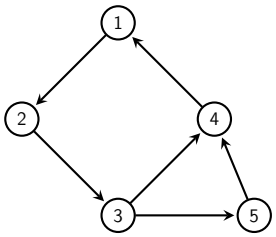
$$\forall v_1, v_2 \in V(G), \exists \text{ un cammino } \mid v_1 \to v_2$$

Definiamo invece G come **fortemente connesso** se

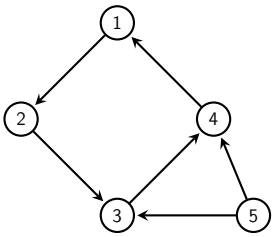
$$\forall v_1, v_2 \in V(G), \exists \text{ due cammini } \mid v_1 \to v_2, v_2 \to v_1$$

Esempio:

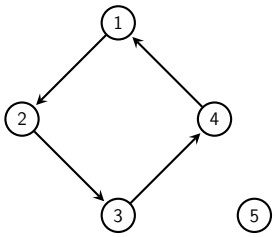
Fortemente connesso



Connesso



Non connesso



Definition 12. Passeggiata chiusa ed aperta

Sia $G = (V, E)$ un grafo. Definiamo una passeggiata $v_0 e_1 \dots e_k v_k$ su G come **chiusa** se $v_0 = v_k$, altrimenti essa viene definita **aperta**

Definition 13. Passeggiata Hamiltoniana

Sia $G = (V, E)$ un grafo. Definiamo una passeggiata come **hamiltoniana** se tale passeggiata contiene tutti i vertici in $V(G)$ ed ogni vertice è presente una sola volta.

In altre parole, una passeggiata hamiltoniana è un cammino contenente tutti i vertici in $V(G)$

Definition 14. Passeggiata Euleriana

Sia $G = (V, E)$ un grafo. Definiamo una passeggiata come **euleriana** se tale passeggiata contiene tutti gli archi in $E(G)$ ed ogni arco è presente una sola volta.

In altre parole, una passeggiata euleriana è una traccia contenente tutti gli archi in $E(G)$

Theorem 4. Teorema di Eulero

Dato un grafo $G = (V, E)$, esiste una passeggiata euleriana chiusa in G se e solo se G è connesso e il grado di ogni vertice è pari:

$$\exists \text{ passeggiata euleriana chiusa in } G \iff \begin{cases} \forall v_1, v_2 \in V(G), \exists v_1 \to v_2 \\ \forall v \in V(G), \exists k \in \mathbb{Z} \mid deg(v) = 2k \end{cases}$$

Dimostrazione (implicazione $\Longleftarrow$ omessa):

- Supponiamo per assurdo che esista una passeggiata euleriana chiusa in G e che $\exists v \in V \mid deg(v) = 2k + 1, \exists k \in \mathbb{Z}$, ossia che esista un vertice avente grado dispari.
- In tal caso, una volta effettuata la $2k + 1$ esima visita su v utilizzando ogni volta un diverso arco incidente ad esso, non sarebbe possibile raggiungere un altro vertice $x \in V(G) \mid (v, x) \in E(G)$ senza necessariamente riutilizzare uno degli archi incidenti a v, contraddicendo l'ipotesi per cui tale passeggiata sia euleriana.
- Inoltre, nel caso particolare in cui x sia il vertice iniziale della passeggiata, se $deg(v) = 2k + 1$ non sarebbe possibile raggiungere x come vertice finale della passeggiata, contraddicendo l'ipotesi per cui la passeggiata sia chiusa.
- Supponiamo quindi per assurdo che esista una passeggiata euleriana chiusa in G e che G non sia connesso. In tal caso, ne seguirebbe automaticamente che tale passeggiata non possa essere euleriana, poiché esisterebbe un vertice sconnesso avente un arco non utilizzabile nella passeggiata

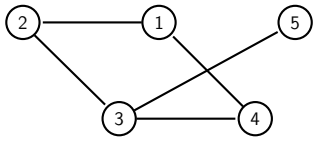
□

Definition 15. Ciclo

Sia $G = (V, E)$ un grafo. Definiamo come **ciclo** una **passeggiata chiusa** dove solo il **primo** e l'**ultimo** vertice sono ripetuti.

Esempio:

- Consideriamo il seguente grafo



- La seguente passeggiata è un ciclo in G

$$1(1,2)2(2,3)3(3,4)4(4,1)1$$

Algorithm 1. Trovare un ciclo

Sia $G = (V, E)$ un grafo non diretto dove $deg(v) \geq 2, \forall v \in V(G)$. Il seguente algoritmo restituisce, se esistente, un ciclo in G.

Il **costo computazionale** di tale algoritmo corrisponde a:

- $O(n)$ se G sia rappresentato tramite liste di adiacenza
- $O(n^2)$ se G sia rappresentato tramite matrice di adiacenza

dove $|V(G)| = n$

Algorithm 1: Trovare un ciclo in un grafo non diretto con grado minimo 2

Input:
G: grafo non diretto dove $deg(v) \geq 2, \forall v \in V(G)$
Output:
Ciclo in G
Function `findCycle(G):`

```
    x := x ∈ V(G);
    Vis := {x};
    y := z ∈ V(G) | (x, y) ∈ E(G);
    while y ∉ Vis do
        Vis.add(y);
        y := w ∈ V(G) | (y, w) ∈ E(G), w ≠ Vis[Vis.length - 2];
    end
    return Vis[Vis.index(y) : Vis.length-1];
end
```

Dimostrazione correttezza dell'algoritmo:

- Preso un vertice iniziale $x \in V$ qualsiasi, l'algoritmo costruisce una passeggiata scegliendo ad ogni iterazione un vertice non già visitato, in modo che esso non possa essere ripetuto. L'insieme `Vis` contiene i vertici visitati durante la passeggiata.
- In particolare, la condizione interna al while $w \neq$ `Vis[Vis.length -2]` impedisce all'algoritmo di selezionare il penultimo vertice interno alla passeggiata, impedendo che essa possa tornare indietro e conseguentemente impedendo che venga riutilizzato un vertice precedente.
- Il ciclo while viene terminato quando $y \in Vis$, implicando che y sia un vertice già visitato. Di conseguenza, lo slice `Vis[Vis.index(y) : Vis.length-1]`, corrisponderà ad un ciclo $y, w_0, \ldots, w_k, y$.

□

Dimostrazione costo dell'algoritmo:

- Se ogni vertice successivo selezionato non è mai stato visitato, il ciclo while verrà eseguito un massimo di n volte, dando vita a due scenari:

– Se G fosse rappresentato tramite matrice di adiacenza, il costo della ricerca di un vertice adiacente a y risulta essere $O(n)$, di conseguenza il costo del ciclo while sarà $n \cdot O(n) = O(n^2)$

– Se G fosse rappresentato tramite liste di adiacenza, il vertice adiacente selezionato sarà necessariamente $L_y[0]$, nel caso in cui $L_y[0] \notin$ `Vis`, oppure $L_y[1]$, nel caso in cui $L_y[0] \in$ `Vis`.

Di conseguenza, il costo della ricerca di un vertice adiacente a y risulta essere $2 \cdot O(1) = O(1)$, rendendo il costo del while pari a $n \cdot O(1) = O(n)$

□

1.3 DAG e Ordinamento topologico

Definition 16. Grafo diretto aciclico (DAG)

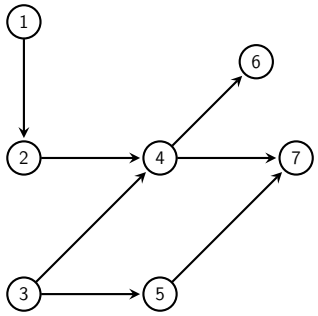
Sia $G = (V, E)$ un grafo. Definiamo G come **grafo diretto acicliclo (DAG)** se non esistono cicli in G

Definition 17. Ordinamento topologico

Sia $G = (V, E)$ un grafo diretto. Dati i suoi vertici $V(G) = \{v_0, \ldots, v_n\}$, definiamo come **ordinamento topologico** un ordinamento di tali vertici in cui ogni vertice viene prima di tutti i vertici raggiungibili da un suo arco uscente

Esempio:

- Consideriamo il seguente grafo



- Le due seguenti sequenze di vertici sono due ordinamenti topologici possibili di tale grafo:
  – Precedenza ai vertici più in alto: $1, 2, 3, 4, 6, 5, 7$
  – Precedenza ai vertici più a sinistra: $3, 1, 2, 4, 5, 6, 7$

Theorem 5

Dato un grafo diretto $G = (V, E)$, si ha che:

$$\exists \text{ ordinamento topologico in } G \iff \nexists \text{ ciclo in } G$$

Dimostrazione:

- Supponiamo per assurdo che esista un ordinamento topologico in G e che esista un ciclo $v_0 e_1 v_1 e_2 \dots e_k v_0$ in G, implicando che v_1 sia un vertice uscente di v_0.

  In tal caso, verrebbe contraddetta l'ipotesi per cui in G esista un ordinamento topologico, poiché v_0 verrebbe sia prima di v_1 sia dopo v_1. Di conseguenza, l'unica possibiltà è che non esista alcun ciclo in G

- Viceversa, supponiamo per assurdo che non esista un ciclo in G e che non esista un ordinamento topologico in G, implicando che esista un vertice $v \in V(G)$ tale che v sia raggiungibile da un arco uscente di un vertice v' a sua volta raggiungibile da un arco uscente v.

  Di conseguenza, si avrebbe che $v \to v' \to v$, contraddicendo l'ipotesi per cui in G non esistano cicli, dunque l'unica possibilità è che in G esista un ordinamento topologico.

□

Observation 2

Dato un grafo diretto aciclico $G = (V, E)$, si ha che:

- $\exists v \in V(G) \mid deg_{in}(v) = 0$
- $\exists v' \in V(G) \mid deg_{out}(v) = 0$

Dimostrazione:

- Supponiamo per assurdo che G sia un DAG e che $\nexists v \in V(G) \mid deg_{in}(v) = 0$.
- Poiché G è aciclico, esiste un ordinamento topologico $v_0, \dots, v_n$ in G, dove $|V(G)| = n$. Tuttavia, poiché $deg_{out}(v_n) \neq 0$, ne segue che $\exists v_k \in V(G) \mid k \in [0, n-1]$ tale che $(v_n, v_k) \in E(G)$, implicando che $v_k \to v_n \to v_k$, contraddicendo l'ipotesi per cui G sia aciclico
- Analogamente, poiché $deg_{in}(v_0) \neq 0$, ne segue che $\exists v_j \in V(G) \mid j \in [1, n]$ tale che $(v_j, v_0) \in E(G)$, implicando che $v_j \to v_0 \to v_j$, contraddicendo ancora l'ipotesi per cui G sia aciclico
- Di conseguenza, l'unica possibilità è che $deg_{in}(v_0) = 0$ e $deg_{out}(v_n) = 0$

□

Algorithm 2. Trovare un ordinamento topologico

Sia $G = (V, E)$ un DAG. Il seguente algoritmo restituisce un possibile ordinamento topologico di G

Il **costo computazionale** di tale algoritmo è $O(n^2+nm)$, dove $|V(G)| = n$ e $|E(G)| = m$

Algorithm 2: Trovare un ordinamento topologico in un DAG

Input:
G: grafo diretto aciclico
Output:
Ordinamento topologico in G
Function `findTopologicalSorting(G)`:

```
    List L := ∅;
    while V(G) ≠ ∅ do
        v := v ∈ V(G) | deg_in(v) = 0;
        L.head_insert(v);
        G.remove(v);
    end
    return L;
end
```

Dimostrazione correttezza algoritmo:

- Siano $G_0, \ldots, G_k$ le istanze del grafo G ad ogni iterazione del ciclo while. Poiché G è aciclico, ne segue che anche $G_0, \ldots, G_k$ siano aciclici, poiché rimuovere vertici non crea cicli in tali grafi.
- Per l'osservazione precedente, dunque $\forall i \in [0, n]$ si ha che $\exists v_i \in V_i(G_i) \mid deg_{in}(v_i) = 0 \mid$ implicando che ad ogni iterazione esista sempre un vertice selezionabile finché $V(G) \neq \emptyset$. Di conseguenza, si ha che $k = |V(G)|$.
- Notiamo inoltre che, ad ogni rimozione di un vertice $x \in V(G)$, il grado di tutti i vertici $x' \in V(G) \mid (x, x') \in E(G)$ venga decrementato di uno.
- Siano quindi $v_0, \ldots, v_k$ i vertici selezionati e rimossi ad ogni iterazione. Supponiamo per assurdo che $L := v_0, \ldots, v_k$ non sia un ordinamento topolgico, implicando che $\exists v_i, v_j \in L$ tali che $(v_i, v_j) \in E(G)$ e v_j venga prima di v_i nell'ordinamento. In tal caso, l'algoritmo avrebbe sbagliato a selezionare v_j prima di v_i, poiché $(v_i, v_j) \in E(G) \implies deg_{in}(v_j) > 0$.
- Di conseguenza, l'unica possibilità è che v_j venga selezionato dopo v_i, implicando quindi che L sia un ordinamento topologico

□

Dimostrazione costo dell'algoritmo:

- Come dimostraato nella correttezza dell'algoritmo, il ciclo while viene iterato sempre $|V(G)| = n$ volte.

- In entrambe le tipologie di rappresentazione di G, l'inserimento in testa nella lista risulta avere un costo pari a $O(1)$
- Nel caso in cui G sia rappresentato tramite matrice di adiacenza, nel caso peggiore sarebbe necessario scorrere ogni singola entrata della matrice durante la selezione del prossimo vertice, risultando quindi in un costo pari a $O(n^2)$. essario rimuovere tutti gli $|E(G)| = m$ archi dalla lista di uscita di v e tutti gli $|E(G)| = m$ archi distribuiti nelle liste di entrata degli altri $n-1$ vertici, risultando quindi in un costo pari a $O(n) + O(2m) = O(n+m)$

  Di conseguenza, in tal caso il costo finale del ciclo while risulta essere $n \cdot O(n^2) = O(n^3)$
- Nel caso in cui G sia rappresentato tramite liste di adiacenza, invece, nel caso peggiore sarebbe necessario controllare il grado d'entrata di ogni vertice durante la selezione del prossimo vertice,risultando quindi in un costo pari a $O(n)$.

  Inoltre, nel caso peggiore esiste un unico nodo $v \in V(G) \mid \forall v' \in V(G), \exists (v, v') \in E(G)$, implicando che sia necessario rimuovere tutti gli $|E(G)| = m$ archi dalla lista di uscita di v e tutti gli $|E(G)| = m$ archi distribuiti nelle liste di entrata degli altri $n-1$ vertici, risultando quindi in un costo pari a $O(n) + O(2m) = O(n+m)$

  Di conseguenza, in tal caso il costo finale del ciclo while risulta essere $n \cdot O(n+m) = O(n^2 + nm)$

□

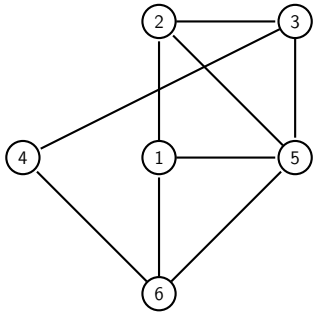
1.4 Depth-First Search (DFS)

Definition 18. Depth-First Search (DFS)

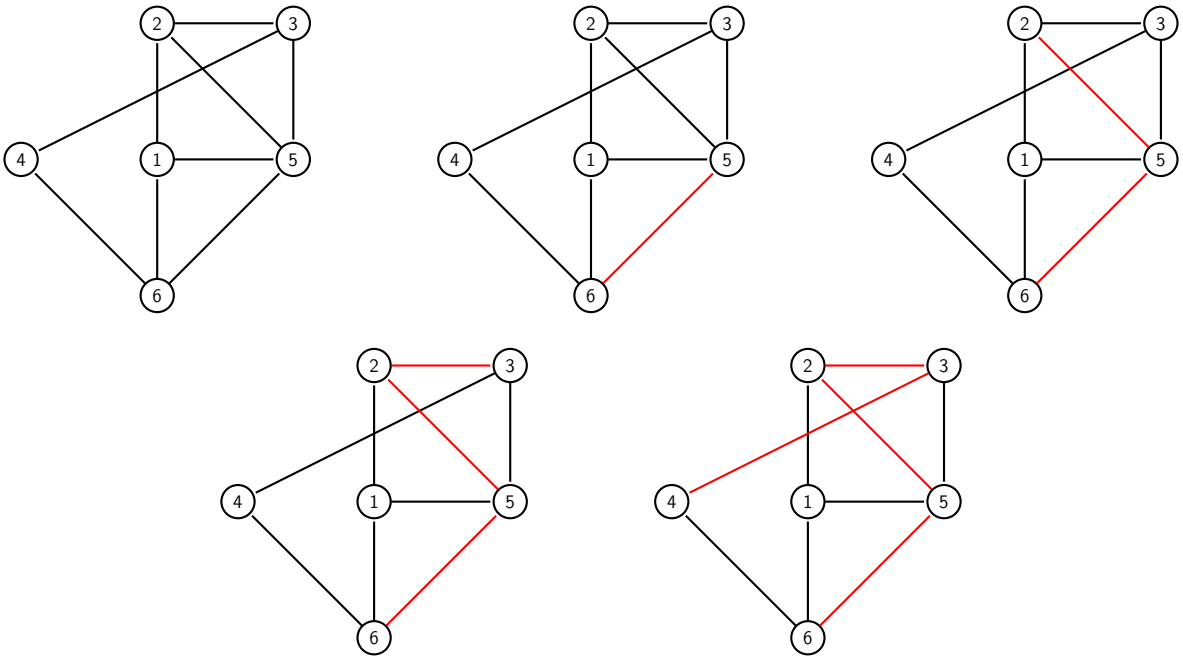
Sia $G = (V, E)$ un grafo. Dato un vertice iniziale $x \in V(G)$, detto **radice**, definiamo come **depth-first search (DFS)** un **criterio di visita** su G basato sul procedere **in profondità**, ossia dando precedenza ai vertici più lontani dalla radice, raggiungendo ogni vertice **una ed una sola volta**, tornando al vertice precedente se e solo se non è più possibile procedere in profondità tramite il vertice attuale, ossia quando tutti i vertici adiacenti sono già stati visitati

Esempio:

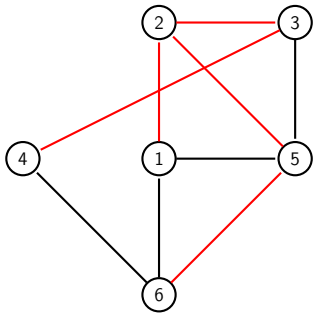
- Consideriamo il seguente grafo.



- Scelto 6 come radice, selezioniamo casualmente uno dei tre archi incidenti a 6, ripetendo tale procedimento finché non sia più possibile scendere in profondità



- Una volta raggiunto il vertice 4, non è più possibile scendere il profondità poiché tutti i vertici adiacenti a 4 sono già stati visitati. Di conseguenza, la ricerca DFS tornerà al vertice precedente, ossia il vertice 3. Tuttavia, anche per tale vertice è impossibile procedere in profondità. Di conseguenza, la ricerca DFS tornerà al vertice precedente, ossia il vertice 2. A questo punto, la ricerca DFS è in grado di procedere in profondità poiché il vertice 1 non è ancora stato visitato



- A questo punto, poiché ogni vertice del grafo è già stato visitato, la ricerca non sarà più in grado di procedere in profondità. Dunque, ad ogni iterazione essa tornerà continuamente al vertice precedente, fino a raggiungere la radice stessa, per poi concludersi. L'ordine finale di visita, dunque, corrisponde a $6, 5, 2, 3, 4, 1$

Algorithm 3. Depth-first Search

Sia $G = (V, E)$ un grafo e sia $x \in V(G)$ un vertice. Il seguente algoritmo effettua una DFS, restituendo l'insieme di vertici visitabili dalla radice x

Il **costo computazionale** di tale algoritmo corrisponde a $O(n^2)$, dove $|V(G)| = n$

Algorithm 3: Depth-first Search

Input:
G: grafo, $x : x \in V(G)$
Output:
Vertici visitabili da x
Function `DFS(G,x)`:

```
Vis := {x};
Stack S := ∅;
S.push(x);
while S ≠ ∅ do
    y := S.top();
    if ∃z ∈ V(G) | (y, z) ∈ E(G), z ∉ Vis then
        S.push(z);
        Vis.add(z);
    else
        S.pop();
    end
end
return Vis;
end
```

Dimostrazione correttezza algoritmo:

- Sia $y \in V(G)$ un vertice visitabile da x tramite una passeggiata. Di conseguenza, esiste anche un cammino $v_0e_1v_1 \dots v_{k-1}e_kv_k$ tale che $x \to y$, implicando quindi che $x := v_0$ e $y := v_k$.
- Supponiamo per assurdo che venga raggiunta l'iterazione del while per cui $S = \emptyset$ e che $y \notin$ `Vis`.
- Sia v_i il vertice di tale cammino avente indice maggiore dove $v_i \in$ `Vis`, implicando che $v_{i+1} \notin$ `Vis`. Se tale vertice esistesse, esso verrebbe tolto dallo stack prima che il vertice v_{i+1} sia visitato dall'algoritmo, poiché $v_{i+1} \notin$ `Vis`, $\exists(v_i, v_{i+1}) \in E(G)$ e $v_{i+1} \neq v_j, \forall j \in [0, i]$, implicando quindi che l'algoritmo abbia sbagliato l'esecuzione.
- Di conseguenza, l'unica possibilità è che una volta raggiunta l'iterazione del while per cui $S = \emptyset$ si abbia che $y \in$ `Vis`

□

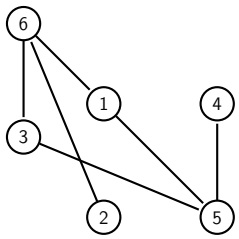
Dimostrazione costo dell'algoritmo:

- Nel caso peggiore in cui $\forall v \in V(G) - \{x\}$ si abbia che $x \to v$, il ciclo while verrebbe eseguito un totale di $2n - 1$ volte, poiché ogni vertice, eccetto la radice, verrebbe aggiunto e rimosso dallo stack 2 volte, dando vita a due scenari:
  - Se G fosse rappresentato attraverso una matrice di adiacenza, la ricerca del vertice successivo ad ogni iterazione avrebbe un costo pari a $O(n)$, poiché potenzialmente verrebbe analizzata l'intera riga associata al vertice attuale, rendendo il costo del ciclo while pari a $O((2n-1)n) = O(2n^2 - n) = O(n)$

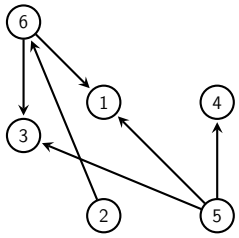
– Se G fosse rappresentato attraverso liste di adiacenza, la ricerca del vertice successivo ad ogni iterazione avrebbe un costo pari a $O(n-1) = O(n)$, poiché,assumendo il caso peggiore, la sua lista di adiacenza associata al vertice attuale conterrebbe ogni vertice del grafo eccetto se stesso, rendendo l'intera riga, rendendo potenzialmente necessario scorrere l'intera lista. Di conseguenza, il costo del ciclo while pari sarebbe pari a $O((2n-1)n) = O(2n^2 - n) = O(n)$

Esempi:

1. • Eseguendo una DFS partendo da 2, l'insieme di vertici ottenuto corrisponde a $2, 6, 3, 5, 1, 4$



2. • Eseguendo una DFS partendo da 2, l'insieme di vertici ottenuto corrisponde a $2, 6, 3, 1$



Observation 3

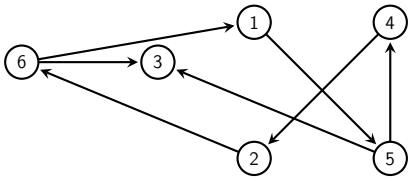
Sia $G = (V, E)$ un grafo. Dato un vertice $x \in V(G)$, l'insieme di archi e vertici utilizzati da una DFS avente x come radice costituisce un **sottografo** $H = (V', E')$ **connesso ed aciclico**

Definition 19. Arborescenza

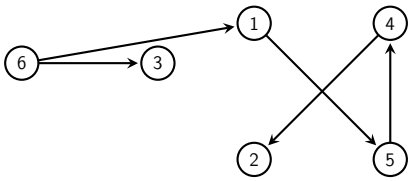
Sia $G = (V, E)$ un grafo diretto. Dato un vertice $x \in V(G)$, definiamo come **arborescenza** il sottografo $H = (V', E')$ connesso ed aciclico ottenuto effettuando una DFS avente x come radice

Esempio:

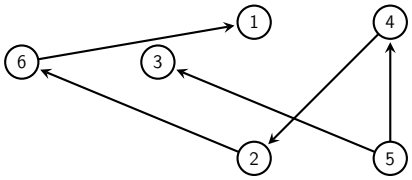
• Consideriamo il seguente grafo:



- L'arborescenza ottenuta effettuando una DFS avente come radice il vertice 6 corrisponde a



- L'arborescenza ottenuta effettuando una DFS avente come radice il vertice 5 corrisponde a



Problem 1

Una fabbrica ha diviso un processo di produzione in n fasi. Tra ogni coppia di vasi vi è una dipendenza, ossia una di essere deve essere completata prima dell'altra. Vogliamo trovare una possibile programmazione (se esistente) del processo di produzione rispettante tutte le dipendenze.

Soluzione:

- Siano $V(G) := x_1, \ldots, x_n$ le fasi del processo di produzione. Modelliamo le dipedenze tra ogni fase come degli archi diretti, dove $\forall x, x' \in V(G) \exists (x, x') \in E(G) \iff x'$ dipende da x
- A questo punto, possiamo tradurre la richiesta del trovare una possibile programmazione nel trovare un ordine topologico di G. Tuttavia, per poter trovare tale ordine, è prima necessario accertarsi che G sia aciclico poiché, per dimostrazione precedente, si ha che $\exists$ ordine topologico in $G \iff \nexists$ ciclo in G
- Per determinare se G sia aciclico, possiamo eseguire su ogni vertice $v \in V(G)$ una versione modificata della DFS dove non appena viene trovato un nodo già visitato viene automaticamente decretatata l'esistenza di un ciclo.
- Nel caso in cui nessuna delle varie DFS eseguite abbia rilevato la presenza di un ciclo in G, possiamo utilizzare l'algoritmo 2 `findTopologicalSorting(G)` per trovare un ordine valido. In caso contrario, non sa

Algorithm 4. Depth-first Search (Ottimizzato)

Sia $G = (V, E)$ un grafo con liste di adiacenza e sia $x \in V(G)$ un vertice. Il seguente algoritmo effettua una DFS, restituendo l'insieme di vertici visitabili dalla radice x

Il **costo computazionale** di tale algoritmo corrisponde a $O(n+m)$, dove $|V(G)| = n$ e $|E(G)| = m$

Algorithm 4: Depth-first Search (Ottimizzato)

Input:
G: grafo, $x \in V(G)$: radice
Output:
Vertici visitabili da x

```
Function DFS_Optimized(G,x):
    Vis := {x};
    Stack S := ∅;
    S.push(x);
    while S ≠ ∅ do
        y := S.top();
        while y.adiacenti ≠ ∅ do
            z = y.adiacenti[0];
            y.remove(0);
            if z ∉ Vis then
                Vis.add(z);
                S.push(z);
                break;
            end
        end
        if y == S.top() then
            S.pop();
        end
    end
    return Vis;
end
```

Dimostrazione costo dell'algoritmo:

- Analogamente alla versione non ottimizzata, nel caso in cui $\forall v \in V(G), \exists (x, v) \in E(G)$, il ciclo while verrà eseguito $2n - 1$ volte.
- Tuttavia, a differenza della versione non ottimizzata, ogni volta che un vertice viene analizzato come potenziale vertice successivo, esso viene rimosso dalla lista di adiacenza del vertice attuale, diminuendo la dimensione di quest'ultima, implicando che il numero totale di controlli effettuati corrisponda esattamente al numero di archi presenti nel grafo, ossia $|E(G) = m|$
- Nel caso particolare il cui il grafo sia diretto, è possibile considerare solo la lista di uscita di ogni vertice attuale, rendendo il ragionamento analogo alla lista di adiacenza completa
- Di conseguenza, per entrambe le tipologie di grafo, il costo totale del ciclo while sarà $O(2n - 1 + m) = O(n + m)$

□

Algorithm 5. Esistenza ciclo contenente arco specifico

Sia G un grafo rappresentatoa tramite liste di adiacenza. Dato un arco $f \in E(G)$, il seguente algoritmo stabilisce se esista un ciclo in G tale che f appartenga a tale ciclo.

Il **costo computazionale** di tale algoritmo risulta essere $O(n+m)$, dove $|V(G)| = n$ e $|E(G)| = m$

Algorithm 5: Stabilire se esista un ciclo in un grafo G contenente un arco $f \in E(G)$

Input:
G: grafo a liste di adiacenza, $f : f \in E(G)$
Output:
True se esiste un ciclo, False altrimenti
Function `findCycleWithF(G: grafo, f: arco):`

```
    x := f.head;
    y := f.tail;
    G.remove(f);
    Vis := DFS(G, x);
    if y ∈ Vis then
        return True;
    else
        return False;
    end
end
```

Dimostrazione correttezza algoritmo:

- Poiché può esistere un ciclo $xe_1 \dots e_k y f x$ se e solo se esiste un cammino $xe_1 \dots e_k y$ e poiché tale cammino può esistere se e solo se $x \to y \iff y \in$ `Vis`, ne segue automaticamente che esista un ciclo $xe_1 \dots e_k y f x$ se e solo se $y \in$ `Vis`

□

Dimostrazione costo algoritmo:

- Per poter rimuovere l'arco f dal grafo G, è necessario scorrere la lista di uscita del nodo x e lista di entrata del nodo y, rendendo quindi il costo pari a $O(deg(x)) + O(deg(y)) = O(deg(x) + deg(y))$
- Poiché il costo della DFS è $O(n+m)$ e poiché $deg(x) + deg(y) < m$, ne segue che il costo finale dell'algoritmo sia $O(deg(x) + deg(y)) + O(n+m) = O(n+m)$

□

Definition 20. Tempo di visita e Tempo di chisura

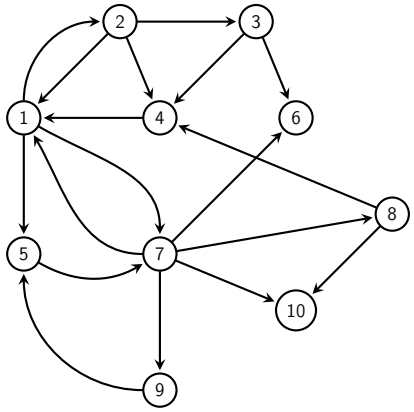
Sia $G = (V, E)$ un grafo e sia C un **contatore** inizializzato a 0 inserito all'interno dell'algoritmo DFS, il quale viene **incrementato ogni volta che viene visitato un nuovo nodo**.

Per ogni vertice $v \in V(G)$, definiamo come **tempo di visita di** v, indicato come $t(v)$, il valore assunto da C nell'istante in cui v viene aggiunto allo stack, e come **tempo di chiusura di** v, indicato come $T(V)$, il valore assunto da C nell'istante in cui v viene rimosso dallo stack.

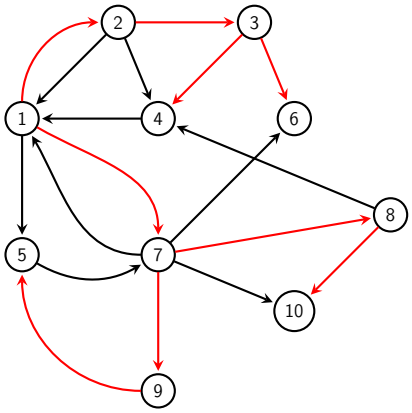
Definiamo inoltre come **intervallo di visita di** v l'intervallo $Int(v) := [t(v), T(v)]$

Esempio:

- Consideriamo il seguente multigrafo:



- Effettuando una DFS avente come radice il vertice 1, una delle possibili arborescenze generate e il suo corrispettivo insieme di intervalli di visita corrisponde a:



v	t(v)	T(v)
1	1	10
2	2	5
3	3	5
4	5	5
5	10	10
6	4	4
7	6	10
8	7	8
9	9	10
10	8	8

Proposition 6

Sia $G = (V, E)$ un grafo. Dato un arco $(u, v) \in E(G)$, dove u è detto **coda** e v è detto **testa**, solo una delle seguenti condizioni è verificata:

- $Int(u) \subseteq Int(v)$
- $Int(u) \supseteq Int(v)$
- $Int(u) \cap Int(v) = \emptyset$

Dimostrazione:

- Suppiamo per assurdo che $t(u) < t(v) < T(u) < T(v)$, ossia che i due intervalli si intersechino, ma nessuno dei due è interamente contenuto dell'altro. Poiché $t(u) < t(v)$, ne segue che u sia stato aggiunto allo stack prima di v. Di conseguenza, è impossibile che u sia stato tolto dallo stack prima di v, contraddicendo l'ipotesi per cui $T(u) < T(v)$.
- Analogamente, dimostriamo che $t(v) < t(u) < T(v) < T(u)$ è un caso impossibile
- Di conseguenza, le uniche possibilità sono:
  - $t(u) < t(v) < T(v) < T(u) \implies Int(u) \supseteq Int(v)$
  - $t(v) < t(u) < T(u) < T(v) \implies Int(u) \subseteq Int(v)$
  - $t(u) < T(u) < t(v) < T(v) \implies Int(u) \cap Int(v) = \emptyset$
  - $t(v) < T(v) < t(u) < T(u) \implies Int(u) \cap Int(v) = \emptyset$

□

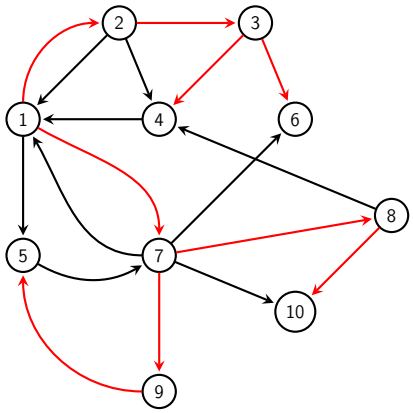
Definition 21. Archi all'indietro, in avanti e di attraversamento

Sia $G = (V, E)$ un grafo e sia $A = (V', E')$ un'arborescenza generata da una DFS su G. Per ogni arco di G **non appartenente all'arborescenza**, dunque $\forall (u, v) \in E(G) \mid (u, v) \notin E'(A)$, definiamo tale arco come:

- **Arco in avanti (forward edge)** se l'intervallo della coda è contenuto in quello della testa, ossia se $Int(u) \subseteq Int(u)$
- **Arco all'indietro (back edge)** se l'intervallo della testa è contenuto in quello della coda, ossia se $Int(u) \supseteq Int(v)$
- **Arco di attraversamento (cross edge)** se l'intervallo della testa non è in relazione con l'intervallo della coda, ossia se $Int(u) \cap Int(v) = \emptyset$

Esempio:

- Riprendiamo l'arborescenza generata nell'esempio precedente



v	t(v)	T(v)
1	1	10
2	2	5
3	3	5
4	5	5
5	10	10
6	4	4
7	6	10
8	7	8
9	9	10
10	8	8

- Classifichiamo quindi gli archi non appartenenti all'arborescenza:
  - L'arco $(2, 1)$ è un back edge, poiché $[2, 5] \subseteq [1, 10]$
  - L'arco $(2, 4)$ è un forward edge, poiché $[2, 5] \supseteq [5, 5]$
  - L'arco $(4, 1)$ è un back edge, poiché $[5, 5] \subseteq [1, 10]$
  - L'arco $(1, 5)$ è un forward edge, poiché $[1, 10] \supseteq [10, 10]$
  - L'arco $(5, 7)$ è un back edge, poiché $[10, 10] \subseteq [6, 10]$
  - L'arco $(7, 1)$ è un back edge, poiché $[6, 10] \subseteq [1, 10]$
  - L'arco $(7, 6)$ è un cross edge, poiché $[6, 10] \cap [4, 4] = \emptyset$
  - L'arco $(7, 10)$ è un foward edge, poiché $[6, 10] \supseteq [8, 8]$
  - L'arco $(8, 4)$ è un cross edge, poiché $[7, 8] \cap [5, 5] = \emptyset$
- Una volta colorati di verde tutti i forward edge e di blu tutti i back edge, la classificazione finale del grafo corrisponde a: