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

17 aprile 2023

Indice

Capitolo 0

Introduzione

Capitolo 1

Teoria dei grafi

1.1 Grafi, vertici e archi

Definition 1. Grafo

Definiamo come **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:

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

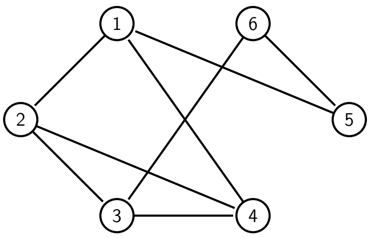
Di conseguenza, in un grafo non sono presenti né **archi ripetuti tra due vertici**, né **cappi**, ossia archi da un vertice in se stesso.

Definition 2. Multigrafo

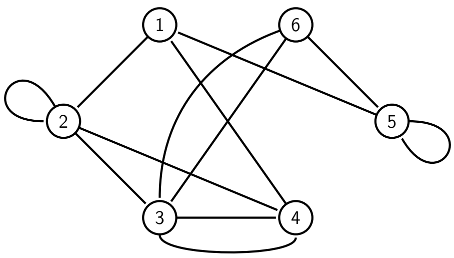
Definiamo come **multigrafo** $G = (V, E)$ è un particolare tipo di grafo dove **sono concessi archi ripetuti e cappi** nell'insieme degli archi E

Esempio:

Grafo



Multigrafo



Definition 3. Incidenza e adiacenza

Sia G 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 un grafo. Definiamo G come **grafo diretto**, o **digrafo**, se i suoi archi possiedono un **orientamento**, ossia se

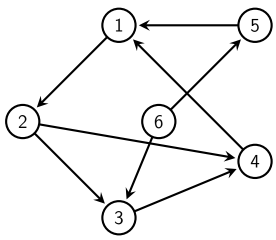
$$(v_1, v_2) \in E(G) \implies (v_2, v_1) \notin E(G)$$

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

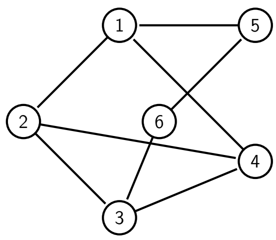
$$(v_1, v_2) \in E(G) \implies (v_2, v_1) \in E(G)$$

Esempio:

Grafo diretto



Grafo non diretto



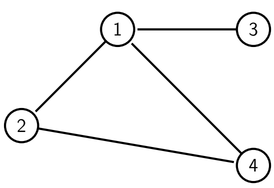
Definition 5. Grado di un vertice

Dato un grafo o multigrafo G e dato $v \in V(G)$, definiamo come **grado** di v, indicato come $deg(v)$, il **numero di archi incidenti** a v_1

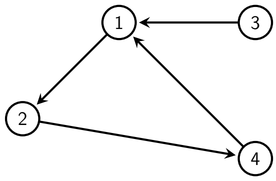
In particolare, se G è **diretto**, definiamo come **grado entrante** di v il numero di archi $(x, v) \in E(G), x \in V(G)$ e come **grado uscente** di v il numero di archi $(v, y) \in E(G), y \in V(G)$

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 tale che $|E(G)| = m$, 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 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 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 un grafo avente n vertici, dunque $|V(G)| = n$. Definiamo come **liste di adiacenza** l'insieme di liste $L_0, \ldots, L_n$ 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 $L_0^{in}, \ldots, L_n^{in}$ e come **liste di uscita** l'insieme di liste $L_0^{out}, \ldots, L_n^{out}$ 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 dove $|V(G)| = n$ e siano $L_0, \ldots, L_n$ 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:

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

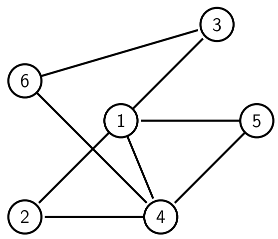
  Inoltre, poiché sono necessari $2n$ puntatori ognuno facente riferimento alla testa di una lista di entrata o di uscita, il costo spaziale finale pari a $O(2n+2m) = O(n+m)$

- 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

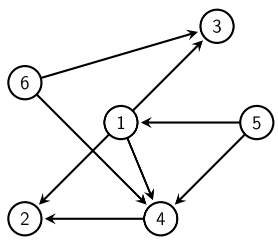


   - La sua rappresentazione tramite matrice di adiacenza e liste di adiacenza corrisponderà a

	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

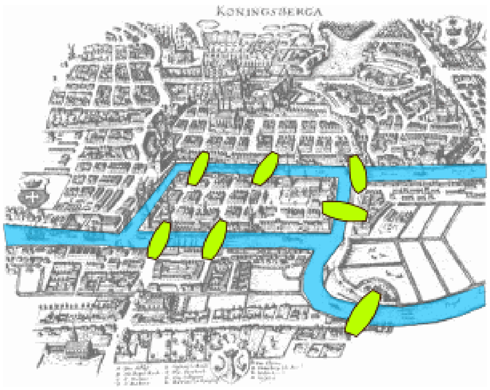
	1	2	3	4	5	6
1	0	1	1	1	0	0
2	0	0	0	0	0	0
3	0	0	0	0	0	0
4	0	0	0	0	0	0
5	1	0	0	1	0	0
6	0	0	1	1	0	0

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

1.1.1 Passeggiate, tracce, cammini e cicli

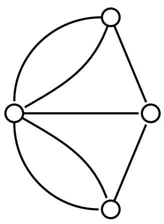
Lo studio della teoria dei grafi deriva da un problema all'apparenza semplice, seppur richiedente un'attenta analisi. Tale problema corrisponde al **problema dei sette ponti di Königsberg**:

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

In altre parole, definiamo la seguente sequenza come passeggiata:

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

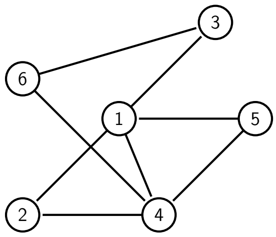
Definition 9. Traccia e Cammino

Sia G 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 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 si ha che:

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

Definition 11. Grafo connesso e fortemente connesso

Sia G un grafo. Definiamo G come **connesso** se

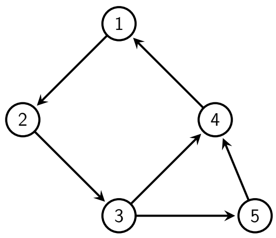
$$\forall v_1, v_2 \in V(G), \exists \text{ un cammino } \mid v_1 \to v_2 \lor v_2 \to v_1$$

Definiamo invece G come **fortemente connesso** se

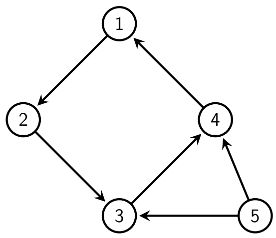
$$\forall v_1, v_2 \in V(G), \exists \text{ due cammini } \mid v_1 \to v_2 \land v_2 \to v_1$$

Esempio:

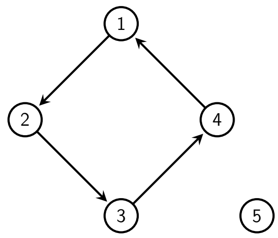
Fortemente connesso



Connesso



Non connesso



Observation 2

Un **grafo non diretto** è **connesso** se e solo se è **fortemente connesso**

Definition 12. Passeggiata chiusa ed aperta

Sia G un grafo. Definiamo una passeggiata $v_0e_1 \dots e_kv_k$ su G come **chiusa** se $v_0 = v_k$, altrimenti essa viene definita **aperta**

Definition 13. Passeggiata Hamiltoniana

Sia G 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 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, 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

$\square$

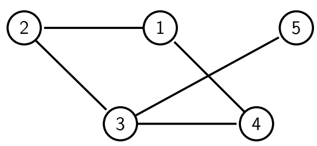
Definition 15. Ciclo

Sia G un grafo. Definiamo come **ciclo** una **passeggiata chiusa** dove solo il **primo** e l'**ultimo** vertice sono ripetuti.

Se G è un grafo diretto aciclico, definiamo G come **DAG (Directed Acyclic Graph)**

Esempio:

- Consideriamo il seguente grafo



- La seguente passeggiata è un ciclo in G

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

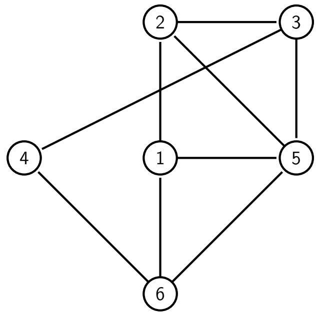
1.2 Depth-first Search (DFS)

Definition 16. Depth-first Search (DFS)

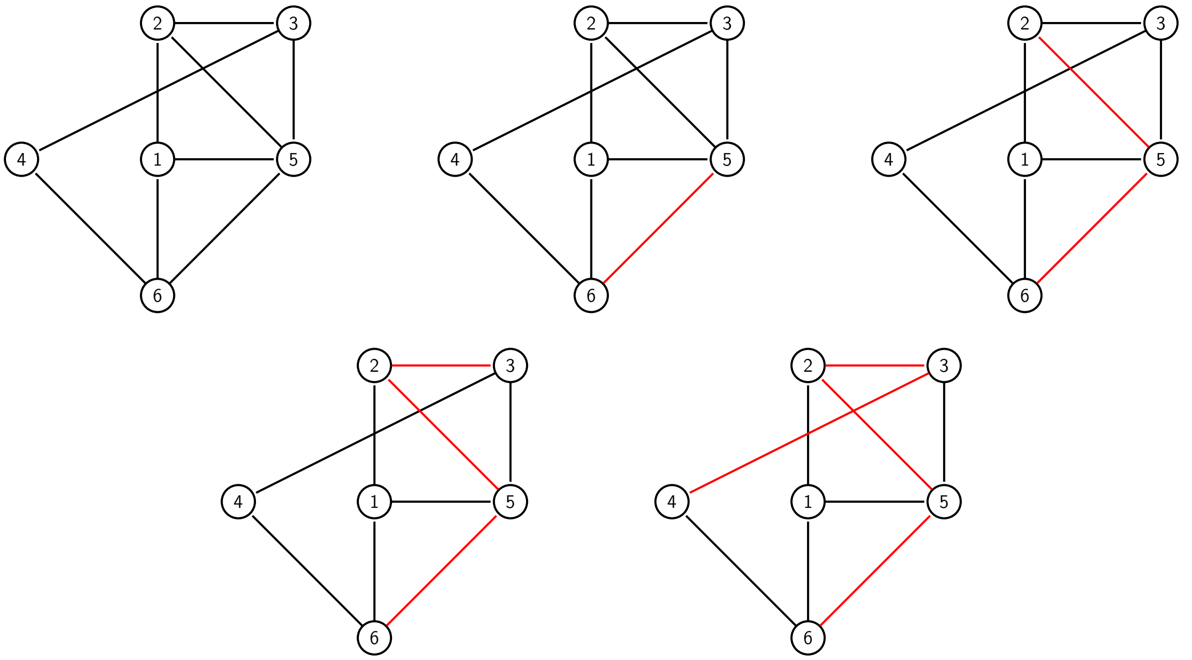
Sia G un grafo. Dato un vertice iniziale $x \in V(G)$ definiamo come **depth-first search (DFS)** un **criterio di visita** su G basato sul procedere **in profondità**, ossia dando precedenza ai vertici più lontani dal vertice iniziale, raggiungendo ogni vertice **una ed una sola volta**, tornando al 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:

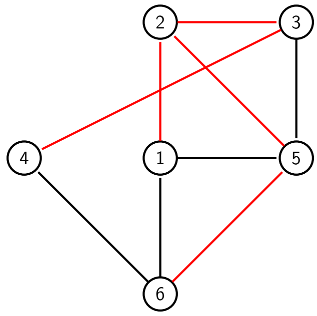
- Consideriamo il seguente grafo.



- Scelto 6 come vertice iniziale, 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 vertice iniziale stessa, per poi concludersi. L'ordine finale di visita, dunque, corrisponde a $6, 5, 2, 3, 4, 1$

Algorithm 1. Depth-first Search

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

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

Algorithm 1: Depth-first Search

Input:
G: grafo, $x \in V(G)$
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 $v_0 e_1 v_1 \dots v_{k-1} e_k v_k$ 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 $S = \varnothing$ e che $y \notin \texttt{Vis}$.
- Sia v_i il vertice di tale cammino avente indice maggiore dove $v_i \in \texttt{Vis}$, implicando che $v_{i+1} \notin \texttt{Vis}$. Se tale vertice esistesse, esso verrebbe tolto dallo stack prima che il vertice v_{i+1} sia visitato dall'algoritmo, poiché $v_{i+1} \notin \texttt{Vis}, \exists(v_i, v_{i+1}) \in E(G)$ 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 $S = \varnothing$ si abbia che $y \in \texttt{Vis}$

$\square$

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 vertice iniziale, 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 $O((2n - 1)n) = O(2n^2 - n) = O(n^2)$
  - 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 $O((2n - 1)n) = O(2n^2 - n) = O(n^2)$

1.2.1 Albero ed Arborescenza

Definition 17. Albero e Albero radicato

Sia T un grafo non diretto. Definiamo T come **albero** se $\forall x, y \in V(T)$ esiste un solo cammino tale che $x \to y$ e $y \to x$ passante per gli stessi vertici.

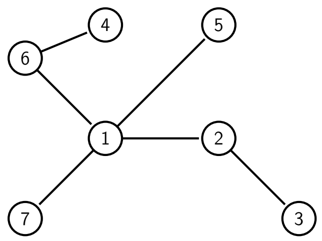
Se viene scelto un vertice privilegiato, detto **radice**, all'interno di T, definiamo T come **albero radicato**.

Definition 18. Arborescenza

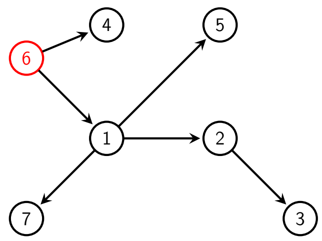
Sia A un grafo diretto. Dato un vertice $x \in V(A)$, detto **radice**, definiamo G come **arborescenza** se $\forall y \in V(A)$ esiste un solo cammino tale che $x \to y$

Esempio:

Albero



Arborescenza



Observation 3

Dato un grafo G, se G è un **albero** o un'**arborescenza**, allora

$$|E(G)| = |V(G)| - 1$$

(dimostrazione omessa)

Observation 4

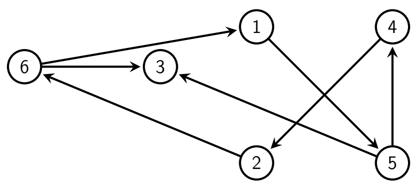
Sia G un grafo. Dato un vertice $x \in V(G)$, il **sottografo** $H \subseteq G$ generato dall'insieme di archi e vertici percorsi da una DFS avente x come vertice iniziale corrisponde a:

- Un albero radicato (se G non è diretto), detto **albero di visita**
- Un'arborescenza (se G è diretto), detta **arborescenza di visita**

Attenzione: eseguendo più di una volta una DFS su un vertice, non è detto che venga generato lo stesso albero/arborescenza di visita

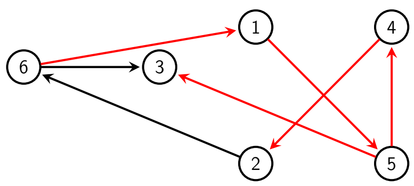
Esempio:

- Consideriamo il seguente grafo:

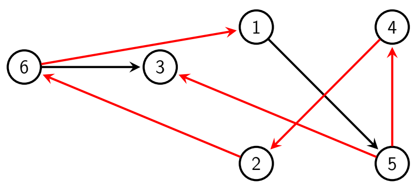


- Le arborescenze di visita ottenute effettuando una DFS avente come vertice iniziale il vertice 6 e il vertice 5 corrispondono a

Arborescenza di visita su 5



Arborescenza di visita su 6



1.2.2 DFS ottimizzata e ricorsiva

Algorithm 2. Depth-first Search (Ottimizzata)

Sia G 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 2: Depth-first Search (Ottimizzata)

Input:
G: grafo, $x \in V(G)$: vertice iniziale
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.uscenti ≠ ∅ do
            z = y.uscenti[0];
            y.uscenti.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.
- 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 $|E(G) = m|$
- Di conseguenza, il costo totale del ciclo while sarà $O(2n - 1 + m) = O(n + m)$. $\square$

Algorithm 3. Depth-first Search (Ottimizzata e Ricorsiva)

Sia G un grafo con liste di adiacenza e sia $x \in V(G)$ un vertice. Il seguente algoritmo effettua ricorsivamente 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 3: Depth-first Search (Ottimizzata e Ricorsiva)

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

```
Function DFS_recursive(G, x, Vis):
    for y ∈ x.uscenti do
        if y ∉ Vis then
            Vis.add(y);
            DFS_recursive(G, y, Vis);
        end
    end
end
Function DFS(G,x)
    Vis = {x};
    DFS_recursive(G, x, Vis);
    return Vis;
end
```

Dimostrazione correttezza e costo algoritmo:

- Analogamente alla DFS ottimizzata, tramite il ciclo for vengono analizzati solo una ed una volta tutti gli archi uscenti dell'attuale vertice x, applicando automaticamente le operazioni di rimozione degli archi, richiamando la risorsione solamente sui vertici mai visitati prima
- Inoltre, nonostante non sia presente una vera struttura dati, l' stack è stato "nascosto" sfruttando le chiamate ricorsive (in particolare, utilizzando lo stack della memoria di sistema), ottenendo lo stesso effetto della versione iterativa
- Per i motivi sopraelencati, il costo dell'algoritmo risulta essere $O(n+m)$

□

1.2.3 Tempi di visita, di chiusura e classificazione degli archi

Definition 19. Tempi di visita e Tempo di chisura

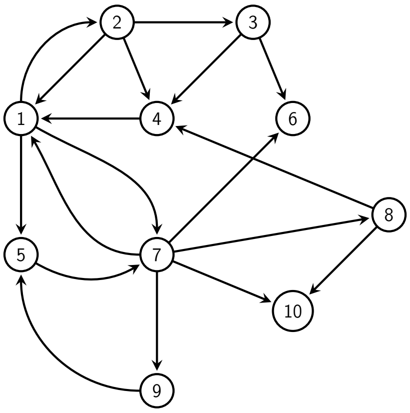
Sia G un grafo e sia C un **contatore** inizializzato a 0 inserito all'interno dell'algoritmo DFS, il quale viene **incrementato ogni volta che viene visitato un nuovo vertice**.

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.

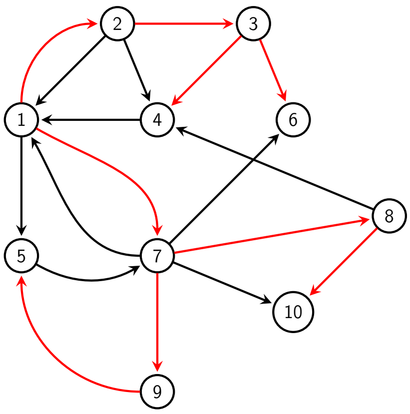
Definiamo inoltre come **intervallo di visita di** v l'intervallo $Int(v) := [t(v), T(v)]$

Esempio:

- Consideriamo il seguente multigrafo:



- Effettuando una DFS avente come vertice iniziale 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 5

Sia G 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:

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

Dimostrazione:

- Suppiamo per assurdo che $t(u) < t(v) \leq T(u) \leq T(v)$, 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 $T(u) \leq T(v)$.

- Analogamente, dimostriamo che $t(v) < t(u) \leq T(v) \leq T(u)$ è un caso impossibile
- Di conseguenza, le uniche possibilità sono:
  - $t(u) < t(v) \leq T(v) \leq T(u) \implies Int(u) \supseteq Int(v)$
  - $t(v) < t(u) \leq T(u) \leq T(v) \implies Int(u) \subseteq Int(v)$
  - $t(u) \leq T(u) < t(v) \leq T(v) \implies Int(u) \cap Int(v) = \varnothing$
  - $t(v) \leq T(v) < t(u) \leq T(u) \implies Int(u) \cap Int(v) = \varnothing$

$\square$

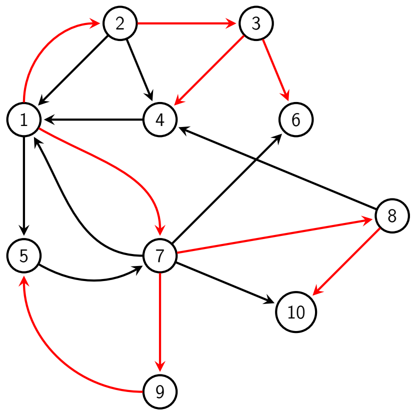
Definition 20. Archi all'indietro, in avanti e di attraversamento

Sia G un grafo e sia A 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 all'indietro (back edge)** se l'intervallo della coda è contenuto in quello della testa, ossia se $Int(u) \subseteq Int(v)$
- **Arco in avanti (forward 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 $Int(u) \cap Int(v) = \varnothing$

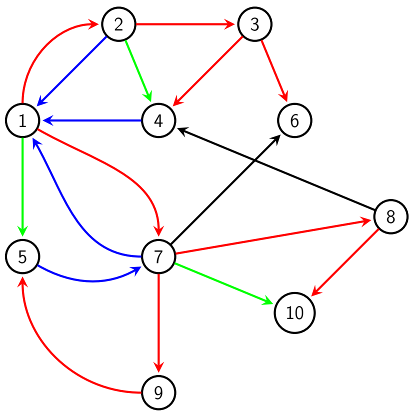
Esempio:

- 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 (blu), poiché $[2,5] \subseteq [1,10]$
  - L'arco $(2,4)$ è un forward edge (verde), poiché $[2,5] \supseteq [5,5]$
  - L'arco $(7,6)$ è un cross edge (nero), poiché $[6,10] \cap [4,4] = \varnothing$
  - ...



Algorithm 4. Trovare archi all'indietro, in avanti e di attraversamento

Sia G un grafo con liste di adiacenza e sia $x \in V(G)$ un vertice. Il seguente algoritmo effettua una DFS avente x come vertice iniziale, restituendo l'insieme degli archi all'indietro, in avanti e di attraversamento generati.

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

Algorithm 4: Trovare back edge, forward edge e cross edge generati da una DFS con vertice iniziale $x \in V(G)$ su un grafo diretto G

Input:
G: grafo diretto, x : $x \in V(G)$
Output:
Un insieme di back edge, un insieme di forward edge e un insieme di cross edge

```
Function classifyDirectEdges(G, x):
    Vis = [0, ..., 0]      //n elementi inizializzati a 0;
    t = [0, ..., 0];
    T = [0, ..., 0];
    Padri = [-1, ..., -1];
    Stack S := ∅;
    c = 1;
    Vis[x] = 1;
    S.push(x);
    t[x] = c;
    Padri[x] = x;
    while S ≠ ∅ do
        y = S.top();
        while y.uscenti ≠ ∅ do
            z = y.uscenti[0];
            y.uscenti.remove(0);
            if Vis[z] = 0 then
                Vis[z] = 1;
                S.push(z);
                c++;
                t[z] = c;
                Padri[z] = y;
        end
        if y == S.top() then
            S.pop();
            T[y] = c;
    end
    Back, Forward, Cross = ∅;
    for v ∈ V(G) do
        for u ∈ v.entranti do
            if Padri[v] ≠ u then
                if T[u] < t[v] ∨ T[v] < t[u] then
                    Cross.add((u, v));
                else if T[u] ≤ T[v] then
                    Back.add((u, v));
                else
                    Forward.add((u, v));
        end
    end
    return Back, Forward, Cross;
end
```

Dimostrazione correttezza algoritmo:

- In linea di massima, l'algoritmo risulta essere una versione modificata della versione ottimizzata della DFS (algoritmo 2). In particolare, sono stati aggiunti:
  - Un contatore c, il quale viene incrementato ogni volta che un vertice viene aggiunto allo stack (ossia quando viene visitato per la prima volta)
  - Un'array `Vis` di n elementi, dove se l'i-esimo elemento vale 1 allora il vertice $v_i \in V(G)$ è stato visitato (0 se non visitato)
  - Un'array `t`, dove l'i-esimo elemento dell'array corrisponde al tempo di visita del vertice $v_i \in V(G)$
  - Un'array `T`, dove l'i-esimo elemento dell'array corrisponde al tempo di chiusura del vertice $v_i \in V(G)$
  - Un'array `Padri`, dove l'i-esimo elemento dell'array corrisponde al padre del vertice $v_i \in V(G)$, ossia il vertice $u \in V(G)$ tramite cui è stato raggiunto il vertice v_i nella DFS
- Analizziamo quindi il comportamento degli ultimi due cicli for aggiunti:
  - Per ogni vertice $v \in V(G)$, vengono considerati tutti i suoi vertici entranti. Di conseguenza, stiamo considerando tutti gli archi $(u, v) \in E(G)$
  - Sia A l'arborescenza generata dalla DFS. Se `Padri[v]=u`, allora si ha che $(u, v) \in E(A)$, poiché v è stato visitato tramite u all'interno della DFS. Analogamente, se `Padri[v]=u`, allora si ha che $(u, v) \notin E(A)$, dunque (u, v) dovrà necessariamente essere un arco all'indietro, in avanti o di attraversamento
  - Poiché per definizione $t(u) \leq T(u)$ e $t(v) \leq T(v)$, all'interno dell'if si ha che:
    - $T(u) < t(v) \implies t(u) \leq T(u) < t(v) \leq T(v) \implies Int(u) \cap Int(v) = \varnothing$
    - $T(v) < t(u) \implies t(v) \leq T(v) < t(u) \leq T(u) \implies Int(u) \cap Int(v) = \varnothing$

    dunque (u, v) risulta essere un arco di attraversamento
  - All'interno dell'else if, dunque la condizione $T(u) \leq T(v)$, se si verificasse che $T(v) < t(v)$ si rientrerebbe nel caso precedente, dunque l'unica possibilità è che $t(v) < t(u) \leq T(u) \leq T(v) \implies Int(u) \subseteq Int(v)$, implicando che (u, v) sia un arco all'indietro
  - Infine, se (u, v) non è né un arco all'indietro e né di attraversamento, l'unica possibilità è che esso sia un arco in avanti

□

Dimostrazione costo computazionale:

- Poiché le uniche istruzioni aggiunte all'interno del ciclo while hanno un costo pari a $O(1)$, ne segue che il costo di tale while rimanga inalterato, ossia $O(n + m)$
- Per quanto riguarda i due cicli for annidati, il numero di iterazioni corrisponde a:

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

Di conseguenza, poiché le istruzioni al suo interno hanno tutte costo pari a $O(1)$, il costo finale dei due cicli for corrisponde a $O(m)$

- Infine, concludiamo che il costo dell'algoritmo sia $O(n + m) + O(m) = O(n + m)$

□

Algorithm 5. Trovare intervalli di visita (Ricorsivo)

Sia G un grafo con liste di adiacenza e sia $x \in V(G)$ un vertice. Il seguente algoritmo effettua ricorsivamente una DFS avente x come vertice iniziale, restituendo gli intervalli di visita di ogni vertice.

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

Algorithm 5: Trovare gli intervalli di visita generati da una DFS con vertice iniziale $x \in V(G)$ su un grafo diretto G

Input:
G: grafo diretto, x : $x \in V(G)$
Output:
Un insieme di back edge, un insieme di forward edge e un insieme di cross edge

```
Function DFS_recursive(G, x, Vis, t, T, c):
    for y ∈ x.adiacenti do
        if y ∉ Vis then
            Vis.add(y);
            t[u] = c;
            c.increment();
            DFS_recursive(G, y, Vis, t, T, c);
        end
    end
    T[x] = c;
end
Function getVisitTimes(G,x)
    Vis := {x};
    t := [0, ..., 0]          //n elementi inizializzati a 0;
    T := [0, ..., 0];
    Counter c := 1            //oggetto contatore;
    t[x] = c;
    DFS_recursive(G, x, Vis, t, T, c);
    return t, T;
end
```

Dimostrazione correttezza e costo algoritmo:

- Poiché sono stati aggiunti solo due array ed un oggetto contatore, il costo e la correttezza risultano essere analoghi alla normale DFS ricorsiva. Dunque, il costo è $O(n + m)$

□

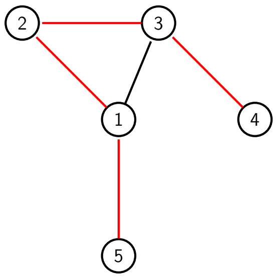
Observation 5

Se G è un grafo non diretto, **non vi è distinzione tra arco all'indietro ed arco in avanti**, poiché, non essendo gli archi orientati, non vi è distinzione tra testa e coda.

Di conseguenza, utilizziamo solo il termine arco all'indietro per indicare entrambe le situazioni

Esempio:

- Consideriamo il seguente grafo non diretto e l'albero di visita generato avente il vertice 3 come vertice iniziale.



v	t(v)	T(v)
1	3	4
2	2	4
3	1	5
4	5	5
5	4	4

- Notiamo che l'arco $(1, 3)$ non appartenente all'albero è sia un arco all'indietro sia un arco in avanti:
  - Se consideriamo 1 come coda ed 3 come testa, allora esso risulta essere un arco all'indietro, poiché $[3, 4] \subseteq [1, 5]$
  - Se consideriamo 3 come coda ed 1 come testa, allora esso risulta essere un arco in avanti, poiché $[1, 5] \supseteq [3, 4]$
- Dunque, per risolvere l'ambiguità, utilizziamo solo il termine arco all'indietro nel caso dei grafi non diretti

Proposition 6

Sia G un grafo non diretto. Dato l'albero di visita T generato da una DFS su G, per ogni arco di $(u, v) \in E(G) \mid (u, v) \notin E(T)$, si ha che $Int(u) \cap Int(v) \neq \varnothing$

Di conseguenza, in un **grafo non diretto non possono esistere cross-edge**

Dimostrazione:

- Supponiamo per assurdo che $Int(u) \cap Int(v) = \varnothing$, implicando che $t(u) \leq T(u) < t(v) \leq T(v)$ o che $t(v) \leq T(v) < t(u) \leq T(u)$.
- Eseendo G un grafo non diretto, si ha che $(u, v) \in E(G) \implies (v, u) \in E(G)$. Di conseguenza, ne segue che entrambi i casi siano impossibili:
  - Se viene vistato prima u, allora è impossibile che u venga tolto dallo stack prima di v, poiché l'arco $(u, v) \in E(G)$ verrebbe obbligatoriamente utilizzato dalla DFS, implicando che $Int(v) \subseteq Int(u)$

- Se viene vistato prima v, allora è impossibile che v venga tolto dallo stack prima di u, poiché l'arco $(v, u) \in E(G)$ verrebbe obbligatoriamente utilizzato dalla DFS, implicando che $Int(u) \subseteq Int(v)$

$\square$

Corollary 1

Sia G un **grafo non diretto connesso**. Dato l'albero di visita T generato da una DFS su G, ogni arco $(u, v) \in E(G) \mid (u, v) \notin E(T)$ è un **arco all'indietro**

1.3 Studio dei grafi ciclici e aciclici

1.3.1 Trovare cicli in un grafo

Algorithm 6. Trovare un ciclo in un grafo (con grado minimo 2)

Sia G 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 6: Trovare un ciclo in un grafo (con grado minimo 2)

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

```
Function findCycleMinDegree2(G):
    x = x ∈ V(G);
    Vis = [x];
    y = y ∈ V(G) | (x, y) ∈ E(G);
    while y ∉ Vis do
        Vis.add(y);
        y = z ∈ V(G) | (z, w) ∈ E(G), w ≠ Vis[Vis.length - 2];
    end
    return Vis[Vis.index(y) : Vis.length - 1];
end
```

Dimostrazione correttezza 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 $n \cdot O(1) = O(n)$

□

Theorem 7. Presenza di cicli in un grafo connesso non diretto

Dato un **grafo connesso non diretto** G, si ha che:

$$\forall \text{ DFS su } G, \exists \text{ arco all'indietro in } G \iff \exists \text{ ciclo in } G$$

Dimostrazione:

- Consieriamo un qualsiasi albero di visita T generato da una DFS su G. Poiché G è connesso, ogni vertice è raggiungibile dalla DFS, dunque si ha che $V(T) = V(G)$. Inoltre, poiché anche T è connesso, si ha che $\forall x, y \in V(T)$ esiste un cammino su T tale che $x \to y$
- Supponiamo quindi che esista un arco all'indietro $(u, v) \in E(G)$ generato da tale DFS, implicando che $(u, v) \notin E(T)$. Poiché $u, v \in V(T)$, ne segue che esisterà un cammino C su T tale che $u \to v$ e $(u, v) \notin C$. Infine, poiché in un grafo diretto si ha che $(u, v) \in E(G) \implies (v, u) \in E(G)$, la passeggiata $C \cup (v, u)$ risulta essere un ciclo.

- Viceversa, supponiamo che esista un ciclo in G composto dai vertici $c_0, \dots, c_k$. Poiché G è connesso, eseguendo una DFS su un qualsiasi vertice $x \in V(G)$, tale DFS dovrà necessariamente visitare almeno una volta ogni vertice $c_0, \dots, c_k$.
- Per comodità, assumiamo che c_0 sia il primo vertice appartenente al ciclo ad essere visitato. Una volta raggiunto il vertice c_k, l'arco (c_k, c_0) non potrà appartenere all'arborescenza generata, poiché c_0 risulta già essere stato visitato.
- Dunque, poiché in un grafo non diretto ogni arco ogni arco non appartenente ad un'arborescenza è un arco all'indietro, ne segue che (c_k, c_0) sia un arco all'indietro
- Inoltre, poiché $G = T$ e in un albero si ha che $\forall u, v \in V(G)$ esiste un solo cammino non diretto tale che $x \to y$ e $y \to x$, ogni DFS su G genererà l'albero di visita T

$\square$

Theorem 8

Un grafo G è **aciclico connesso e non diretto** se e solo se è un **albero**.

Dimostrazione:

- Se G è un grafo aciclico connesso e non diretto, per il teorema precedente si ha che

$$\nexists \text{ ciclo in } G \iff \exists \text{ DFS in } G \mid \nexists \text{ arco all'indietro in } G$$

- Sia quindi T l'albero di visita generato da tale DFS. Poiché G non è diretto e poiché non esistono archi all'indietro, ne segue che ogni arco appartenga necessariamente all'albero di visita, dunque $e \in E(G) \implies e \in E(T)$. Inoltre, poiché per definizione stessa si ha che $f \in E(T) \implies f \in E(G)$, concludiamo che $E(G) = E(T)$, implicando a sua volta che $G = T$
- Viceversa, se G è un albero allora, per definizione stessa, ne segue che $\forall x, y \in V(G)$ esista un solo cammino indiretto tale che $x \to y$ e $y \to x$, implicando quindi che G sia connesso ed aciclico

$\square$

Theorem 9. Presenza di cicli in un grafo connesso diretto

Dato un **grafo connesso diretto** G, si ha che:

$$\exists \text{ DFS su } G \mid \exists \text{ arco all'indietro in } G \iff \exists \text{ ciclo in } G$$

Dimostrazione:

- Consideriamo una DFS su G in cui viene generato un arco all'indietro $(u, v) \in E(G)$. Sia inoltre A l'arborescenza di visita generata da una DFS tale DFS.
- Poiché (u, v) è un arco all'indietro, ne segue che $t(v) < t(u) \leq T(u) \leq T(v)$, dunque u è stato aggiunto allo stack dopo v e prima che v venisse rimosso, implicando che esista un cammino C tale che $v \to u$.

  Di conseguenza, la passeggiata $C \cup (u, v)$ risulta essere un ciclo

- Viceversa, supponiamo che esista un ciclo $c_0, \dots, c_k$ in G e consideriamo una DFS avente radice $x \in V(G)$ in cui uno dei vertici del ciclo viene visitato (per comodità, supponiamo che venga visitato c_0), implicando che anche ogni vertice del ciclo debba essere visitato da tale DFS.
- Supponiamo per assurdo che esista un vertice del ciclo c_i con indice minimo $i \in [1, k]$ tale che c_i che non sia stato visitato prima della chiusura di c_0.
- Poiché c_i è stato scelto con indice minimo, ne segue che c_{i-1} sia stato visitato prima della chiusura di c_0, implicando che $t(c_0) < t(c_{i-1}) \leq T(c_0)$.
- Poiché $t(c_0) < t(c_{i-1}) \leq T(c_0) \leq T(c_{i-1}) \implies Int(c_0) \cap Int(c_{i-1}) \neq \varnothing$ è un caso impossibile, ne segue necessariamente che

  $$t(c_0) < t(c_{i-1}) \leq T(c_{i-1}) \leq T(c_0) \implies Int(c_{i-1}) \subseteq Int(c_0)$$

  dunque c_i viene chiuso prima della chiusura di c_0, implicando che la DFS abbia sbagliato a non visitare c_i, poiché $c_{i-1} \to c_i$
- Dunque, l'unica possibilità è che ogni vertice del ciclo venga visitato prima della chiusura di c_0, implicando che $Int(c_j) \subseteq Int(c_0), \forall j \in [1, k]$
- In particolare, quindi, ne segue che l'arco $(c_k, c_0) \in E(G)$ risulti essere un arco all'indietro poiché $Int(c_k) \subseteq Int(c_0)$

$\square$

Corollary 2

Dato un **grafo fortemente connesso diretto** G, si ha che:

$$\forall \text{ DFS su } G, \exists \text{ arco all'indietro in } G \iff \exists \text{ ciclo in } G$$

Dimostrazione:

- Se G è fortemente connesso, ne segue che i vertici $c_0, \dots, c_k$ componenti il ciclo vengano raggiunti da ogni DFS, dunque (per dimostrazione analoga alla precedente) l'arco $(c_k, c_0) \in E(G)$ sarà sempre un arco all'indietro

$\square$

Algorithm 7. Trovare un ciclo in un grafo

Sia G un grafo rappresentato tramite liste di adiacenza. Il seguente algoritmo restituisce, se esistente, un ciclo in G.

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

- $O(n)$ se G è non diretto
- $O(n + m)$ se G è diretto

dove $|V(G)| = n$ e $|E(G)| = m$

Algorithm 7: Trovare un ciclo in un grafo

Input:
G: grafo
Output:
Ciclo in G

```
Function DFS_Cycle(G, x, c, t, T, Padri, Cycle):
    c.increment();
    t[x] = c;
    for y ∈ x.uscenti do
        if Cycle == ∅ then
            if t[y] == 0 then
                Padri[y] = x;
                findCycle(G, y, c, t, T, Padri, Cycle);
            else if y ≠ Padri[x] and T[y] == 0 then
                z = x;
                while z ≠ y do
                    Cycle.head_insert(z);
                    z = Padri[z];
                end
                Cycle.head_insert(y);
            end
        end
    end
    T[x] = c;
end
Function findCycle(G):
    t = [0, ..., 0];
    T = [0, ..., 0];
    Padri = [-1, ..., -1];
    Counter c = 0;
    List Cycle = ∅;
    for v ∈ V(G) do
        if Cycle == ∅ and t[v] == 0 then
            Padri[v] = v;
            findCycle(G, y, c, t, T, Padri, Cycle);
        end
    end
    return Cycle;
end
```

Dimostrazione correttezza algoritmo:

- Sia $x \in V(G)$ il vertice attualmente analizzato dalla DFS e siano $y_0, \ldots, y_k \in V(G)$ i vertici tali che $(x, y_i) \in E(G)$,`t[`y_i`]`$= 0, \forall i \in [0, k]$, implicando dunque che $y_0, \ldots, y_k$ siano discendenti di x
- Sia $y' \in V(G)$ il vertice tale che $(x, y) \in E(G)$, `t[`y'`]` $\neq 0$ e `t[`T`]` $= 0$. Poiché x è il vertice attualmente visitato dalla DFS, ne segue che $t(y) < t(x) \leq T(x)$.

- Tuttavia, poiché `T[`y'`]` $= 0$, la visita del vertice y' non è stata ancora conclusa, implicando necessariamente che

$$t(y) < t(x) \leq T(x) \leq T(y) \implies (x, y') \text{ è un arco all'indietro}$$

- Di conseguenza, ne segue che y' sia un antenato di x e che tutti i nodi $z_0, \ldots, z_h \in V(G)$ aventi come antenato y e come discendente x siano appartenenti al ciclo

$\square$

Dimostrazione costo algoritmo:

- Trattandosi di una DFS modificata, ne segue automaticamente che il costo dell'algoritmo sia $O(n + m)$
- Supponiamo quindi che G sia non diretto. Poiché il caso peggiore dell'algoritmo viene raggiunto nel caso in cui G sia aciclico, per dimostrazione precedente ne segue necessariamente che G sia un'unione disgiunta di alberi $T_1, \ldots, T_k$ (o un singolo albero T se G è anche connesso)
- Di conseguenza, si ha che

$$m = |E(G)| = \sum_{i=0}^{k} |E(T_i)| = \sum_{i=0}^{k} |V(T_i)| - 1 \leq V(G) = n \implies m \leq n$$

  implicando dunque che il costo dell'algoritmo sia $O(n + m) = O(n)$

$\square$

Observation 6

Se G è un grafo non diretto, è possibile rimuovere dall'algoritmo precedente l'array `T`, i controlli e le operazioni inerenti ad esso, poiché in G possono esistere solo archi all'indietro.

Di conseguenza, è sufficiente verificare che $t(y) < t(x)$ affinché $(x, y) \in E(G)$ sia un arco all'indietro

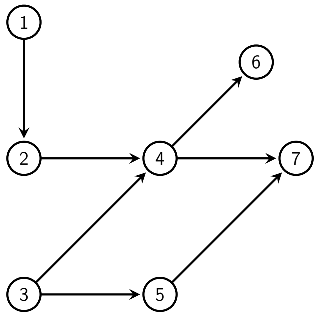
1.3.2 Ordinamenti topologici

Definition 21. Ordinamento topologico

Sia G 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 10

Dato un grafo diretto G, 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 possibilità 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.

$\square$

Observation 7

Dato un grafo diretto aciclico G, 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 $\nexists v \in V(G) \mid deg_{out}(v) = 0$. Poiché G è aciclico, esiste un ordinamento topologico $v_0, \ldots, v_n$ in G, dove $|V(G)| = n$.
- Poiché ogni vertice ha almeno un vertice uscente, per comodità supponiamo che $(v_i, v_{i+1}) \in E(G), \forall i \in [1, n-1]$, implicando quindi che $v_1 \to v_2 \to \ldots \to v_n$
- Poiché $deg_{out}(v_n) \neq 0$, ne segue che $\exists v_k \in V(G) \mid k \in [0, n-1]$ tale che $(v_n, v_k) \in E(G)$. Di conseguenza, esiste un ciclo in G tale che $v_k \to v_n \to v_k$, contraddicendo l'ipotesi per cui G sia aciclico.
- Seguendo un ragionamento analogo, dimostriamo che se si avesse $\nexists v \in V(G) \mid deg_{in}(v) = 0$ si otterrebbe una contraddizione
- Di conseguenza, l'unica possibilità è che $deg_{in}(v_0) = 0$ e $deg_{out}(v_n) = 0$

□

Algorithm 8. Trovare un ordinamento topologico

Sia G un DAG. Il seguente algoritmo restituisce un ordinamento topologico di G

Il **costo computazionale** di tale algoritmo è $O(n(n+m))$, dove $|V(G)| = n$ e $|E(G)| = m$, se G è rappresentato tramite liste di adiacenza

Algorithm 8: 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 $V(G) \neq \varnothing$. 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 dimostrato 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 vertice $v \in V(G) \mid \forall v' \in V(G), \exists (v, v') \in E(G)$, 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 $n \cdot O(n+m) = O(n(n+m))$

□

Problem 1. Programmazione di un processo di produzione

Una fabbrica ha diviso un processo di produzione in n fasi. Tra alcune coppie di fasi vi è una dipendenza, ossia una di esse 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 dipendenze tra ogni fase come degli archi diretti, dove $\exists (x_i, x_j) \in E(G) \iff x_j$ dipende da x_i

- 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 utilizzare l'algoritmo 7 `findCycle` avente costo $O(n + m)$. Nel caso in cui non venga restituito alcun ciclo, possiamo utilizzare l'algoritmo 8 `findTopologicalSorting` avente costo $O(n(n + m))$ per trovare una programmazione valida. In caso contrario, non sarebbe possibile trovare una programmazione valida.
- Il costo finale di tale algoritmo, dunque, sarebbe $O(n(n + m))$

Proposition 11

Sia G un DAG connesso. Dato $(u, v) \in E(G)$, si ha che

$$t(v) \leq T(v) \leq T(u)$$

Dimostrazione:

- Sia A l'arborescenza generata da una DFS su G. Se $(u, v) \in E(A)$, ne segue automaticamente che $t(u) < t(v) \leq T(v) \leq T(u)$
- Sia quindi $(u, v) \in E(G) \mid (u, v) \notin E(A)$. Supponiamo per assurdo che $t(v) > T(u)$, implicando che v venga aggiunto allo stack dopo la chisura di u. In tal caso, la DFS avrebbe sbagliato a non percorrere l'arco $(u, v) \in E(G)$, implicando che u non possa essere tolto dallo stack prima di v.
- Di conseguenza, l'unica possibilità è che $t(v) \leq T(u)$. Inoltre, poiché G è un DAG connesso, dunque non esistono archi all'indietro in G, ne segue necessariamente che:
  - $Int(u) \supseteq Int(v) \implies t(u) < t(v) \leq T(v) \leq T(u)$
  - $Int(u) \cap Int(v) = \varnothing \implies t(v) \leq T(v) < t(u) \leq T(u)$

$\square$

Algorithm 9. Trovare un ordinamento topologico (Ottimizzato)

Sia G un DAG rappresentato tramite liste di adiacenza. Il seguente algoritmo restituisce un possibile ordinamento topologico di G.

Il **costo computazionale** di tale algoritmo è $O(n + m)$

Algorithm 9: Trovare un ordinamento topologico in un DAG

```
Input:
G: grafo diretto aciclico connesso
Output:
Ordinamento topologico in G
Function DFS_Ord(G, u, Vis, L):
    Vis.add(u);
    for v ∈ u.uscenti do
        if v ∉ Vis then
            DFS_Ord(G, v, Vis, L);
    end
    L.head_insert(v);
end
Function findTopologicalSorting_2(G):
    List L = ∅;
    Vis = ∅;
    for u ∈ V do
        if u ∉ Vis then
            recursive_DFS_ord(G, u, Vis, L);
    end
    return L;
end
```

Dimostrazione correttezza e costo algoritmo:

- Consideriamo gli archi $(u, v) \in E(G)$ generati in `recursive_DFS_ord()`. Poiché G è un DAG, per dimostrazione precedente si ha che $t(v) \leq T(v) \leq T(u)$.

  Di conseguenza, ordinando i vertici in modo che il loro tempo di chiusura sia decrescente, svolto implicitamente dalla ricorsione appendendo il vertice attualmente analizzato all'inizio della lista, otteniamo un ordine topologico, poiché ogni vertice uscente v verrà inserito in testa prima del vertice attuale u

- Consideriamo quindi gli elementi $L_i := u_i, \ldots, v_k$ aggiunti dal vertice u_1 all'iterazione i-esima del ciclo for di `findTopologicalSorting_2()`. Nel caso in cui esista un arco $(v_i, v_{i+1}) \in E(G) \mid v_i \in L_i, v_{i+1} \in L_{i+1}$, si ha che

  $$(v_i, v_{i+1}) \implies t(v_{i+1}) \leq T(v_{i+1}) \leq T(v_i) \implies v_{i+1} \in L_i \implies L_{i+1} \subseteq L_i$$

  Di conseguenza, le varie sottoliste $L_1, \ldots, L_j$ sono disgiunte tra loro, implicando che esse possano essere inserite nell'ordinamento in un ordine qualsiasi

- Inoltre, essendo l'algoritmo una semplice DFS ricorsiva modificata, il suo costo risulta automaticamente essere $O(n + m)$

□

1.3.3 Ponti di un grafo

Definition 22. Ponte

Sia G un grafo **non diretto**. Dato un arco $f \in E(G)$, definiamo f come **ponte** se esso non appartiene a nessun ciclo in G:

$$f \text{ ponte} \iff \nexists \text{ ciclo in } G \mid f \text{ è nel ciclo}$$

Algorithm 10. Stabilire se un arco è un ponte

Sia G un grafo non diretto rappresentato tramite liste di adiacenza. Dato un arco $f \in E(G)$, il seguente algoritmo stabilisce se f è un ponte.

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

Algorithm 10: Stabilire se $f \in E(G)$ è un ponte

Input:
G: grafo non diretto a liste di adiacenza,
$f : f \in E(G)$
Output:
True se f è un ponte, False altrimenti

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

Dimostrazione correttezza algoritmo:

- Sia G' il grafo in cui è stato rimosso $f := (x, y)$, dunque dove $E(G') := E(G) - f$.
- Supponiamo che $x \in$ `Vis`. Poiché $x \in$ `Vis` $\iff y \to x$, ne segue che esista un cammino $ye_1 \dots e_k x$. In particolare, poiché $e_1, \dots, e_k \in E(G') \subset E(G)$, tale cammino esiste anche in G, implicando che $ye_1 \dots e_k xfy$ sia un ciclo e dunque che f non sia un ponte.
- Viceversa, supponiamo per assurdo che f non sia un ponte e che $x \notin$ `Vis`, implicando che $y \not\to x$ e dunque che non esista una passeggiata $yh_1 \dots h_k x$, contraddicendo l'ipotesi per cui f non sia un ponte, poiché il ciclo $yh_1 \dots h_k xfy$ non potrebbe esistere. Di conseguenza, l'unica possibilità è che $x \in$ `Vis`

- Dunque, concludiamo che f non è un ponte se e solo se $x \in \texttt{Vis}$

$\square$

Dimostrazione costo algoritmo:

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

$$O(deg_{in}(x) + deg_{out}(y)) + O(n+m) = O(deg_{in}(x) + deg_{out}(y) + n + m) = O(n+m)$$

$\square$

Algorithm 11. Trovare i ponti di un grafo (Soluzione naïve)

Sia G un grafo non diretto rappresentato tramite liste di adiacenza. Il seguente algoritmo trova i ponti presenti in G.

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

Algorithm 11: Trovare i ponti in un grafo

Input:
G: grafo a liste di adiacenza
Output:
Insieme dei ponti presenti in G

```
Function findBridges_1(G: grafo):
    Bridges = {};
    for f ∈ E(G) do
        if isBridge(f) then
            Bridges.add(f);
        end
    end
    return Bridges;
end
```

Dimostrazione correttezza e costo algoritmo:

- Iterando su ogni arco in $E(G)$, stabiliamo se $f \in E(G)$ sia un ponte utilizzando l'algoritmo 10 `isBridge()`, il cui costo è $O(n+m)$. Di conseguenza, il costo finale sarà $O(m(n+m))$

$\square$

Observation 8

Sia G un grafo non diretto connesso. Se $f \in E(G)$ è un arco all'indietro generato da una DFS su G, allora f non è un ponte.

Dimostrazione:

- Poiché $f := (u, v)$ è un arco all'indietro, ne segue che

$$[t(u), T(u)] \subseteq [t(v), T(v)] \implies t(v) < t(u) \leq T(u) \leq T(v)$$

  dunque u è stato aggiunto allo stack dopo v e prima che v venisse rimosso, implicando che esista un cammino C tale che $v \to u$.

  Di conseguenza, la passeggiata $C \cup (u, v)$ risulta essere un ciclo, implicando che (u, v) non sia un ponte

$\square$

Algorithm 12. Trovare i ponti di un grafo non diretto connesso

Sia G un grafo non diretto connesso rappresentato tramite liste di adiacenza. Il seguente algoritmo trova i ponti presenti in G.

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

Algorithm 12: Trovare i ponti in un grafo non diretto connesso

Input:
G: grafo non diretto connesso a liste di adiacenza
Output:
Insieme dei ponti presenti in G

```
Function findBridges_2(G: grafo):
    Bridges = {};
    Backedges = classifyDirectEdges(G, x ∈ V(G));
    for f ∈ E(G) do
        if f ∉ Backedges then
            if isBridge(G, f) then
                Bridges.add(f);
            end
        end
    end
    return Bridges;
end
```

Dimostrazione correttezza e costo algoritmo:

- Sia T l'albero di visita generato da una DFS su $x \in V(G)$. Poiché G è un grafo non diretto connesso, ne segue che tutti gli archi $e \in E(G) \mid e \notin E(T)$ siano degli archi all'indietro. Di conseguenza, tali archi non possono essere dei ponti, rendendo sufficiente esaminare solo gli archi in $E(T)$.

- Poiché T è un albero, dunque $|E(T)| = |V(T)| - 1 = n - 1$, e poiché il costo dell'algoritmo 10 `isBridge()` è $O(n + m)$, il costo del ciclo for risulta essere pari a $O((n - 1)(n + m)) = O(n(n + m))$

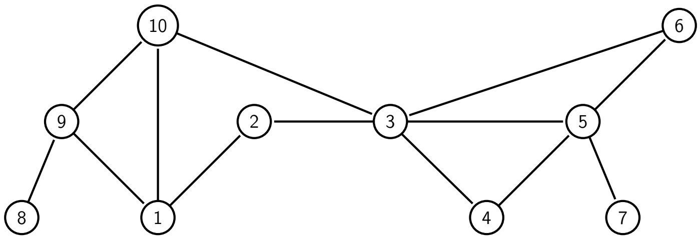
□

Definition 23. Sottoalbero dei discendenti

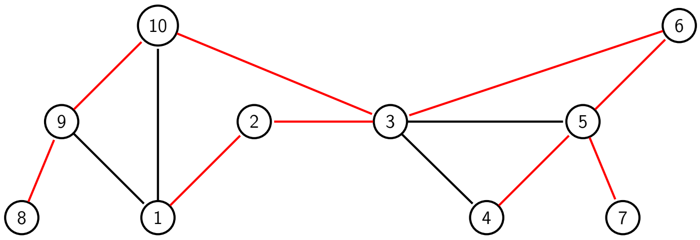
Sia G un grafo non diretto e sia T un albero di visita generato da una DFS su G. Dato un arco $(x, y) \in E(T)$, definiamo $T_y \subseteq T$ il **sottoalbero dei discendenti di** y **nell'albero** T costituito dai vertici e gli archi raggiunti tramite y nella DFS.

Esempio:

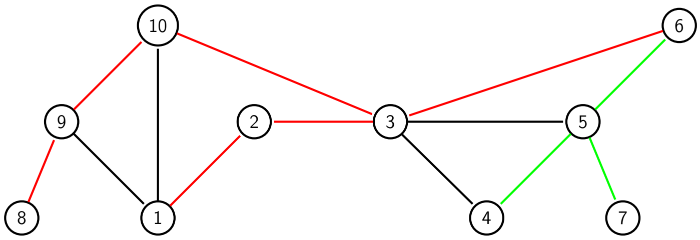
- Consideriamo il seguente grafo.



- Eseguendo una DFS sul vertice 1, otteniamo il seguente albero di visita T:



- Dato l'arco $(3, 6) \in E(T)$, il sottoalbero dei discendenti T_6 generato corrisponde a



Theorem 12. Esistenza di un ciclo contenente un arco

Siano G un grafo non diretto connesso, T un albero di visita generato da una DFS su G e T_y l'albero dei discendenti di y in T generato da un arco $(x, y) \in E(T)$.

Dati un vertice $u \in V(T_y)$ ed un vertice $v \in V(T - T_y)$, si ha che:

$$\exists (u, v) \neq (x, y) \in E(G) \iff \exists \text{ ciclo in } G \text{ contenente } (x, y)$$

Dimostrazione:

- Poiché G è un grafo non diretto connesso, ogni vertice in $V(G)$ verrà visitato dalla DFS, implicando che $V(T) = V(G)$. Di conseguenza, si ha che $\forall z \notin V(T_y) \implies z \in V(T - T_y)$
- Sia quindi $g := (x, y) \in E(A)$. Supponiamo che esista un arco $f := (u, v) \in E(G)$ tale che $u \in V(T_y)$ e $v \in V(T - T_y)$. Poiché $x \notin V(T_y) \implies x \in V(T - T_y)$ e poiché T è connesso, esiste un cammino $ve_1 \dots e_k x$ non contenente (u, v) tale che $v \to x$. Inoltre, poiché $u \in V(T_y)$, ne segue automaticamente che esiste un cammino $yh_1 \dots h_j u$ tale che $y \to u$.

  Dunque, la passeggiata $ve_1 \dots e_k xgyh_1 \dots h_j ufv$ risulta essere un ciclo contenente g
- Viceversa, supponiamo per assurdo che non esista tale arco e che esista un ciclo contenente g, implicando che esista un cammino $yd_1 \dots d_p x$ non passante per g tale che $y \to x$. Poiché $y \in V(T_y)$ e $x \in V(T - T_y)$, esisterà necessariamente un arco $d_i : (a, b) \neq (x, y) \in E(G)$ all'interno del ciclo tale che $a \in V(T_y)$ e $b \in V(T - T_y)$, contraddicendo l'ipotesi iniziale.

  Di conseguenza, l'unica possibilità è

  $$\nexists (u, v) \neq (x, y) \in E(G) \implies \nexists \text{ ciclo in } G \text{ contenente } (x, y)$$

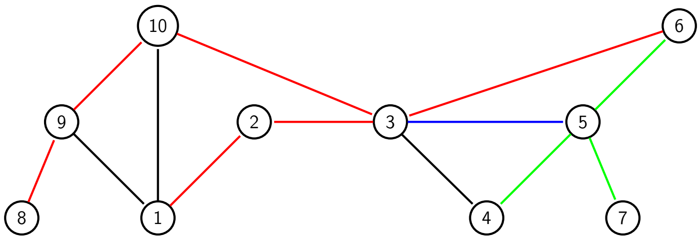
  da cui per contro-nominale otteniamo che

  $$\exists \text{ ciclo in } G \text{ contenente } (x, y) \implies \exists (u, v) \neq (x, y) \in E(G)$$

$\square$

Esempio:

- Riprendendo l'esempio precedente, l'arco $(5, 3) \in E(G)$ dove $5 \in V(T_6)$ e $3 \in V(T) - V(T_6)$ crea un ciclo in G



Observation 9

Sia G un grafo non diretto connesso. Dato un arco $(u, v) \in E(T)$, se (u, v) è un ponte, eseguendo una DFS su G radicata in u, si ha che $(u, v) \in E(T)$, dove T è l'albero generato dalla DFS

Dimostrazione:

- Poiché (u, v) è un ponte, ne segue che non esista un ciclo contenente (u, v), implicando a sua volta che non esista un cammino non contenente (u, v) tale che $u \to v$.
- Di conseguenza, poiché G è connesso non diretto, dunque $V(T) = V(G)$, l'unica possibilità affinché $u, v \in V(T)$ e se $(u, v) \in E(T)$

□

Observation 10

Sia G un grafo non diretto e sia T un albero di visita generato da una DFS su G.

Se esiste un arco $(x, y) \in E(G - T)$ tale che $deg^T(x) = deg^T(y) = 1$ in T, allora x o y devono essere la radice di T

Dimostrazione:

- Supponiamo per assurdo che né x né y siano la radice di T, dunque che $\exists (u, x), (v, y) \in E(T)$ tramite cui vengono visitati x e y nella DFS.
- Poiché $deg^T(x) = 1$, ne segue che al momento della visita di x il vertice y fosse stato già visitato, poiché altrimenti si avrebbe che $(x, y) \in E(T)$. Analogamente, poiché $deg^T(y) = 1$, ne segue che al momento della visita di y il vertice x fosse stato già visitato, poiché altrimenti si avrebbe che $(y, x) \in E(T) \implies (x, y) \in E(T)$.
- Di conseguenza, si ha che $Int(x) \cap Int(y) = \varnothing$, implicando che l'arco $(x, y) \in E(G - T)$ sia un arco di attraversamento, contraddicendo la proposizione per cui in G, essendo un grafo non diretto, non possano esistere archi di attraversamento. Dunque, l'unica possibilità è che x o y sia necessariamente la radice del DFS

□

Algorithm 13. Trovare i ponti di un grafo non diretto connesso (Ottim.)

Sia G un grafo non diretto connesso rappresentato tramite liste di adiacenza. Il seguente algoritmo trova i ponti presenti in G.

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

Algorithm 13: Trovare i ponti in un grafo non diretto connesso (Ottimizzato)

Input:
G: grafo non diretto connesso a liste di adiacenza,
c: contatore,
t: array tempi di visita,
Back: array dei vertici esterni ai sottoalberi più lontani e adiacenti ad un vertice interno ai sottoalberi
Padri: array dei padri
Output:
Insieme dei ponti presenti in G

```
Function DFS_Bridges(G, x, c, t, Back, Padri):
    c.increment();
    t[x] = c;
    Back[x] = t[x];
    for y ∈ V(G) | (x, y) ∈ E(G) do
        if t[y] = 0 then
            Padri[y] = x;
            DFS_Bridges(G, y, c, t, Back, Padri);
            if Back[y] < Back[x] then
                Back[x] = Back[y];
        else if y ≠ Padri[x] then
            if t[y] < Back[x] then
                Back[x] = t[y];
    end
end
Function findBridges_3(G):
    v = v ∈ V(G);
    Counter c := 0;
    t = [0, . . . , 0];
    Padri = [-1, . . . , -1];
    Back = [0, . . . , 0];
    Padri[v] = v            //v è la radice;
    DFS_Bridges(G, v, c, t, Back, Padri);
    Bridges = ∅;
    for u ∈ V(G) do
        if Back[u] = t[u] and u ≠ Padri[u] then
            Bridges.add((Padri[u], u));
        end
    end
    return Bridges;
end
```

Dimostrazione correttezza e costo algoritmo:

- Sia x il vertice attualmente esplorato durante la ricorsione della funzione `DFS_Bridges`.
- Tramite il ciclo for, proseguiamo con la ricorsione sui vertici $y \in V(G) \mid t[x] = 0, (x, y) \in E(G)$, implicando che y non sia già stato visitato. L'effetto ottenuto, dunque, è quello di una DFS.

- Dato l'albero T generato dalla DFS, siano $y_0, \dots, y_k$ i vertici adiacenti ad x esplorati nella DFS per la prima volta tramite x stesso, implicando che essi siano discendenti di x, dunque che $y_0, \dots, y_k \in V(T_x)$.
- Siano invece $z_0, \dots z_h$ i vertici adiacenti ad x già visitati dalla DFS, implicando che $z_0, \dots, z_j \in V(T - T_x)$, dove in particolare, per via dell'else-if, si ha che $z_i \neq p_x, \forall i \in [0, h]$, dove $p_x :=$ `Padri[`x`]`
- Sia quindi `Back[`x`] =` $t(b_x)$, dove b_x è il vertice in $V(T - T_x)$ con tempo di visita minore possibile tale che $\exists (d_x, b_x) \in E(G)$ dove $d_x \in V(T_x)$, implicando che:

$$\texttt{Back}[x] = \min(\texttt{Back}[y_0],\ \dots,\ \texttt{Back}[y_k],\ \texttt{t}[y],\ \texttt{t}[z_0],\ \dots,\ \texttt{t}[z_h])$$

- Nel caso in cui `Back[`x`]` $\neq$ `t[`x`]`, dunque $b_x \neq x$, esisterebbe un arco $(b_x, d_x) \neq (p_x, x) \in E(G)$ tale che $b_x \in V(T - T_x)$ e $d_x \in V(T_x)$. Per il teorema precedente, tale arco può esistere se e solo se esiste un ciclo in G contenente l'arco $(p_x, x) \in E(G)$. Di conseguenza, si ha che (p_x, x) è un ponte se e solo se `Back[`x`]` $=$ `t[`x`]`.
- Poiché l'algoritmo effettua una DFS ricorsiva modificata e il costo di tutte le operazioni della ricorsione è $O(1)$, il costo computazionale totale risulta essere $O(n + m)$

$\square$

1.4 Componenti di un grafo

Proposition 13

Sia G un grafo diretto. Dato un vertice $x \in V(G)$, si ha che:

$$G \text{ fortemente connesso} \iff \forall y \in V(G), \exists \text{ due cammini } \mid x \to y, y \to x$$

- Se G è fortemente connesso, per definizione stessa ne segue automaticamente che $\forall y \in V(G), \exists$ due cammini $\mid x \to y, y \to x$
- Viceversa, se $\forall y \in V(G), \exists$ due cammini $\mid x \to y, y \to x$, per ogni coppia di vertici $u, v \in V(G)$ si ha che $u \to x \to v$ e $v \to x \to u$, dunque G è fortemente connesso

$\square$

Observation 11

Sia G un grafo. Se $|V(G)| = 1$ e $|E(G)| = 0$, allora G è fortemente connesso

Dimostrazione:

- Essendo $v \in V(G)$ l'unico vertice in $V(G)$, esso è raggiungibile da se stesso e può raggiungere se stesso tramite un cammino nullo

$\square$

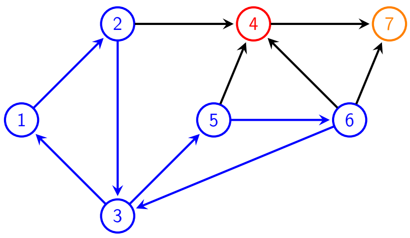
Definition 24. Componenti di un grafo

Sia G un grafo. Definiamo come **componente** di G un sottografo $H \subseteq G$ **fortemente connesso e massimale**, ossia $\nexists H' \subset H$ fortemente connesso.

Dato un vertice $v \in V(G)$, indichiamo come $comp(v)$ il componente $comp(v) \subseteq G$ tale che $v \in comp(v)$

Esempio:

- I componenti del seguente grafo corrispondono a $H_1 := \{1, 2, 3, 5, 6\}$, $H_2 := \{4\}$, $H_3 := \{7\}$



Observation 12

Sia G un grafo. Date le sue componenti $H_1, \ldots, H_k \subseteq G$, si ha che

$$H_i \cap H_j = \varnothing, \forall i \neq j$$

Dimostrazione:

- Date le componenti $H_1, \ldots, H_k$ diverse tra loro, supponiamo per assurdo che $\exists i, j \in [1, k] \mid H_i \cap H_j \neq \varnothing$, implicando che $\exists v \in V(H_i) \cap V(H_j) \iff v \in V(H_i), v \in v(H_j)$.
- Poiché H_i è fortemente connesso, si ha che $\forall x \in H_i$ esistono due cammini tali che $v \to x, x \to v$. Analogamente, $\forall y \in H_j$ esistono due cammini tali che $v \to y, y \to v$. Di conseguenza, si avrebbe che $\forall x \in H_i, \forall y \in H_j$ esistono due cammini tali che $x \to v \to y$ e $y \to v \to x$, implicando che $H_i = H_j$ e contraddicendo l'ipotesi

□

Definition 25. Contrazione in un vertice

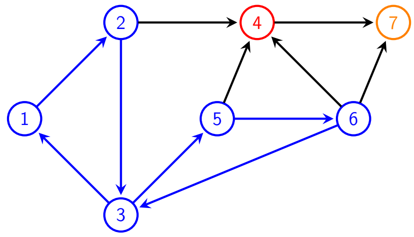
Sia G un grafo. Dato un sottografo fortemente connesso $H \subseteq G$, definiamo come **contrazione di H in un vertice v_H** l'operazione tramite cui:

- Vengono rimossi da $V(G)$ i vertici in $V(H)$, sostituendoli con un vertice v_H
- Vengono rimossi tutti gli archi $(u, v), (v', u') \in E(G)$ tali che $u, u' \in V(H)$ e $v, v' \in V(G - H)$, sostituendoli con un arco (v_H, v) e un arco (v', v_H)

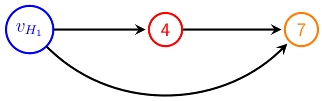
Il grafo ottenuto viene indicato come $G/V(H)$, letto "G **contratto** $V(H)$"

Esempio:

- Consideriamo ancora il grafo precedente.



- Poiché il componente $H_1 := \{1, 2, 3, 5, 6\}$ è un grafo fortemente connesso, possiamo contrarre H_1 nel vertice v_{H_1}. Il grafo $G/V(H_1)$ risultante corrisponde a:



Theorem 14. Contrazione di un sottografo fortemente connesso

Sia G un grafo **fortemente connesso**. Dato un **sottografo fortemente connesso** $H \subseteq G$, allora $G/V(H)$ è **fortemente connesso**

Dimostrazione:

- Dato $u \neq v_H \in V(G/V(H))$, si ha che $u \in V(G)$. Poiché G è fortemente connesso, esistono due cammini tali che $u \to h$ e $h \to u$ in G, dove $h \in V(H)$.
- Sia quindi $v_H \in G/V(H)$ il vertice in cui è stato contratto H. Poiché $u \to h$ in G, ne segue che esista un cammino $u \to v_H$ in $G/V(H)$. Analogamente, poiché $h \to u$ in G, ne segue che esista un cammino $v_H \to u$ in $G/V(H)$. Dunque, concludiamo che $G/V(H)$ sia fortmente connesso.

□

Lemma 15

Un **grafo fortemente connesso diretto** G dove $|V(G)| > 1$ è sempre **ciclico**

Dimostrazione:

- Dati $u, v \in V(G)$, poiché G è fortemente connesso si ha che esiste un cammino diretto $ue_1 \dots e_k v$ tale che $u \to v$ ed un cammino diretto $vh_1 \dots h_j u$ tale che $v \to u$.
- Di conseguenza, esiste sempre un ciclo $ue_1 \dots e_k vh_1 \dots h_j u$

□

Lemma 16

Sia G un grafo. Dato un ciclo $v_0 e_1 v_1 \dots v_{k-1} e_{k-1} v_k e_k v_0$ in G, il sottografo $C \subseteq G$ tale che $v_0, v_1, \dots, v_{k-1}, v_k \in V(C)$ e $e_1, \dots, e_k \in E(C)$ è un grafo **fortemente connesso**

Dimostrazione:

- Sia $C \subseteq G$ tale che $v_0, v_1, \dots, v_{k-1}, v_k \in V(C)$ e $e_1, \dots, e_k \in E(C)$
- Essendo $v_0 e_1 v_1 \dots v_{k-1} e_{k-1} v_k e_k v_0$ un ciclo in G, tale ciclo risulta esistere anche in C, dunque si ha che $\forall v_i, v_j \in V(C) \mid i \neq j \implies v_i \to v_j, v_j \to v_i$ in C

□

Algorithm 14. Trovare i componenti di un grafo diretto

Sia G un grafo diretto rappresentato tramite liste di adiacenza. Il seguente algoritmo trova i componenti di G.

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

Algorithm 14: Trovare i componenti di un grafo diretto

Input:
G: grafo a liste di adiacenza
Output:
Insieme di componenti in G

```
Function getComponents(G):
    C = findCycle(G);
    if C == ∅ then
        return {{v} | ∀v ∈ V(G)}        //è un insieme di insiemi;
    else
        G/V(C), v_C = contractGraph(G, C);
        {H_1, ..., H_k} = getComponents(G/V(C));
        uncontrComponents = ∅;
        for i ∈ [1, k] do
            if v_C ∉ H_i then
                uncontrComponents.add(H_i);
            else
                H'_i = (H_i − {v_C}) ∪ V(C);
                uncontrComponents.add(H'_i);
            end
        end
    end
    return uncontrComponents;
end
```

Dimostrazione correttezza algoritmo:

- Sia H un componente di G. Se $|V(H)| > 1$, per dimostrazione precedente esiste un ciclo in H poiché H è un sottografo diretto fortemente connesso.
- Sia quindi C il sottografo composto dagli archi e i vertici di tale ciclo, implicando che, per il lemma precedente, C sia fortemente connesso. Per il teorema precedente, dunque, anche $H' := H/V(C)$ risulta essere fortemente connesso, implicando che esso sia un componente di $G/V(C)$.
- Applicando tale procedimento ricorsivamente, l'intero componente H arriverà ad essere contratto in un singolo vertice v_H, il quale risulterà essere un componente connesso della versione finale del grafo G_f.
- Una volta raggiunto il caso base, ossia una volta che $C == \varnothing$, ogni punto del grafo G_f sarà un componente di quest'ultimo, implicando che i vertici $v_{H_1}, \dots, v_{H_k} \in V(G_f)$ siano le contrazioni massime dei componenti $comp(v_{H_1}), \dots, comp(v_{H_k})$ di G.
- Sia quindi $H_i = comp(v_{H_i})$ e siano $H_i/V(C_0), \dots, H_i/V(C_q)$ le contrazioni interne ad H_i tramite i cicli $C_0, \dots, C_q$ generati dalla ricorsione ad ogni contrazione.
- Durante la risalita della ricorsione, ogni contrazione viene invertita, sostituendo nella contrazione $H_i/V(C_j)$ il vertice $v_{C_{j-1}}$ con i vertici originali $V(C_{j-1})$. Una volta terminata la risalita, dunque, si avrà che $H_i \in$ `uncontrComponents`
- L'insieme finale restitutito dalla prima chiamata della ricorsione, dunque, corrisponderà a $\{comp(v_{H_1}), \dots, comp(v_{H_k})\}$

$\square$

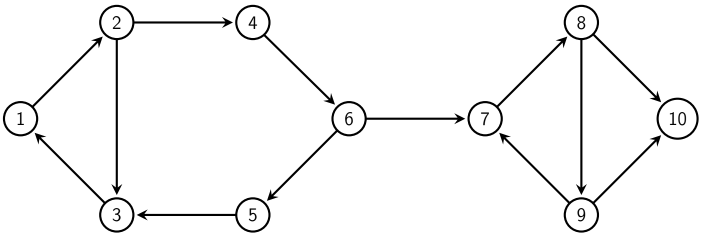
Dimostrazione costo algoritmo:

- Il costo dell'algoritmo 7 `findCycle()` proposto in precedenza è pari a $O(n+m)$
- Consideriamo quindi la contrazione del grafo G tramite la funzione `contractGraph()`. Per poter eliminare tutti i vertici in $V(C)$ e gli archi in $E(C)$, sostituendo quest'ultimi con gli archi connessi a v_c, nel caso peggiore è necessario scorrere tutte le liste di adiacenza di tutti i vertici, rendendo il costo di tale operazaione pari a $O(n+m)$
- Per quanto riguarda il ciclo for, invece, nel caso peggiore in cui ogni vertice di G sia un componente, si ha che $k = |V(G) = n|$. Inoltre, poiché ogni operazione all'interno del ciclo ha un costo $O(1)$, il costo dell'intero ciclo risulta essere $O(n)$
- Dunque, concludiamo che il costo di una singola chiamata ricorsiva sia $O(n+m) + O(n+m) + O(n) = O(n+m)$. Infine, poiché ad ogni ricorsione viene contratto un sottografo di G, ne segue che vi possano essere massimo n chiamate ricorsive, rendendo il costo totale dell'algoritmo pari a $O(n(n+m))$

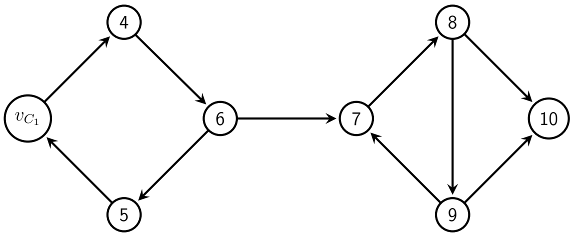
$\square$

Esempio:

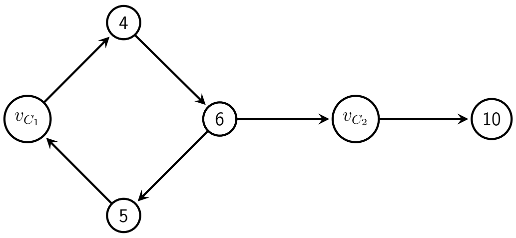
- Consideriamo il seguente grafo su cui applicheremo l'algoritmo precedente



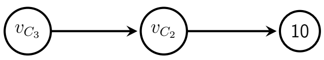
- Alla prima chiamata ricorsiva, viene trovato il ciclo $C_1 := \{1, 2, 3\}$, il quale viene contrarro nel vertice v_{C_1}



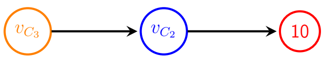
- Alla seconda chiamata ricorsiva, viene trovato il ciclo $C_2 := \{7, 8, 9\}$, il quale viene contratto nel vertice v_{C_2}

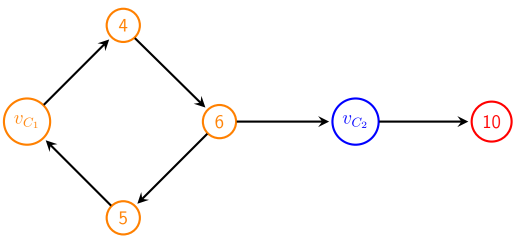


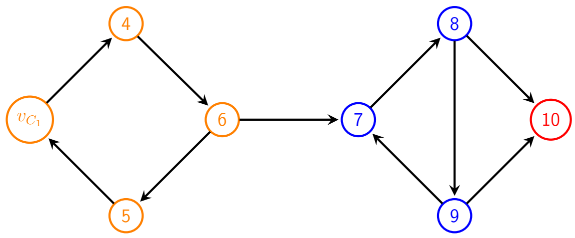
- Alla terza chiamata ricorsiva, viene trovato il ciclo $C_3 := \{v_{C_1}, 4, 5, 6\}$, il quale viene contratto nel vertice v_{C_3}

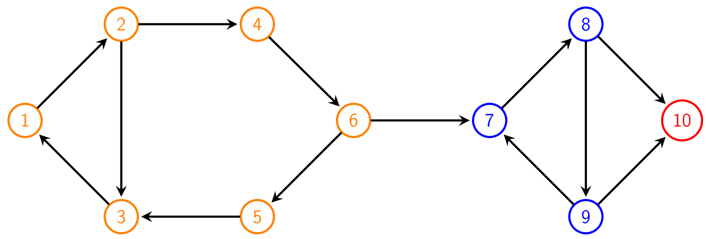


- A questo punto, raggiunto il caso base, i vertici rimanenti risultano essere le contrazioni massime dei componenti di G. Durante la risalita della ricorsione, decontraendo tali componenti otteniamo che:









- Dunque, l'output dell'algoritmo sarà $\{\{1, 2, 3, 4, 5, 6\}, \{7, 8, 9\}, \{10\}\}$

1.4.1 Algoritmo di Tarjan

Definition 26. C-Radice di un componente

Sia G un grafo e sia A un albero o un'arborescenza di visita generata da una DFS su G. Dato un vertice $v \in V(G)$, definiamo come **c-radice di** $comp(v)$ **in** A il vertice $u \in comp(v)$ visitato per primo dalla DFS

Proposition 17

Sia G un grafo diretto e sia A un'arborescenza di visita generata da una DFS su G. Dato $u \in V(G)$, se u è la c-radice di $comp(u)$ si ha che:

1. $V(comp(u)) \subseteq V(A_u)$, dove A_u è l'arborescenza dei discendenti di u in A
2. $V(A_u) = V(comp(u)) \cup V(comp(u_1)) \dots V(comp(u_k))$, dove $u_1, \dots, u_k$ sono le c-radici in G tali che $u_1, \dots, u_k \in V(A_u)$

Dimostrazioni:

1.
   - Per definizione stessa, si ha che $\forall v \in V(comp(u))$ esistono due cammini tali che $u \to v$ e $v \to u$. Di conseguenza, poiché $u \in V(A)$, ne segue necessariamente che v sia raggiungibile dalla DFS, dunque che $v \in V(A)$.
   - Inoltre, poiché u è la c-radice di $comp(u)$, ne segue che $t(u) < t(v)$. Nel caso assurdo in cui $t(u) \leq T(u) < t(v) \leq T(v)$, la DFS avrebbe sbagliato a non visitare v prima di rimuovere u dallo stack, poiché, essendo u c-radice, ogni vertice in $V(comp(u))$ non è stato ancora visitato. Di conseguenza, l'unica possibilità è che $t(u) < t(v) \leq T(v) \leq T(u)$

- Dunque, poiché $Int(v) \subseteq Int(u)$ e $v \in V(A)$, ne segue che v sia un discendente di u in A, implicando quindi che $V(comp(u)) \subseteq V(A_u)$

2.
- Dati $u_1, \ldots, u_k \in V(A_u)$, si ha che $A_{u_1}, \ldots, A_{u_k} \subseteq A_u$. Inoltre, poiché $u_1, \ldots, u_k$ sono rispettivamente c-radici di $comp(u_1), \ldots, comp(u_k)$, per la proposizione appena dimostrata si ha che

$$V(comp(u_i)) \subseteq V(A_{u_i}) \subseteq V(A_u), \forall i \in [1, k]$$

- Analogamente, per lo stesso motivo si ha che $V(comp(u)) \subseteq V(A_u)$. Di conseguenza, otteniamo che:

$$V(comp(u)) \cup V(comp(u_1)) \ldots V(comp(u_k)) \subseteq V(A_u)$$

- Viceversa, consideriamo $w \in V(A_u)$, implicando che esiste un cammino in $A_u \subseteq A \subseteq G$, tale che $u \to w$. Supponiamo che in G esista anche un cammino tale che $w \to u$. In tal caso, si avrebbe che $w \in V(comp(u))$.
- Supponiamo quindi che non esista tale cammino $w \to u$ in G. Poiché esiste un cammino $u \to w$ in A, ne segue che $\forall y \in V(comp(w))$ esiste un cammino tale che $u \to w \to y$, implicando che ogni vertice in $comp(w)$ possa essere raggiunto dalla DFS, dunque che $y \in V(A)$.
- Sia quindi $z \in V(comp(w))$ la c-radice di $comp(z) = comp(w)$. Supponiamo per assurdo che $z \in V(comp(w)) \cap V(A)$ ma che $z \notin V(A_u)$, implicando necessariamente che $t(z) < t(u)$, poiché altrimenti, dato che $u \to w \to z$, si avrebbe che $z \in V(A_u)$
- Poiché $w \in V(A_u)$, ne segue che $t(z) < t(u) < t(w) \leq T(w) \leq T(u)$, per cui si ha che:
  - Nel caso in cui $t(z) \leq T(z) < t(u) < t(w) \leq T(w) \leq T(u)$, la DFS avrebbe sbagliato a non visitare w prima che z venisse rimosso dallo stack, poiché $z \in comp(z) = comp(w) \implies z \to w$.
  - Nel caso in cui $t(z) < t(u) < t(w) \leq T(w) \leq T(u) \leq T(z)$, si avrebbe che $A_u \subseteq A_z$, dunque che esiste un cammino tale che $z \to u$. Tuttavia, poiché esiste anche un cammino tale che $u \to w \to z$, otterremmo che $u \in comp(u) = comp(z)$, contraddicendo l'ipotesi per cui u sia la c-radice di $comp(u)$

  Dunque, poiché ognuno dei due casi ipotetici crea una contraddizione, concludiamo che l'unica possibilità sia che $z \in V(A_u)$
- Di conseguenza, poiché z è una c-radice e $z \in V(A_u)$, per definizione stessa di $u_1, \ldots, u_k$ ne segue che $z \in \{u_1, \ldots, u_k\}$, da cui traiamo che:

$$w \in V(A_u) \implies w \in comp(w) = comp(z) = comp(u_i), \exists i \in [1, k] \mid z = u_i$$

- Infine, poiché $w \in V(comp(u))$ oppure $\exists i \in [1, k] \mid w \in V(comp(u_i))$, concludiamo che

$$V(A_u) \subseteq V(comp(u)) \cup V(comp(u_1)) \ldots V(comp(u_k))$$

□

Algorithm 15. Algoritmo di Tarjan

Sia G un grafo diretto rappresentato tramite liste di adiacenza. Il seguente algoritmo trova i componenti di G.

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

Algorithm 15: Trovare i componenti di un grafo diretto (Algoritmo di Tarjan)

Input:
G: grafo diretto a liste di adiacenza,
Comp: array dei componenti in G e delle visite aperte
c: Counter tempi di visita, cc: Counter ID componente
Output:
Array dei componenti in G

```
Function DFS_SCC(G, u, S, c, cc):
    c.increment();
    Comp[u] = −c;
    S.push(u);
    back = c;
    for v ∈ V(G) | (u, v) ∈ E(G) do
        if Comp[v] = 0 then
            back_v = DFS_SCC(G, v, Comp, S, c, cc);
            back = min(back, back_v);
        else if Comp[v] < 0 then
            back = min(back, -Comp[v]);
    end
    if back == -Comp[u] then
        cc.increment();
        w = S.pop();
        Comp[w] = cc;
        while w ≠ u do
            w = S.pop();
            Comp[w] = cc;
        end
    return back;
end
Function getComponents_2(G):
    Comp = [0, ..., 0];
    Counter c, cc := 0;
    Stack S = ∅;
    for u ∈ V(G) do
        if Comp[u] = 0 then
            DFS_SCC(G, u, Comp, S, c, cc);
    end
    return Comp;
end
```

Dimostrazione correttezza e costo algoritmo:

- Siano $r_1, \ldots, r_p \in V(G)$ i vertici non ancora visitati su cui viene chiamata non ricorsivamente la funzione `DFS_SCC`. Per la proposizione precedente, sappiamo che $V(A_{r_1} \cup \ldots \cup A_{r_p})$ conterranno necessariamente i vertici di tutti componenti di G.
- Sia quindi `Comp[]` un array tale che:
  - Se $x \in V(G)$ non è mai stato visitato, allora $\texttt{Comp[}x\texttt{] = 0}$
  - Se $x \in V(G)$ viene visitato per la prima volta, allora $\texttt{Comp[}x\texttt{] = } -t(x)$, dove $t(x)$ è il tempo di visita di x (corrispondente al valore del contatore c al momento della visita di x)
  - Se il componente di $x \in V(G)$ è stato già completamente determinato, allora $\texttt{Comp[}x\texttt{] = } \texttt{cc}_{comp(x)}$, dove $\texttt{cc}_{comp(x)}$ è il valore del contatore cc nel momento in cui viene determinato $comp(x)$
- Notiamo inoltre che lo stack `S` contiene tutti i vertici $y \in V(G)$ per cui ancora non è stato determinato $comp(y)$.
- Sia $u \in V(G)$ il vertice attualmente visitato dalla DFS. Dati i vertici $v_1, \ldots, v_k \in V(G)$ tali che $(u, v_i) \in E(G), \forall i \in [1, k]$, si ha che:
  - Se $\texttt{Comp[}v_i\texttt{] = 0}$, ne segue che $v_i \in V(A_u)$, dunque che v_i sia discendente di u
  - Se $\texttt{Comp[}v_i\texttt{] < 0}$, ne segue che v_i sia un vertice già visitato ma per cui non è ancora terminata la ricorsione, implicando che $u \in V(A_{v_i})$, dunque che v_i sia un antenato di u
  - Se $\texttt{Comp[}v_i\texttt{] > 0}$, ne segue che $\texttt{Comp[}v_i\texttt{] = } \texttt{cc}_{comp(v_i)}$, implicando che $u \notin V(comp(v_i))$, venendo quindi direttamente saltato

  dove A è l'arborescenza di visita generata dalla DFS
- Siano quindi $w_1, \ldots, w_j \in V(G)$ gli antenati di u e siano $v'_1, \ldots, v'_h \in V(A_u)$.

  Dato `back` il tempo di visita dell'antenato w di u, dunque $u \in V(A_w)$, dove il componente $comp(w)$ non è ancora stato determinato dall'algoritmo e $\exists (v, w) \in E(G)$, ne segue che

  $$\mathsf{back} = \min\{t(w_1), \ldots, t(w_j), t(u), \mathsf{back}_{v'_1}, \ldots, \mathsf{back}_{v'_h}\}$$
- Dato $v \in \{v'_1, \ldots, v'_h\}$, dimostriamo che

  $$u \text{ non è c-radice} \iff \exists (v, w) \in E(G)$$
  - Sia $\alpha \neq u \in V(comp(u))$ la c-radice di $comp(u) = comp(\alpha)$, implicando che esistono due cammini tali che $u \to \alpha$ e $\alpha \to u$. Per la proposizione precedente, si ha che $V(comp(u)) = V(comp(\alpha)) \subseteq V(A_\alpha)$.
  - Poiché $\alpha \in V(A_\alpha - A_u)$ e $u \in V(A_u)$, affinché $u \to \alpha$ e $\alpha \to u$ ne segue necessariamente che $\exists (v, w) \in E(G)$, dove $v \in V(comp(\alpha) \cap A_u), w \in V(comp(\alpha) \cap (A_\alpha - A_u)) = comp(\alpha)$

    Inoltre, poiché u è il vertice attualmente visitato, ne segue che il componente $comp(u) = comp(v) = comp(w) = comp(\alpha)$ non sia stato ancora determinato, implicando quindi che $\texttt{Comp[}w\texttt{] < 0}$, dunque che $u \in V(A_w)$

  - Viceversa, supponiamo che esista un arco $(v, w) \in E(G)$ dove $v \in V(A_u)$ e $w \in V(A - A_u)$ è un antenato di u per cui il componente $comp(w)$ non è ancora stato determinato.
  - Sia $z \in comp(w)$ la c-radice di $comp(w)$. Poiché $comp(w) = comp(z)$ non è ancora stato determinato, ne segue che `Comp[`z`] < 0`, dunque che non sia ancora terminata la ricorsione sui discendenti di z. Inoltre, poiché z è c-radice di $comp(w)$, si ha che $A_u \subseteq A_w \subseteq A_z$.
  - Di conseguenza, esistono due cammini tali che $u \to v \to w \to z$ e $z \to w \to u$, implicando che $comp(u) = comp(z)$ e dunque che u non sia c-radice

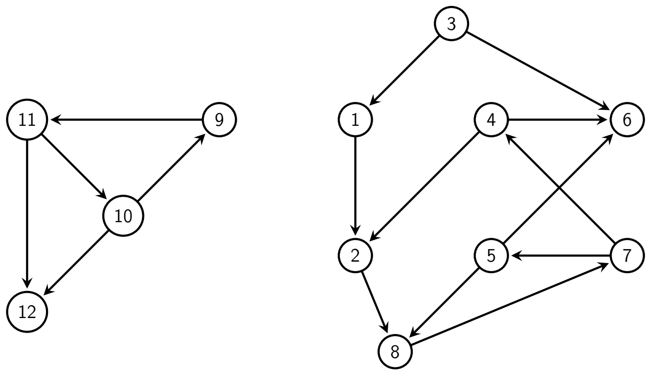
- A questo punto, nel caso in cui `back` $=$ `-Comp[`u`]` $= t(u)$, ne segue che $\nexists (v, w) \in E(G)$, implicando quindi che u sia la c-radice di $comp(u)$.
- Per la proposizione precedente, sappiamo che $V(comp(u)) \subseteq V(A_u)$.

  Nel caso in cui $V(comp(u)) = V(A_u)$, ne segue che tutti i vertici $v'_1, \ldots, v'_h$ aggiunti allo stack `S` dopo u siano i vertici interni a $comp(u)$. Di conseguenza, il componente $comp(u)$ viene determinato e vengono posti `Comp[`u`] = Comp[`v'_1`] = ... = Comp[`v'_h`] = cc`$_{comp(u)}$
- Se invece $V(comp(u)) \subseteq V(A_u)$, ma $V(comp(u)) \neq V(A_u)$, tutti i vertici non appartenenti a $comp(u)$ saranno già stati tolti dallo stack `S`, implicando che i vertici rimanenti $v''_1, \ldots, v''_i$ inseriti dopo u siano i vertici interno a $comp(u)$, ponendo quindi `Comp[`u`] = Comp[`v''_1`] = ... = Comp[`v''_h`] = cc`$_{comp(u)}$
- Infine, trattandosi di una DFS modificata con solo operazioni in $O(1)$ aggiunte, il costo dell'algoritmo risulta essere $O(n + m)$

□

Esempio:

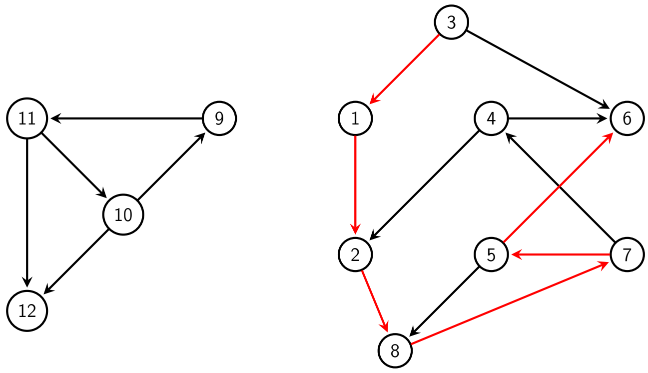
- Consideriamo il seguente grafo diretto, lo stack `S` e l'array `Comp` dell'algoritmo precedente



$$\texttt{Comp} = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]$$

$$\texttt{S} = [\,]$$

- Eseguiamo la prima DFS dell'algoritmo, partendo dal vertice 3



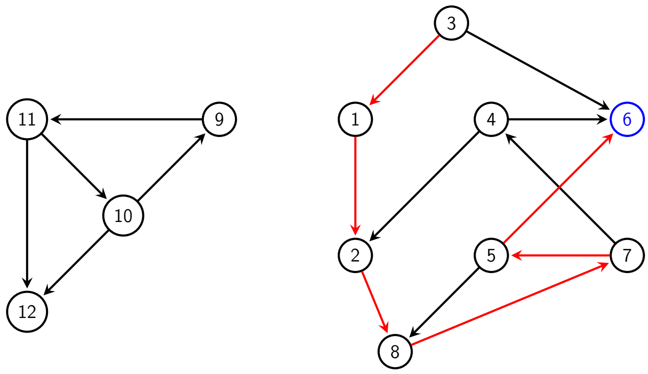
$$\texttt{Comp} = [-2, -3, -1, 0, -6, -7, -5, -4, 0, 0, 0, 0]$$

$$\texttt{S} = [3, 1, 2, 8, 7, 5, 6]$$

(ricordiamo che $\texttt{Comp[}x\texttt{] < 0} \implies \texttt{Comp[}x\texttt{] = } -t(x)$)

- Poiché il vertice 6 non ha archi uscenti, ne segue che $\texttt{back}_6 = t(6) = 7$, implicando quindi che $\texttt{back}_6 = -\texttt{Comp[6]}$ e dunque che 6 sia c-radice di *comp*(6).

  Di conseguenza, tutti i vertici aggiunti allo stack dopo 6 (ossia nessun altro vertice) appartiene a *comp*(6), venendo rimossi dallo stack e marchiati con valore attuale del contatore *cc*, ossia 1



$$\texttt{Comp} = [-2, -3, -1, 0, -6, 1, -5, -4, 0, 0, 0, 0]$$

$$\texttt{S} = [3, 1, 2, 8, 7, 5]$$

- Proseguiamo quindi la DFS ricorsiva avente 3 come radice, tornando al vertice 5. Poiché $(5, 8) \in E(G)$ e 8 è già stato visitato ma non chiuso, ne segue che $\texttt{back}_5 = t(8) = 4$, ritornando tale valore alla chiamata precedente, ossia la visita del vertice 7.

  Inoltre, poiché $\texttt{back}_5 = t(8) = 4 < \texttt{back}_7 = t(7) = 5$, viene sovrascritto $\texttt{back}_7 = t(8)$.

- Successivamente, la DFS procederà verso il vertice 4, dunque si ha che

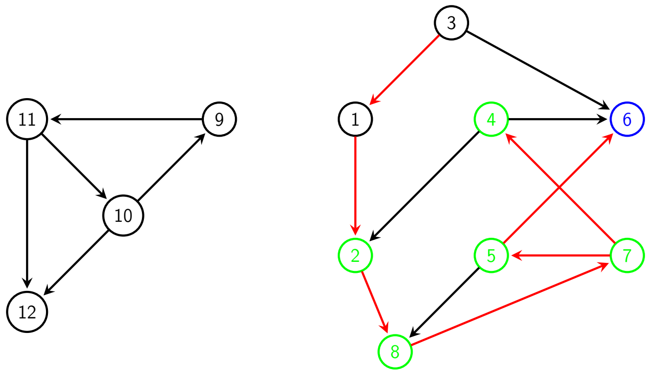
$$\texttt{Comp} = [-2, -3, -1, -8, -6, 1, -5, -4, 0, 0, 0, 0]$$

$$\texttt{S} = [3, 1, 2, 8, 7, 5, 4]$$

- Poiché $(4, 2) \in E(G)$ e 2 è stato già visitato ma non ancora chiuso, si ha che $\texttt{back}_4 = t(2) = 3$, ritornando tale valore al vertice 7 e sovrascrivendo $\texttt{back}_7 = t(2)$ poiché $\texttt{back}_4 = t(2) = 3 < \texttt{back}_7 = t(8) = 4$.

  Poiché 7 non ha altri vertici adiacenti, il valore $\texttt{back}_7 = t(2)$ viene ritornato alla visita del vertice 8. Poiché $\texttt{back}_7 = t(2) = 3 < \texttt{back}_8 = t(8) = 4$, viene sovrascritto $\texttt{back}_8 = t(2)$. Analogamente, poiché 8 non ha altri vertici adiacenti, il valore $\texttt{back}_8 = t(2)$ viene ritornato alla visita del vertice 2. Poiché $\texttt{back}_8 = t(2) = \texttt{back}_2 = -\texttt{Comp[2]}$, l'algoritmo determina che 2 sia la c-radice di *comp*(2).

  Di conseguenza, tutti i vertici aggiunti allo stack dopo 2, ossia 8, 7, 5 vengono rimossi dallo stack e marchiati con valore attuale del contatore *cc*, ossia 2

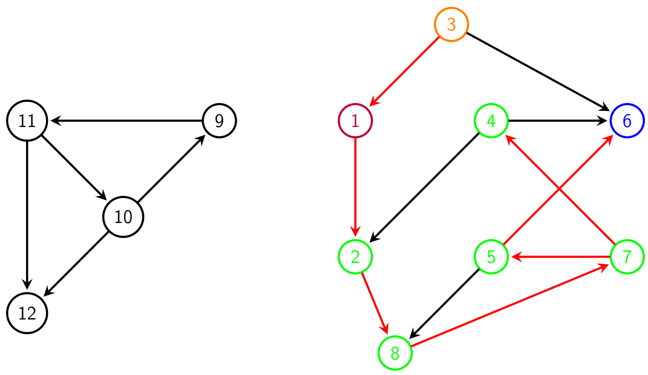


$$\texttt{Comp} = [-2, 2, -1, 2, 2, 1, 2, 2, 0, 0, 0, 0]$$

$$\texttt{S} = [3, 1]$$

- Una volta tornati alla visita di 1, poiché esso non ha altri vertici adiacenti ne segue che $\texttt{back}_1 = -\texttt{Comp[1]}$, dunque 1 è la c-radice di *comp*(1), rimuovendo dallo stacak e marchiando con $cc = 3$ gli elementi inseriti dopo di esso nello stack (nessuno)

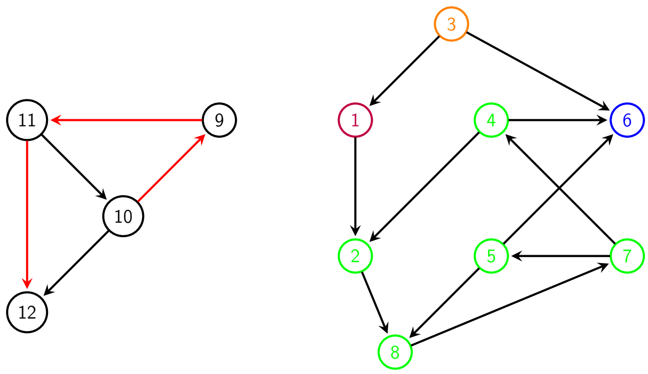
  Analogamente, 3 risulta essere la c-radice di 3, dunque si ha che.



$$\texttt{Comp} = [3, 2, 4, 2, 2, 1, 2, 2, 0, 0, 0, 0]$$

$$\texttt{S} = [\,]$$

- A questo punto, la DFS avente 3 come radice termina. Di conseguenza, viene effettuata una nuova DFS su uno dei vertici non ancora esplorati (sceglieremo il vertice 10).

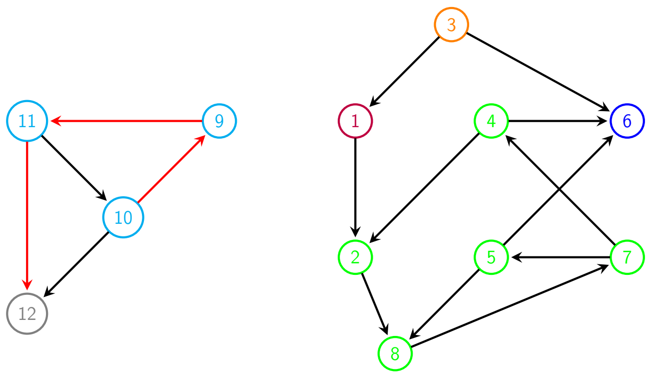


$$\texttt{Comp} = [3, 2, 4, 2, 2, 1, 2, 2, -10, -9, -11, -12]$$

$$\texttt{S} = [10, 9, 11, 12]$$

- Analogamente a $1, 3$ e 6, il vertice 12 risulta essere la c-radice di $comp(12)$, marchiando i suoi elementi con $cc = 5$.

  Infine, analogamente al vertice 2, il vertice 10 viene decretato c-radice di $comp(10)$, marchiando i suoi elementi con $cc = 6$



$$\texttt{Comp} = [3, 2, 4, 2, 2, 1, 2, 2, 6, 6, 6, 5]$$

$$\texttt{S} = [\,]$$

- Poiché non vi sono altri vertici visitabili, tramite l'array `Comp` concludiamo che
  - $comp(1) = \{1\}$
  - $comp(2) = \{2, 4, 5, 7, 8\}$
  - $comp(3) = \{3\}$
  - $comp(10) = \{9, 10, 11\}$
  - $comp(12) = \{12\}$

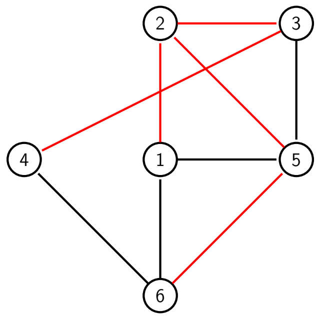
1.5 Breadth-first Search (BFS)

Definition 27. Breadth-first Search (BFS)

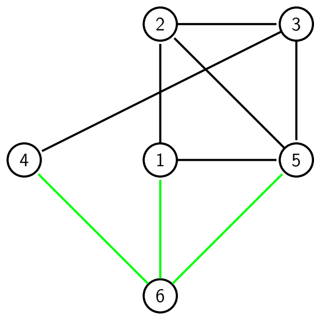
Sia G un grafo. Dato un vertice iniziale $x \in V(G)$ definiamo come **breadth-first search (BFS)** un **criterio di visita** su G basato sul procedere **in ampiezza**, ossia dando precedenza ai vertici più vicini dal vertice iniziale, raggiungendo ogni vertice **una ed una sola volta**, procedendo con il vertice successivo se e solo se non è più possibile procedere in ampiezza tramite il vertice attuale, ossia quando tutti i vertici adiacenti sono già stati visitati

Esempio:

- Dato il seguente grafico, eseguendo una DFS sul vertice 6 otterremmo la seguente arborescenza con il seguente ordine di visita $[6, 5, 2, 3, 4, 1]$



- Eseguiamo invece una BFS su 6, dando quindi precedenza a tutti i suoi archi uscenti, ossia $(6, 5), (6, 1), (6, 4)$



- A questo punto, procediamo con il prossimo vertice, ossia 5, dando quindi precedenza agli archi $(5, 2), (5, 3)$. A questo punto, poiché non vi sono altri vertici visitabili, la BFS termina con il seguente ordine di visita $[6, 5, 1, 4, 2, 3]$

Definition 28. Distanza tra vertici

Sia G un grafo. Dati due vertici $x, y \in V(G)$, definiamo come **distanza tra x e y**, indicata come $dist(x, y)$, il numero minimo di archi appartenenti ad un cammino tale che $x \to y$

Se non esiste un cammino tale che $x \to y$, diciamo che $dist(x, y) = +\infty$

Proposition 18

Sia G un grafo. Dato un vertice $x \in V(G)$, si ha che:

$$\forall y \neq x \in V(G), \exists z \in V(G) \mid (z, y) \in E(G), dist(x, y) = dist(x, z) + 1$$

Dimostrazione:

- Sia $dist(x, y) = L$, implicando che esista un cammino $xe_1v_1e_2 \dots e_{L-1}v_{L-1}e_Ly$ di lunghezza minima tale che $x \to y$.
- Posti $z := v_{L-1}$ e $k := dist(x, z)$, se per assurdo $xe_1v_1e_2 \dots e_{L-1}z$ non fosse un cammino di lunghezza minima tale che $x \to z$, esisterebbe un cammino di lunghezza minima $xh_1u_1e_2 \dots e_kz$ tale che $x \to z$ dove $k < L - 1$, implicando che $xh_1u_1h_2 \dots h_kze_Ly$ sia un cammino di lunghezza $k + 1 < L$ tale che $x \to y$, contraddicendo l'ipotesi per cui $dist(x, y) = L$
- Di conseguenza, l'unica possibilità è che

$$dist(x, z) = L - 1 = dist(x, y) - 1 \implies dist(x, y) = dist(x, z) + 1$$

□

Definition 29. Livello di un vertice

Sia T un albero radicato o un'arborescenza radicata, dove $u \in V(G)$ è la radice. Dato un vertice $v \in V(G)$, definiamo $L := dist(u, v)$ come **livello di v**

Algorithm 16. Breadth-first Search

Sia G un grafo rappresentato tramite liste di adiacenza. Dato un vertice $x \in V(G)$, il seguente algoritmo restituisce la distanza $dist(x, y)$ per ogni vertice $y \in V(G)$ raggiunto da una BFS su x e il vettore dei padri rappresentante l'albero/arborescenza di visita generato

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

Algorithm 16: Breadth-first Search

Input:
G: grafo a liste di adiacenza,
u: $u \in V(G)$,
Output:
Distanze dalla radice dei vertici visitati e albero/arborescenza di visita tramite vettore dei padri

```
Function BFS(G, u):
    Padri = [-1, ..., -1];
    Dist = [-1, ..., -1];
    Queue Q = ∅;
    Q.enqueue(u);
    Padri[u] = u;
    Dist[u] = 0;
    while Q ≠ ∅ do
        v = Q.dequeue();
        for x ∈ v.uscenti do
            if Dist[x] = −1 then
                Padri[x] = v;
                Dist[x] = Dist[v] + 1;
                Q.enqueue(x);
            end
        end
    end
    return Dist, Padri;
end
```

Dimostrazione correttezza e costo algoritmo:

- Posto `Dist[u] = 0`, la radice u è il primo vertice inserito e rimosso dalla queue.
- Siano quindi $x_1, \ldots, x_k \in V(G)$ i vertici adiacenti alla radice tali che $(u, x_i) \in E(G), \forall i \in [1, k]$, implicando che $dist(u, x_i) = 1, \forall i \in [1, k]$. Poiché `Dist[u] = 0`, ne segue che `Dist[`x_i`] = Dist[u] + 1 = 1`, $\forall i \in [1, k]$
- Assumiamo quindi per ipotesi induttiva che `Dist[v]` $=$ $dist(u, v)$ per ogni vertice $v \in V(G)$ rimosso dalla queue.
- Dato il vertice $z \in V(G)$ tale che $dist(u, z) = k + 1$, per la proposizione precedente $\exists y \in V(G) \mid (y, z) \in E(G), dist(u, z) = dist(u, y) + 1$, implicando che z verrà visitato dalla BFS nel momento in cui y verrà rimosso dalla queue.
- Per ipotesi induttiva, dunque, ne segue che `Dist[y]` $= dist(u, y) = dist(u, z) - 1 = k$, implicando quindi che `Dist[z] = Dist[y] + 1 = k+1` $= dist(u, z)$
- Il calcolo del costo computazionale di una BFS risulta analogo al calcolo per una DFS. Di conseguenza, il costo dell'algoritmo è $O(n + m)$

□

Algorithm 17. Trovare numero di cammini minimi dalla radice

Sia G un grafo rappresentato tramite liste di adiacenza. Dato un vertice $x \in V(G)$, il seguente algoritmo restituisce il numero di cammini di lunghezza minima tale che $x \to y$ per ogni vertice $y \in V(G)$ raggiunto dalla BFS

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

Algorithm 17: Trovare numero di cammini minimi dalla radice

Input:
G: grafo a liste di adiacenza,
u: $u \in V(G)$,
Output:
Numero di cammini minimi dalla radice di ogni vertice visitabile

```
Function countMinPaths(G, u):
    Dist = [-1, ..., -1];
    Count = [0, ..., 0];
    Queue Q = ∅;
    Q.enqueue(u);
    Dist[u] = 0;
    Count[u] = 1;
    while Q ≠ ∅ do
        v = Q.dequeue();
        for x ∈ v.uscenti do
            if Dist[x] = −1 then
                Dist[x] = Dist[v] + 1;
                Count[x] = Count[v];
                Q.enqueue(x);
            else if Dist[v] = Dist[x] - 1 then
                Count[x] += Count[v];
        end
    end
    return Count;
end
```

Dimostrazione correttezza algoritmo (costo omesso):

- Per ogni vertice $y \in V(G)$, sia `Dist[`y`]` = $dist(u, y)$ e sia `Count[`y`]` il numero di cammini di lunghezza minima tali che $u \to y$
- Dato un vertice $x \in V(g)$, siano $v_1, \dots, v_k \in V(G)$ dei vertici tali che $\forall i \in [1, k], \exists (v_i, x) \in E(G)$ e `Dist[`v_i`] = Dist[`x`] - 1`
- Poiché `Dist[`v_i`] = Dist[`x`] - 1` = $dist(u, x)$ `-1`, è possibile estendere ognuno di tali cammini con un l'arco (v_i, x) per ottenere dei cammini di lunghezza pari a $dist(u, x)$, corrispondenti ai cammini minimi tali che $u \to x$. Di conseguenza, si ha che `Count[`x`]` $=$ `Count[`v_1`]` $+ \dots +$ `Count[`v_k`]`

□

Definition 30. Distanza tra sottoinsiemi di vertici

Sia G un grafo. Dati due sottoinsiemi di vertici $X, Y \subseteq V(G)$, definiamo come **distanza tra X e Y** il valore

$$dist(X, Y) = \min_{x \in X y \in Y} dist(x, y)$$

Algorithm 18. Trovare la distanza tra due sottoinsiemi di vertici

Sia G un grafo rappresentato tramite liste di adiacenza. Dati due sottoinsiemi di vertici $X, Y \subseteq V(G)$, il seguente algoritmo restituisce $dist(X, Y)$

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

Algorithm 18: Trovare la distanza tra due sottoinsiemi di vertici

Input:
G: grafo a liste di adiacenza,
X: $X \subseteq V(G)$,
Y: $Y \subseteq V(G)$,
Output:
Numero di cammini minimi dalla radice di ogni vertice visitabile

```
Function distBetweenSubsets(G, X, Y):
    Dist = [-1, ..., -1];
    Queue Q = ∅;
    for x ∈ X do
        Q.enqueue(x);
        Dist[x] = 0;
    end
    while Q ≠ ∅ do
        u = Q.dequeue();
        for v ∈ u.uscenti do
            if Dist[v] = −1 then
                Dist[v] = Dist[u] + 1;
                Q.enqueue(x);
            end
        end
    end
    minDist = +∞;
    for y ∈ Y do
        if Dist[y] < minDist then
            minDist = Dist[y];
        end
    end
    return minDist;
end
```

Dimostrazione correttezza algoritmo (costo omesso):

- Siano $x_1, \dots, x_k \in X$, dove $|X| = k$. Una volta aggiunti tali vertici alla queue `Q` e posto `Dist[`x_i`] = 0`, $\forall i \in [1, k]$, il risultato ottenuto alla fine del ciclo while è un'unione disgiunta di BFS eseguite sui vertici $x_1, \dots, x_k$
- Di conseguenza, si ha che `Dist[`z`]` $= dist(x, z), \exists x \in X$, ossia la distanza tra un vertice qualsiasi del sottoinsieme X al vertice z
- In particolare, dati i vertici $y_1, \dots, y_h \in Y$ dove $|Y| = h$, si ha che

$$\texttt{minDist} = dist(X, Y) = \min(+\infty, \{dist(x_i, y_j) \mid x_i \in X, y_j \in Y\}) =$$

$$= \min(+\infty, \texttt{Dist[}y_1\texttt{]}, \dots, \texttt{Dist[}y_h\texttt{]})$$

implicando che `minDist =`$+\infty$ se non $\nexists$ cammino $x' \to y'$ tale che $x' \in X, y' \in Y$

□

1.6 Grafi pesati

Definition 31. Peso di un arco

Sia G un grafo. Dato un arco $e \in E(G)$, definiamo come **peso di** e il valore reale positivo $w(e)$, dove

$$w : E(G) \to \mathbb{R}^+ : e \mapsto w(e)$$

Definition 32. Peso di un cammino

Sia G un grafo. Data la funzione dei pesi $w : E(G) \to \mathbb{R}^+$ e un cammino $P := v_1 e_1 \dots e_k v_k$, definiamo come **peso del cammino** il valore reale positivo $w_p(P)$, dove

$$w_p(P) = \sum_{i=1}^{k} w(e_i)$$

Definition 33. Distanza pesata

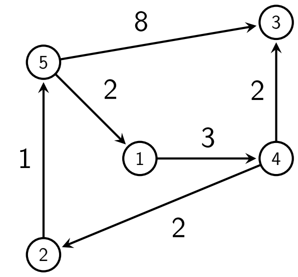
Sia G un grafo pesato. Dati due vertici $x, y \in V(G)$, definiamo come **distanza pesata** il peso del cammino $x \to y$ avente peso minimo

$$dist_w(x, y) = \min_{P \in \mathcal{C} \mid x \to y} w_p(P)$$

dove $\mathcal{C}$ è l'insieme di tutti i cammini su G

Esempio:

- Consideriamo il seguente grafo pesato



- Consideriamo il cammino $P := 5(5,1)1(1,4)4(4,3)3$, avente un peso equivalente a

$$w_p(P) = w(5,1) + w(1,4) + w(4,3) = 2 + 3 + 2 = 7$$

- Poiché il peso dell'arco $(5,3)$ è pari a $w(5,3) = 8$, ne segue che $w_p(P) < w(5,3)$ e di conseguenza che P sia il cammino di peso minimo tale che $5 \to 3$, implicando che $dist_w(5,3) = w_p(P) = 7$

Observation 13. Proprietà delle distanze pesate

Dato un grafo pesato G, si ha che:

1. $\forall x \in V(G), dist_w(x,x) = 0$
2. $\forall x, y \in V(G), dist_w(x,y) \geq 0$
3. $\forall x, y \in V(G)$ non è sempre vero che $dist_w(x,y) \neq dist_w(y,x)$

Lemma 19. Disuguaglianza triangolare

Sia G un grafo pesato. Dati tre vertici $x, y, z \in V(G)$, si ha che:

$$dist_w(x,y) \leq dist_w(x,z) + dist_w(z,y)$$

Dimostrazione:

- Sia P_1 un cammino di peso minimo tale che $x \to z$, implicando che $dist_w(x,z) = w_p(P_1)$, e sia P_1 un cammino di peso minimo tale che $z \to y$, implicando che $dist_w(z,y) = w_p(P_2)$
- La passeggiata $P_1 \cup P_2$ tale che $x \to z$, la quale non è detto che sia un cammino poiché P_1 e P_2 potrebbero avere vertici in comune, ha un peso equivalente a:

$$w_p(P_1 \cup P_2) = w_p(P_1) + w_p(P_2) = dist_w(x,z) + dist(z,y)$$

- Sia quindi Q un cammino di peso minimo tale che $x \to z$. Poiché un cammino non contiene vertici ridondanti e poiché $P_1 \cup P_2$ è una passeggiata tale che $x \to z$, ne segue che:

$$dist(x,y) = w_p(Q) \leq w_p(P_1 \cup P_2) = dist_w(x,z) + dist(z,y)$$

$\square$

Lemma 20

] Sia G un grafo non diretto pesato. Dato un vertice $u \in V(G)$, sia $N(u) := \{v \in V(G) \mid (u,v) \in E(G)\}$ il sottoinsieme di vertici adiacenti a u

Dato $x \in N(u)$ tale che l'arco (u,x) abbia peso minimo tra tutti gli altri vertici in $N(u)$

$$x \in N(u) \mid w(u,x) = \min_{v' \in N(u)} w(u,v')$$

si ha che

$$dist_w(u,x) = w(u,x)$$

Dimostrazione:

- Sia $P := ue_1v_1e_2 \dots e_kx$ un qualsiasi cammino tale che $u \to x$
- Dato il cammino $Q := u(u,x)x$, per definizione stessa di x, si ha che

$$w_p(Q) = w(u,x) < w(u,v_1) < w_p(P)$$

- Di conseguenza, il cammino Q è il cammino di peso minore tale che $u \to x$, implicando che

$$dist_w(u,x) = w_p(Q) = w(u,x)$$

□

Theorem 21

Sia G un grafo non diretto pesato e sia $R \subseteq V(G)$. Dati il vertice $u \in R$ e l'arco $(x,v) \in E(G)$ con $x \in R$ e $v \in V(G) - R$ minimizzante la somma $dist_w(u,a) + w(a,b)$ dove $a \in R, b \in V(G) - R$

$$(x,v) \in E(G) \mid dist_w(u,x) + w(x,v) = \min_{\substack{(a,b) \in E(G) \text{ t.c.} \\ a \in R, b \in V(G)-R}} dist_w(u,a) + w(a,b)$$

si ha che

$$dist_w(u,v) = dist_w(u,x) + w(x,v)$$

Dimostrazione:

- Prima di tutto, analizziamo il cammino P tale che $u \to v$ passante per tale arco (x,v):
  - Dato il vertice $x \in R$, consideriamo il cammino di peso minimo P' tale che $u \to x$.
  - Per il lemma precedente, dato il vicino $v \in N(x)$ tale che l'arco (x,v) abbia peso minimo tra tutti gli altri vertici in $N(x)$, si ha che $dist_w(x,v) = w(x,v)$
  - Di conseguenza, per il lemma della distanza triangolare si ha che

$$P = P' \cup (x,v) \implies w_p(P) = dist_w(u,x) + dist_w(x,v) \geq dist_w(u,v)$$

- Supponiamo quindi per assurdo che $dist_w(u, v) < w_p(P)$, implicando che esista un cammino Q di peso $w_p(Q) = dist_w(u, v)$ tale che $u \to v$.
- Poiché $u \in R$ e $v \in V(G) - R$, esiste necessariamente un arco $(x', v') \in Q \mid x' \in R, v' \in V(G) - R$ per cui si abbia un cammino Q' tale che $u \to x'$ ed un cammino Q'' tale che $v' \to v$, implicando quindi che $Q = Q' \cup (x', v') \cup Q''$.
- Di conseguenza, si ha che

  $$dist_w(u, x') + w(x', v') \leq w_p(Q') + w(x', v') < w_p(Q) < w_p(P) = dist_w(u, x) + w(x, v)$$

  implicando quindi che, poiché $x' \in R$ e $v' \in V(G) - R$, l'arco (x', v') sia l'arco minimizzante la somma $dist_w(u, a) + w(a, b)$ dove $a \in R, b \in V(G) - R$, contraddicendo quindi l'ipotesi per cui (x, v) sia tale arco
- Di conseguenza, l'unica possibilità è che si abbia anche $dist_w(u, v) \geq w_p(P)$, da cui concludiamo che $dist_w(u, v) = w_p(P)$

□

1.6.1 Algoritmo di Dijkstra

Algorithm 19. Algoritmo di Dijkstra

Sia G un grafo non diretto pesato e rappresentato tramite liste di adiacenza. Dato un vertice $u \in V(G)$, il seguente algoritmo restituisce le distanze pesate da u per ogni vertice $v \in V(G)$ tale che $u \to v$

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

Algorithm 19: Algoritmo di Dijkstra

Input:
G: grafo non diretto pesato a liste di adiacenza,
u: $u \in V(G)$,
Output:
Distanze pesate dalla radice u dei vertici raggiungibili

```
Function dijkstra(G, u):
    Dist = [+∞, ..., +∞];
    R = u;
    Dist[u] = 0;
    while ∃(a,b) ∈ E(G) | a ∈ R, b ∈ V(G) − R do
        (x,y) = arco con x ∈ R, y ∈ V(G) − R che minimizza Dist[x]+w(x,y);
        Dist[y] = Dist[x] + w(x,y);
        R.add(y);
    end
    return Dist;
end
```

Dimostrazione correttezza algoritmo:

- Siano $R_0, \dots, R_k$ e $\texttt{Dist}_0, \dots, \texttt{Dist}_k$ le istanze dell'insieme R e l'array `Dist` ad ogni iterazione del while. Dato $R_0 = \{u\}$, per il lemma precedente si ha che

$$x \in N(u) \mid w(u,x) = \min_{v \in N(u)} w(u,v) \implies dist_w(u,x) = w(u,x)$$

Inoltre, poiché $\texttt{Dist}_0\texttt{[}u\texttt{] = 0 = } dist(u,u)$, ne traiamo che

$$\texttt{Dist}_1\texttt{[}x\texttt{] = Dist}_0\texttt{[}u\texttt{] + } w(u,x) \texttt{ = } w(u,x) \texttt{ = } dist_w(u,x)$$

- Supponiamo quindi per ipotesi induttiva che $\forall v \in R_i$ si abbia che $\texttt{Dist}_i\texttt{[}v\texttt{] = } dist_w(u,v)$. Dato l'insieme R_{i+1}, supponiamo che $\exists (a,b) \in E(G) \mid a \in R, b \in V(G) - R$ e, in particolare, che esista l'arco

$$(x,y) \in E(G) \mid dist_w(u,x) + w(x,y) = \min_{\substack{(a,b) \in E(G) \text{ t.c.} \\ a \in R, b \in V(G)-R}} dist_w(u,a) + w(a,b)$$

da cui, per il teorema precedente, otteniamo che

$$dist_w(u,y) = dist_w(u,x) + w(x,y) = \texttt{Dist}_i\texttt{[}x\texttt{]} + w(x,y) = \texttt{Dist}_{i+1}\texttt{[}y\texttt{]}$$

Di conseguenza, l'array $\texttt{Dist}_k$ conterrà le distanze pesate dalla radice u

□

Dimostrazione costo algoritmo:

- Ad ogni iterazione i del ciclo while, per trovare l'arco con $x \in R, y \in V(G) - R$ che minimizza $\texttt{Dist}_i\texttt{[}x\texttt{]+}w(x,y)$, è necessario controllare gli archi incidenti ad ogni nodo appartenente ad R_i. Inoltre, poiché $R_i \subseteq V(G), \forall i \in [1,k]$, si ha che:

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

implicando quindi che il costo di tale operazione sia $O(m)$

- Nel caso peggiore, inoltre, all'interno dell'istanza finale R_k sono stati aggiunti tutti i vertici di $V(G)$, implicando che siano state svolte $|V(G)| = n$ iterazioni. Di conseguenza, il costo totale dell'algoritmo è $n \cdot O(m) = O(nm)$

□

Algorithm 20. Algoritmo di Dijkstra (Ottimizzato)

Sia G un grafo non diretto pesato e rappresentato tramite liste di adiacenza. Dato un vertice $u \in V(G)$, il seguente algoritmo restituisce le distanze pesate da u per ogni vertice $v \in V(G)$ tale che $u \to v$

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

Algorithm 20: Algoritmo di Dijkstra (Ottimizzato)

Input:
G: grafo non diretto pesato a liste di adiacenza,
u: $u \in V(G)$,
Output:
Distanze pesate dalla radice u dei vertici raggiungibili

```
Function dijkstra_2(G, u):
    Dist = [+∞, ..., +∞];
    Min-Heap H = ∅;
    Dist[u] = 0;
    for u ∈ V(G) do
        H.insert(u, Dist[u])        //Dist[u] è la chiave di u nell'heap;
    end
    while H ≠ ∅ do
        v = H.extract_min();
        Dist[v] = H.key(v);
        for x ∈ v.adiacenti do
            if x ∈ H and H.key(v) > Dist[v] + w(v, x) then
                H.update_key(x, Dist[v] + w(v, x));
        end
    end
    return Dist;
end
```

Dimostrazione costo algoritmo (correttezza omessa):

- All'interno di un min-heap, le operazioni di ricerca di un nodo, estrazione del minimo e aggiornamento di un valore hanno tutte un costo computazionale pari a $O(\log k)$, dove k è il numero di nodi nell'heap.
- Poiché all'interno del ciclo for esterno al while vengono inseriti nell'heap tutti $u \in V(G)$, ne segue che il costo di tali operazioni sia $O(\log n)$ e che, in particolare, il costo di tale for stesso sia $O(n \log n)$
- Per quanto riguarda il ciclo while, invece, ogni nodo viene estratto ed analizzato dall'heap esattamente una volta, implicando che ogni vertice in $V(G)$ venga analizzato esattamente una volta.
- In particolare, ad ogni iterazione del suo ciclo for interno, vengono controllati tutti i vertici adiacenti al vertice v attualmente analizzato, per un totale di $deg(v)$ controlli. Inoltre, poiché l'if interno a tale for ha in costo pari a $O(\log n)$, ne segue che il costo totale di tale ciclo while sia

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

- Infine, concludiamo che il costo totale dell'algoritmo sia

$$O(n \log n) + O(n \log m) = O((n + m) \log n)$$

□

Capitolo 2

Algoritmi Greedy

2.1 Definizione e scheletro di dimostrazione

Definition 34. Algoritmo Greedy

Un algoritmo viene detto **greedy** se cerca una soluzione ammissibile da un punto di vista globale attraverso la scelta della soluzione **più conveniente ad ogni passo locale**

Observation 14

Sebbene spesso gli algoritmi greedy risultino essere molto brevi e coincisi, è sempre necessario dimostrare che l'output generato sia una **soluzione ottimale**, ossia una soluzione rispettante la richiesta dell'algoritmo

Proposition 22. Scheletro di dimostrazione per algoritmi greedy

Il seguente **scheletro di dimostrazione** fornisce una base per dimostrare la correttezza di un algoritmo greedy:

1. Dimostrare che l'output rispetti le **caratteristiche previste**
2. Dimostrare per induzione che ogni istanza dell'output (ossia il suo contenuto a seguito di ogni iterazione) sia **contenuto all'interno di una qualsiasi soluzione ottimale**:
   - Supponendo che l'istanza $\texttt{Sol}_k$ sia contenuta in una soluzione ottimale $\texttt{Sol}^*$, dimostrare che anche $\texttt{Sol}_{k+1}$ sia contenuto in una soluzione ottimale, la quale solitamente coincide con $(\texttt{Sol}^* - x) \cup y$, dove $x \in \texttt{Sol}^*$ e $y \notin \texttt{Sol}^*$
3. Dimostrare che l'output finale generato **coincida esattamente con la soluzione ottimale** all'interno di cui è contenuto

2.1.1 Esempi di dimostrazione di algoritmi greedy

Problem 2. Massimi intervalli disgiunti

Dato un insieme di n intervalli nella forma $[a_i, b_i]$, dare un algoritmo che trovi un sottoinsieme di cardinalità massima di intervalli disgiunti in $O(n \log n)$

Input:
I: insieme degli n intervalli,
Output:
Sottoinsieme di cardinalità massima di intervalli disgiunti

```
Function findMaxSubsetOfIntervals(I):
    I.sortByRightBound()      //Estremi destri in ordine crescente;
    Sol = ∅;
    lastRightBound = −∞;
    for [a_i, b_i] ∈ I do
        if a_i > lastRightBound then
            Sol.add([a_i, b_i]);
            lastRightBound = b_i;
        end
    end
    return Sol;
end
```

Dimostrazione correttezza algoritmo:

- Siano $\texttt{Sol}_0, \dots, \texttt{Sol}_n$ le istanze dell'insieme Sol ad ogni iterazione del ciclo for.
- Poiché $\texttt{Sol}_0 = \varnothing$, tale insieme è contenuto in una qualsiasi soluzione ottimale. Supponiamo quindi per ipotesi induttiva che esista una soluzione ottimale $\texttt{Sol}^*$ tale che $\texttt{Sol}_k \subseteq \texttt{Sol}^*$.
- Consideriamo quindi $\texttt{Sol}_{k+1}$. Poiché l'intervallo $int := [a_{k+1}, b_{k+1}] \in I$ considerato all'interno del for può essere aggiunto o no all'insieme Sol, si ha che

$$\texttt{Sol}_{k+1} = \begin{cases} \texttt{Sol}_k & \text{se } \exists [a_i, b_i] \in \texttt{Sol}_k \mid [a_{k+1}, b_{k+1}] \cap [a_i, b_i] \neq \varnothing \\ \texttt{Sol}_k \cup \{[a_{k+1}, b_{k+1}]\} & \text{se } \forall [a_i, b_i] \in \texttt{Sol}_k \mid [a_{k+1}, b_{k+1}] \cap [a_i, b_i] = \varnothing \end{cases}$$

- Nel caso in cui $\exists [a_i, b_i] \in \texttt{Sol}_k \mid [a_{k+1}, b_{k+1}] \cap [a_i, b_i] \neq \varnothing$, per ipotesi induttiva si ha che $\texttt{Sol}_{k+1} = \texttt{Sol}_k \subseteq \texttt{Sol}^*$, dunque $\texttt{Sol}_{k+1}$ è contenuto in una soluzione ottimale
- Consideriamo quindi il caso in cui $\forall [a_i, b_i] \in \texttt{Sol}_k \mid [a_{k+1}, b_{k+1}] \cap [a_i, b_i] = \varnothing$. Supponiamo che $[a_{k+1}, b_{k+1}] \notin \texttt{Sol}^*$, implicando che

$$\exists [a_j, b_j] \in \texttt{Sol}^* \mid [a_{k+1}, b_{k+1}] \cap [a_j, b_j] \neq \varnothing$$

- Poiché $[a_{k+1}, b_{k+1}]$ è disgiunto da tutti gli intervalli in $\texttt{Sol}_k$, ne segue che anche $[a_j, b_j]$ sia disgiunto da essi, implicando quindi che $[a_j, b_j] \in \texttt{Sol}^* - \texttt{Sol}_k$ e in particolare che $k < j$, poiché altrimenti tale intervallo sarebbe stato già considerato prima dell'iterazione $k+1$. Inoltre, poiché $[a_j, b_j] \neq [a_{k+1}, b_{k+1}]$, ne segue che $k+1 < j$

- Dato che l'insieme I è stato ordinato in modo crescente in base all'estremo destro di ogni intervallo, ne segue che

$$b_1 \leq b_2 \leq \ldots \leq b_k \leq b_{k+1} \leq \ldots \leq b_n$$

  Di conseguenza, si ha che $k+1 < j \implies b_{k+1} \leq b_j$. Inoltre, poiché $[a_{k+1}, b_{k+1}] \cap [a_j, b_j]$, ne segue necessariamente che $a_j \leq b_{k+1}$, implicando quindi che $b_{k+1} \in [a_j, b_j]$

- Consideriamo quindi un qualsiasi intervallo $[a_h, b_h] \neq [a_j, b_j] \in \texttt{Sol}^* - \texttt{Sol}_k$. Per discorso analogo al caso di j, si ha che $k+1 < h$, implicando quindi che $b_{k+1} \leq b_h$

- Tuttavia, poiché $b_{k+1} \in [a_j, b_j]$ e poiché $[a_h, b_h]$ e $[a_j, b_j]$ devono essere disgiunti affinché entrambi possano essere in $\texttt{Sol}^*$, ne segue necessariamente che

$$a_j \leq b_{k+1} \leq b_j \leq a_h \leq b_h$$

  e dunque che $[a_{k+1}, b_{k+1}] \cap [a_h, b_h] = \varnothing$

- Di conseguenza, poiché $[a_{k+1}, b_{k+1}]$ è disgiunto da ogni altro intervallo in $\texttt{Sol}^*$, l'insieme $(\texttt{Sol}^* - \{[a_j, b_j]\}) \cup \{[a_{k+1}, b_{k+1}]\}$ è un insieme di intervalli disgiunti di cardinalità pari a $|\texttt{Sol}^*|$ e quindi una soluzione ottimale contenente $\texttt{Sol}_{k+1}$.

  Dunque, concludiamo che $\forall i \in [1, n]$ esiste una soluzione ottimale contenente $\texttt{Sol}_i$

- Sia quindi $\texttt{Sol}^\#$ la soluzione ottimale contenente $\texttt{Sol}_n$, implicando quindi che $\texttt{Sol}_n \subseteq \texttt{Sol}^\#$. Supponiamo quindi per assurdo che $\texttt{Sol}_n \neq \texttt{Sol}^\#$. Di conseguenza, $\exists [a_t, b_t] \in \texttt{Sol}^\# - \texttt{Sol}_n$ e in particolare che $[a_t, b_t]$ sia disgiunto da tutti gli intervalli in $\texttt{Sol}_n$, poiché $\texttt{Sol}^\#$ è un insieme di intervalli disgiunti

- Tuttavia, poiché $t \in [1, n]$, l'algoritmo avrebbe sbagliato a non inserire $[a_t, b_t]$ all'interno di $\texttt{Sol}_n$, poiché tale intervallo è stato analizzato e scartato in quanto incompatibile con gli intervalli in $\texttt{Sol}_n$, generando quindi una contraddizione

- Di conseguenza, concludiamo che $\texttt{Sol}_n = \texttt{Sol}^\#$ e dunque che l'algoritmo calcoli sempre una soluzione ottimale valida

$\square$

Dimostrazione costo algoritmo:

- L'ordinamento degli estremi destri può essere svolto in $O(n \log n)$ tramite uno qualsiasi degli algoritmi noti.

- Il ciclo for itera esattamente una volta su ogni intervallo di I, per un totale di n iterazioni al cui interno vengono svolte solo operazioni costanti, risultando in un costo pari a $O(n)$.

- Dunque, il costo finale dell'algoritmo è pari a $O(n \log n)$

$\square$

Problem 3. Minimi punti ricoprenti insieme di intervalli

Dato un insieme di n intervalli nella forma $[a_i, b_i]$, dare un algoritmo che trovi un sottoinsieme di punti $x_1, \ldots, x_h$ tale che $\forall i \in [1, n]$ si abbia che $[a_i, b_i] \cap \{x_1, \ldots, x_h\} \neq \varnothing$ in $O(n \log n)$

Input:
I: insieme degli n intervalli,
Output:
Sottoinsieme di cardinalità minima di punti intersecanti

```
Function findDisjointPoints(I):
    I.sortByRightBound()      //Estremi destri in ordine crescente;
    Sol = ∅;
    for [a_i, b_i] ∈ I do
        if Sol ∩ [a_i, b_i] = ∅ then
            Sol.add(b_i);
        end
    end
    return Sol;
end
```

Dimostrazione correttezza algoritmo:

- Siano $\texttt{Sol}_0, \ldots, \texttt{Sol}_n$ le istanze dell'insieme Sol ad ogni iterazione del ciclo for
- Poiché $\texttt{Sol}_0 = \varnothing$, tale insieme è contenuto all'interno di una qualsiasi soluzione ottimale. Supponiamo quindi per ipotesi induttiva che esista una soluzione ottimale $\texttt{Sol}^*$ tale che $\texttt{Sol}_k \subseteq \texttt{Sol}^*$.
- Consideriamo quindi l'istanza $\texttt{Sol}_{k+1}$. Se $\texttt{Sol}_{k+1} = \texttt{Sol}_k$, allora $\texttt{Sol}_{k+1} \subseteq \texttt{Sol}^*$, dunque $\texttt{Sol}_{k+1}$ è contenuto in una soluzione ottimale.
- Consideriamo quindi il caso in cui $\texttt{Sol}_{k+1} = \texttt{Sol}_k \cup \{b_{k+1}\}$ dove b_{k+1} è l'estremo destro dell'intervallo $[a_{k+1}, b_{k+1}]$ considerato alla $(k+1)$-esima iterazione, implicando dunque che $b_{k+1} \cap [a_i, b_i] = \varnothing, \forall i \in [1, k]$.
- Se $b_{k+1} \notin \texttt{Sol}^*$, ne segue che esista un punto interno a $\texttt{Sol}^* - \texttt{Sol}_k$ già coprente l'intervallo $[a_{k+1}, b_{k+1}]$, ossia che

$$\exists x \in \texttt{Sol}^* - \texttt{Sol}_k \mid \{x\} \cap [a_{k+1}, b_{k+1}] \neq \varnothing \implies x \in [a_{k+1}, b_{k+1}]$$

- Dato un qualsiasi intervallo $[a_j, b_j] \in \texttt{I}$ tale che $x \in [a_j, b_j]$, si hanno due casi:
  - Se $j \leq k$, allora $[a_j, b_j]$ è stato già considerato dall'algoritmo, implicando che $\exists y \in \texttt{Sol}_k \mid y \in [a_j, b_j]$ e dunque che tale intervallo sia già coperto
  - Se $j > k + 1$, allora $b_j \geq b_{k+1}$, poiché l'insieme degli intervalli è stato ordinato per estremo destro crescente, implicando dunque che

    $$x \in [a_j, b_j] \cap [a_{k+1}, b_{k+1}] \wedge b_j \geq b_{k+1} \implies a_j \leq x \leq b_{k+1} \leq b_j \implies b_{k+1} \in [a_j, b_j]$$

    dunque anche b_{k+1} è in grado di coprire un qualsiasi intervallo coperto da x

- Di conseguenza, in entrambi i casi otteniamo che $(\texttt{Sol}^* \cup \{b_{k+1}\}) - \{x\}$ sia una soluzione ottimale contenente $\texttt{Sol}_{k+1}$
- Sia quindi $\texttt{Sol}^{\#}$ la soluzione ottimale contenente $\texttt{Sol}_n$, implicando quindi che $\texttt{Sol}_n \subseteq \texttt{Sol}^{\#}$.

  Supponiamo quindi per assurdo che $\texttt{Sol}_n \neq \texttt{Sol}^{\#}$. Di conseguenza, $\exists z \in \texttt{Sol}^{\#} - \texttt{Sol}_n$, dove in particolare si ha che $\exists [a_t, b_t] \in \texttt{I} \mid \{z\} \cap [a_t, b_t] \neq \varnothing \implies z \in [a_t, b_t]$ in quanto $\texttt{Sol}^{\#}$ è una soluzione ottimale.
- Tuttavia, poiché $t \in [1, n]$, l'algoritmo avrebbe sbagliato a non inserire b_t all'interno di $\texttt{Sol}_n$ poiché $z \notin \texttt{Sol}_n$ implica che in $\texttt{Sol}_n$ l'intervallo $[a_t, b_t]$ non sia coperto da alcun punto.
- Di conseguenza, concludiamo che l'unica possibilità sia che $\texttt{Sol}_n = \texttt{Sol}^{\#}$ e dunque che l'algoritmo calcoli sempre una soluzione ottimale valida

□

Dimostrazione costo algoritmo:

- L'ordinamento degli estremi destri può essere svolto in $O(n \log n)$ tramite uno qualsiasi degli algoritmi noti.
- Il ciclo for itera esattamente una volta su ogni intervallo di I, per un totale di n iterazioni al cui interno vengono svolte solo operazioni costanti, risultando in un costo pari a $O(n)$.
- Dunque, il costo finale dell'algoritmo è pari a $O(n \log n)$

□

2.2 Minimum Spanning Tree (MST)

Definition 35. Albero di copertura

Sia G un grafo non diretto connesso. Definiamo il sottografo $T \subseteq G$ come **albero di copertura (Spanning Tree - ST)** di G se T è **aciclico** e $V(H) = V(G)$

Observation 15

Dato un grafo G non diretto connesso, possono esistere più alberi di copertura di G

Definition 36. Minimun Spanning Tree (MST)

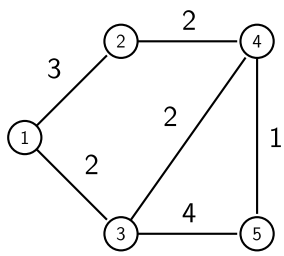
Sia G un grafo non diretto connesso pesato. Dato l'albero di copertura $T \subseteq G$, definiamo T come **albero di copertura minimale (Minimum Spanning Tree - MST)** se la somma di tutti i pesi degli archi di $E(T)$ è il minimo possibile tra tutti gli alberi di copertura esistenti di G

Observation 16

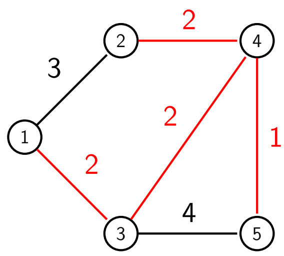
Dato un grafo G non diretto connesso, può esistere più di un MST di G

Esempio:

- Consideriamo il seguente grafo



- Uno dei possibili MST di tale grafo (in tal caso l'unico possibile) corrisponde a



Observation 17

Sia G un grafo non diretto connesso pesato. Dato il sottografo H tale che $V(H) = V(G)$ e tale che la **somma di tutti i pesi degli archi in** $E(H)$ sia il minimo possibile, allora H è necessariamente un **albero**, implicando dunque che H sia un MST

Dimostrazione:

- Per dimostrazione precedente, sappiamo che

$$H \text{ grafo non diretto connesso aciclico} \iff H \text{ è un albero}$$

- Supponiamo per assurdo che H non sia un albero, implicando dunque che esista un ciclo in H tale che $x \to y \to x$, dove $x, y \in V(G)$.
- Siano quindi P_1 e P_2 rispettivamente i cammini distinti tali che $x \to y$ e $y \to x$. In tal caso, si verificano tre casi possibili:
  - Se $w_p(P_1) < w_p(P_2)$, allora il cammino P_2 risulta essere superfluo affinché H sia un sottografo di copertura minimo di G
  - Se $w_p(P_1) = w_p(P_2)$, allora uno dei due cammini risulta essere superfluo affinché H sia un sottografo di copertura minimo di G
  - Se $w_p(P_1) > w_p(P_2)$, allora il cammino P_1 risulta essere superfluo affinché H sia un sottografo di copertura minimo di G

- Di conseguenza, poiché in ognuno dei tre casi può essere rimosso un cammino e di conseguenza alcuni archi di esso, verrebbe contraddetta l'ipotesi per cui il sottografo H ricoprente G sia di peso minimo. Dunque, H deve necessariamente essere privo di cicli, implicando quindi che H sia un albero e dunque un MST di G

$\square$

2.2.1 Algoritmo di Kruskal

Algorithm 21. Algoritmo di Kruskal

Sia G un grafo non diretto connesso pesato e rappresentato tramite liste di adiacenza. Il seguente algoritmo restituisce un MST di G

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

Algorithm 21: Algoritmo di Kruskal

Input:
G: grafo non diretto a liste di adiacenza,
Output:
MST di G

```
Function kruskal(G):
    E(G).sortByWeight()        //Pesi degli archi in ordine crescente;
    Sol = ∅;
    for e ∈ E(G) do
        if findCycle(Sol ∪ e) == ∅ then
            Sol.add(e)        //Aggiungi arco solo se non crea cicli;
        end
    end
    return Sol;
end
```

Dimostrazione correttezza algoritmo:

- Siano $\texttt{Sol}_0, \dots, \texttt{Sol}_m$ le istanze dell'insieme `Sol` ad ogni iterazione del ciclo for
- Per via del modo in vengono scelti gli archi da aggiungere all'insieme `Sol`, ne segue che il sottografo $\texttt{Sol}_m \subseteq G$ sia aciclico
- Inoltre, poiché G è connesso, per ogni vertice $v \in V(G)$ esisterà sempre almeno un arco $e_v \in E(G)$ selezionabile dal ciclo, implicando quindi che $V(\texttt{Sol}_m) = V(G)$
- Supponiamo quindi per assurdo che $V(\texttt{Sol}_m) \neq V(G)$ e che $\texttt{Sol}_m$ non sia connesso. Di conseguenza, esiste un componente connesso $C \subseteq \texttt{Sol}_m$ tale che $V(C) \neq V(G)$.
- Poiché G è connesso, dunque fortemente connesso essendo indiretto, allora $\exists (u, v) \in E(G) \mid u \in C, v \in G - C$. Di conseguenza, l'algoritmo avrebbe sbagliato a non

aggiungere tale arco alla soluzione, poiché esso non creerebbe alcun ciclo all'interno di $\mathtt{Sol}_m$.

- Dunque, l'unica possibilità è che $V(\mathtt{Sol}_m) = V(G)$ e che $\mathtt{Sol}_m$ sia connesso, implicando quindi che esso sia un albero di copertura di G
- Poiché $\mathtt{Sol}_0 = \varnothing$, tale insieme è contenuto in una qualsiasi soluzione ottimale (ossia avente peso minimo). Supponiamo quindi per ipotesi induttiva che esista una soluzione ottimale $\mathtt{Sol}^*$ tale che $\mathtt{Sol}_k \subseteq \mathtt{Sol}^*$.
- Consideriamo quindi $\mathtt{Sol}_{k+1}$ e l'arco $e_{k+1} := (x, y) \in E(G)$ considerato alla (k+1)-esima iterazione. Se $\mathtt{Sol}_{k+1} = \mathtt{Sol}_k$ allora $\mathtt{Sol}_{k+1} = \mathtt{Sol}_k \subseteq \mathtt{Sol}^*$, dunque $\mathtt{Sol}_{k+1}$ è contenuto in una soluzione ottimale.
- Consideriamo quindi il caso in cui $\mathtt{Sol}_{k+1} = \mathtt{Sol}_k \cup e_{k+1}$, dove $e_{k+1} \notin E(\mathtt{Sol}^*)$. Poiché la soluzione ottimale $\mathtt{Sol}^*$ è un MST, ne segue che esista un cammino non diretto P in $\mathtt{Sol}^*$ tale che $x \to y$, implicando dunque che esista un ciclo $F := P \cup e_{k+1}$ in $\mathtt{Sol}^* \cup e_{k+1}$.
- Di conseguenza, poiché $y \in V(\mathtt{Sol}^*) = V(G)$ in quanto $\mathtt{Sol}^*$ è un albero di copertura, ne segue che $\exists e_j := (z, y) \in E(F - \mathtt{Sol}_{k+1})$ dove $x \to z$ tale che $\mathtt{Sol}' := (\mathtt{Sol}^* \cup \{e_{k+1}\}) - \{e_j\}$ sia un albero di copertura
- Poiché $e_j \in E(F - \mathtt{Sol}_{k+1}) \implies e_j \notin E(\mathtt{Sol}_{k+1})$, ne segue necessariamente che $k + 1 < j$. Dunque, essendo gli archi ordinati per peso crescente, ne segue che

  $$w(e_{k+1}) \leq w(e_j) \implies \sum_{e \in E(\mathtt{Sol}')} w(e) \leq \sum_{f \in E(\mathtt{Sol}^*)} w(f)$$

  implicando quindi che anche $\mathtt{Sol}' := (\mathtt{Sol}^* \cup \{e_{k+1}\}) - \{e_j\}$ abbia peso minimo e dunque che sia una soluzione ottimale contenente $\mathtt{Sol}_{k+1}$
- Sia quindi $\mathtt{Sol}^\#$ la soluzione ottimale tale che $\mathtt{Sol}_m \subseteq \mathtt{Sol}^\#$. Poiché $\mathtt{Sol}_m$ e $\mathtt{Sol}^\#$ sono entrambi alberi di copertura, ne segue necessariamente che $\mathtt{Sol}_m = \mathtt{Sol}^\#$

$\square$

Dimostrazione costo algoritmo:

- L'ordinamento degli archi può essere effettuato in $O(m \log m)$ utilizzando uno degli algoritmi noti
- Il ciclo itera su ogni arco del grafo per un totale di m iterazioni, effettuando al suo interno la ricerca di un ciclo utilizzando l'algoritmo 7 `findCycle` avente costo pari a $O(n)$ (poiché G è non diretto), risultando in un costo pari a $O(mn)$
- Dunque, il costo finale dell'algoritmo è pari a $O(m \log m + mn)$

$\square$