Progettazione di Algoritmi

Marco Casu

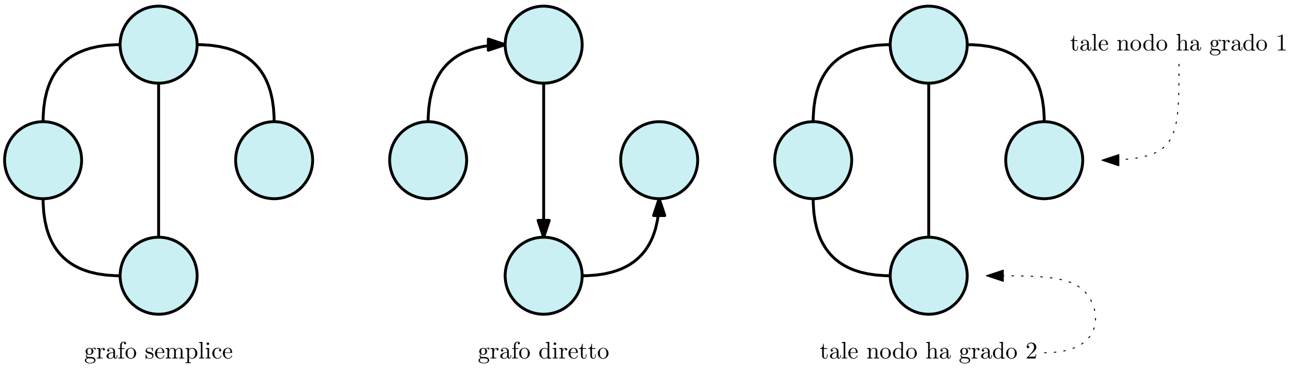
Contents

1 Grafi

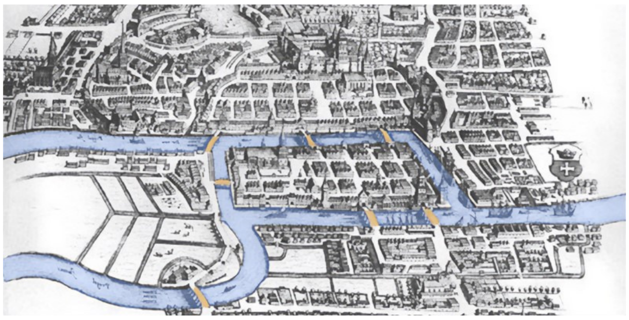
1.1 Introduzione e Definizioni

Un grafo, è una coppia (V, E), dove V è un insieme di *nodi o vertici*, ed E un insieme di archi che collegano i nodi. Un grafo è detto **semplice** se, per ogni coppia di nodi, essi sono collegati da al massimo un arco, e non esistono dei cicli su un singolo nodo. Nel corso ci occuperemo di *visitare* i grafi in profondità ed in ampiezza (concetti che verranno ripresi più in avanti).

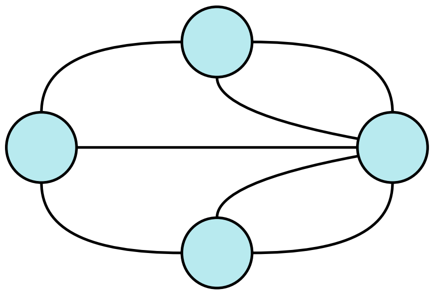
Un grafo, può vedere i suoi archi *orientati*, in questo caso si dice che il grafo è **diretto**. Due nodi sono **adiacenti** se collegati da un arco, ed il **grado** di un nodo non è altro che il numero di nodi adiacenti ad esso.



Esiste un problema classico dal 1700, noto come *problema dei ponti di Königsberg*, si consideri la seguente città posta nei pressi di un fiume che la divide in diversi settori, collegati da appositi ponti, rappresentata con il seguente grafo :



si rappresenta :

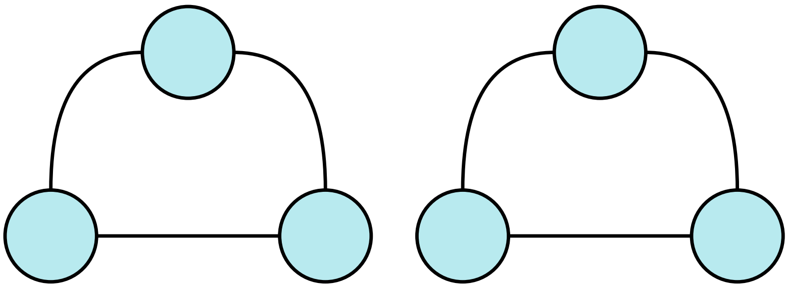


Ci si chiede se è possibile passeggiare per la città, visitando tutti i settori, senza passare per due volte sullo stesso ponte. Consideriamo il modello del grafo, una passeggiata su un grafo non è altro che una sequenza ordinata di vertici ed archi che si alternano, come : $v_0, e_1, v_1 \dots, e_k, v_k$. Esiste una passeggiata su questo grafo, ossia una sequenza che non vede ripetizioni degli archi?

Osservazione : Per visitare un nodo è necessario passare per due archi, uno entrante ed uno uscente. Se entriamo in un nodo di grado 3, resterà un arco non visitato, per visitarlo sarà necessario entravi nuovamente da tale arco, per poi uscire da un altro precedentemente già visitato (questo ovviamente se non si comincia la passeggiata dal nodo in questione).

Ci rende chiaro il seguente fatto : Se il grado di un nodo x è dispari, a meno che la passeggiata non inizi o finisca su x, uno dei suoi archi verrà attraversato più di una volta. *Eulero* studiò questo problema, si dice infatti che la passeggiata su un grafo è **euleriana** se non si passa 2 volte sulle stesso arco.

Si consideri però il seguente grafo :



Pur vedendo ognuno dei suoi nodi avere grado pari, ossia 2, tale grafo non permette alcuna passeggiata aleatoria, in quanto non è *connesso*.

Un grafo si dice **connesso** se, per ogni coppia di vertici, essi sono collegati da una passeggiata, ossia è possibile raggiungere un vertice partendo da un altro. Le precedenti osservazioni ci portano al seguente risultato.

Teorema (Eulero) : Un grafo ha una passeggiata euleriana se e solo se è connesso, ed esistono al massimo 2 vertici di grado dispari.

Il fatto che sono concessi 2 vertici di grado dispari, è dato dal fatto che essi saranno l'inizio e la fine della passeggiata.

1.2 Rappresentazione Fisica

Che struttura dati possiamo utilizzare per rappresentare un grafo? Vediamo due alternative :

- **Matrice di Adiacenza** - Utilizziamo una matrice $n \times n$, dove n è il numero di nodi del grafo. Nella posizione i, j ci sarà 1 se il vertice v_i è adiacente al vertice v_j, altrimenti 0. Il costo di "*check*" per l'adiacenza di due vertici è costante, basta consultare un entrata della matrice, nonostante ciò, lo spazio che occupa tale rappresentazione è $O(n^2)$.

- **Liste di Adiacenza** - Ad ogni vertice del grafo è associata una lista, contenente tutti i suoi vertici adiacenti, per controllare se due vertici sono adiacenti, è necessario fare una ricerca lineare su tale lista, ed ha costo $O(\deg(v))$, dove v è il vertice sulla quale si sta effettuando la ricerca, ed è ovviamente limitato da $n-1$ (numero di vertici).

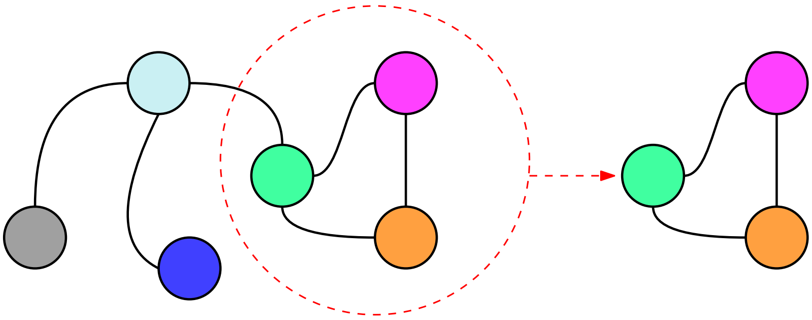
  Le dimensioni della struttura dati sono $O\big(n + \sum_{v \in V(G)} \deg(v)\big)$.

Nel caso in cui un grafo dovesse vedere ogni vertice adiacente a tutti gli altri, la ricerca costerebbe $O(n)$ e le dimensioni sarebbero $O(n^2)$, ciò differisce però dal caso reale, la rappresentazione con liste di adiacenza risulta un buon compromesso fra costo computazionale e dimensioni. Sarà usuale denotare m il numero di archi e n il numero di vertici. Le liste di adiacenza occupano quindi spazio $O(n+m)$, si osservi inoltre la seguente identià :

$$\sum_{v \in V(G)} \deg(v) = 2 \cdot m \text{ dove } m := |E|$$

1.3 Ricerca di un Ciclo

Definizione : Un *ciclo* in un grafo, non p altro che un *sottografo connesso* dove ogni vertice è di grado 2. Identifica un "cammino circolare", e la ricerca dei cicli nei grafi è un problema molto noto.

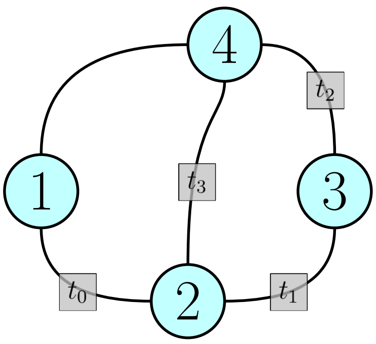


Consideriamo adesso un problema, vogliamo definire un algoritmo che, dato in input un grafo $G = (V, E)$, dove ogni vertice ha grado maggiore o uguale a 2, restituisca in output un qualsiasi ciclo presente nel grafo, mantenendo un costo computazionele $O(n + m) = O(|V| + |E|)$.

Si consideri la seguente *idea* informale di soluzione :

Ogni vertice ha almeno 2 nodi adiacenti, è quindi sempre possibile entrare in un vertice ed uscirne da un arco diverso da quello dalla quale si è entrati. Si parte da un qualsiasi vertice nel grafo, e si procede selezionando uno qualsiasi dei due nodi adiacenti successivi, almeno uno dei due non sarà quello dalla quale si è entrati, procederemo in questa maniera camminando in maniera casuale sul grafo, finchè non troveremo un nodo che è stato già visitato in precedenza, ciò indica che si è eseguito un cammino ciclico.

Utilizzeremo un vettore con lo scopo di salvare i nodi visitati, il ciclo sarà rappresentato dai nodi presenti nel vettore, partendo dall'ultimo elemento, continuando a ritroso fino a trovare il nodo identico all'ultimo. Si consideri il seguente esempio in cui gli archi sono contrassegnati dall'iterazione dell'algoritmo nella quale sono stati attraversati :



Ha prodotto l'array $V = [1, \underbrace{2, 3, 4, 2}_{\text{è il ciclo}}]$

Una volta completato la ricerca del ciclo, elimineremo dal vettore tutti gli elementi a partire dal primo fino all'elemento antecedente a quello identico all'elemento finale.

Pseudocodice

`Input` : Un grafo $G = (V, E)$.
`Output` : I nodi di un sottografo di G che è un ciclo.

```
CercaCiclo(graph G){
    x = V[random] // Un vertice a caso
    W=[x] // Inizializzo il vettore output
    current = V
    y=adiacente di x // Un adiacente a caso
```

```
    next=y
    while(next∉ W){
        W.append(next)
        current=next
        if (1° adiacente di current≠W[W.lenght-2]){ // Il penultimo
            next = 1° adiacente di current
        }else{next = 2° adiacente di current
    }
    while(W[0]≠next){
        W.remove(W[0]) // Rimuove il primo elemento
    }
    return W
}
```

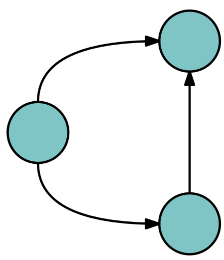
Qual'è la complessita di tale algoritmo? Entrambi i cicli `while` eseguono $O(n)$ iterazioni, il fatto è che, nel primo ciclo while, il controllo `next∉W` deve scorrere comunque tutto il vettore, rendendo il costo dell'algoritmo $O(n^2)$, non rispettando le specifiche iniziali, ossia $O(n+m)$.

1.4 Cammini sui Grafi

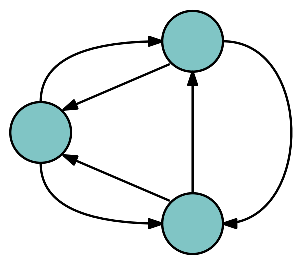
Un **cammino**, non è altro che una passeggiata su un grafo in cui non si passa mai più di una volta sullo stesso vertice, ossia una passeggiata senzza ripetizioni di vertici o archi.

Osservazione : Siano x ed y due nodi di un grafo, se esiste una passeggiata da x ad y, allore esiste anche un cammino.

Nei grafi diretti vale la stessa regola, con ovviamente il vincola che bisogna rispettare l'orientazione degli archi. Un grafo diretto si dice **fortemente connesso** se, per ogni coppia di vertici x, y, esiste un cammino da x ad y e viceversa.



non è fortemente connesso



è fortemente connesso

Un noto problema è il seguente, dato un grafo G e due veritici x, y, esiste un cammino da x ad y? In generale, il carico di lavoro per controllare ciò, equivale al carico di lavoro necessario per controllare tutti i nodi che possono essere "raggiunti" partendo da x.

Prendo quindi un vertici x e trovo tutti i vertici y per i quali esiste un cammino fra essi, per fare ciò, occorre **visitare** il grafo, e può essere fatto in due modi differenti.

1.4.1 Depth-First Search

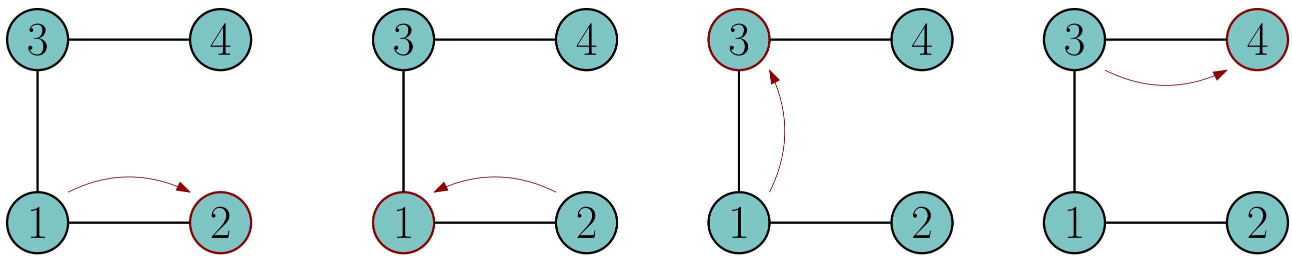
Abbreviato **DFS**, tale algoritmo rappresenta la visita su un grafo in *profondità*. Partendo da un qualsiasi vertice x, inizio a visitare randomicamente uno dei vertici adiacenti, per poi proseguire da esso. Se ad un certo punto non vi sono nuovi vertici da visitare, si esegue il cosiddetto *back tracking*, controllando i nodi a ritroso e cercando dei nuovi vertici. Risulta quindi naturale l'uso di uno *stack* per poter implementare tale ricerca. L'algoritmo alla fine visiterà ogni nodo per la quale esiste un cammino dal nodo iniziale.

Pseudocodice

`Input` : Un grafo $G = (V, E)$, ed un vertice x.
`Output` : L'insieme dei vertici visitati partendo da x.

```
DFS(graph G, vert x){
    S : stack = {x}
    Vis :  set = [x]     // l'insieme che conterrà l'output
    while(S≠ ∅){
        y=S.top()
        if(∃z adiacente ad y∧z∉Vis){
            Vis.add(z)
            S.push(z)
        }
        else{
            S.pop()
        }
    }
    return Vis
}
```

Esempio di applicazione (il nodo di partenza è il nodo 1) :



primo passo
Vis = [1,2]

secondo passo
Vis = [1,2]

terzo passo
Vis = [1,2,3]

quarto passo
Vis = [1,2,3,4]

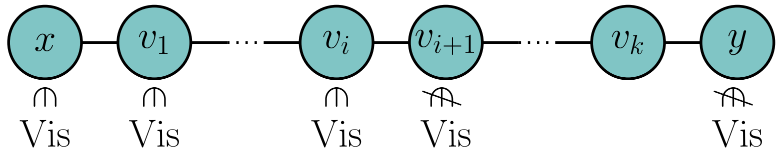
L'output dell'algoritmo sarà proprio l'insieme `Vis`, contenente tutti i nodi raggiungibili dal vettore input, bisogna dimostrare che l'algoritmo sia corretto, mostrando che ogni vertice raggiungibile da x è in `Vis`.

Dimostrazione : Supponiamo per assurdo che vi sia un vertice y tale che, esiste un cammino da x ad y e che y non sia presente in Vis.

$$\exists y | x \to y \land y \notin \text{Vis}$$

Essendo x il vertice di partenza, esso sicuramente si troverà in Vis, per costruzione dell'algoritmo. Questo vuol dire che esiste un vertice nel cammino, per la quale vale la seguente proprietà :

$$\text{Siano } v_1 \dots v_k \text{ vertici nel cammino } x \to y,\ \exists v_i | v_i \in \text{Vis} \land v_{i+1} \notin \text{Vis}$$



Essendo v_i in Vis, vuol dire che ad un certo punto è stato nel top dello stack, ma v_{i+1} è adiacente a v_i, quindi da quest'ultimo l'algoritmo avrà selezionato ad un certo punto v_{i+1}, per poi proseguire da esso, per costruzione, sarà inserito in Vis, ma ciò è in contraddizione con l'ipotesi iniziale che y non è in Vis. ■

Questo algoritmo presenta un problema cruciale, non è efficiente, infatti risulta particolarmente pesante il controllo `if(∃z adiacente ad y∧z≠Vis)`, che ha costo computazionale $O(\deg(y)) + O(n)$. L'algoritmo va migliorato, al posto di un set, è possibile utilizzare un array nella seguente maniera : sarà composto da $n := |V|$ elementi inizializzato con tutti 0, si avrà che $array[i] = 1 \iff i$ fa parte dell'output.

Pseudocodice

```
DFS2(graph G, vert x){
    S : stack = {x}
    Vis : int[n] = [0,0...0]        // L'array in questione
    Vis[x]=1
    while(S≠ ∅){
        y=S.top()
        if(Vis[y.adiacenti[0]]==0){        // Trova un adiacenta non ancora controllato
            z=y.adiacenti[0]
            Vis[z]=1
            S.push(z)
            y.adiacenti.remove(0)
        }
        else{
            y.adiacenti.remove(0)
        }
        if(y.adiacenti== ∅){S.pop()}
    }
    return Vis
}
```

Si è nell'ipotesi in cui il grafo è implementato con le liste di adiacenza, infatti si noti come ogni vertice presenta il campo `adiacenti`. Per rendere più efficiente il tutto senza dover controllare ogni volta se un nodo è stato già visitato, semplicemente si rimuove dalla lista di adiacenza, ed ogni volta se ne prende il primo di tale lista che sicuramente non è stato ancora visitato, rendendo costante tale operazione.

Qual'è ora il costo computazionale? Quante volte viene eseguito il ciclo `while`? Rispondere a ciò risulta difficile, piuttosto ci si chiede quanto lavoro devo fare nel ciclo per ogni vertice? Per ognuno di essi, si esegue un numero limitato di volte il comando `S.top()`. Nello specifico, si esegue tante volte quanto è il grado del vertice, risulta naturale che la complessità finale sia :

$$O(n) + O\big(\sum_{v\in V(G)} \deg(v)\big) = O(n + |E|) = O(n + m) \text{ costo lineare}$$

Lo stesso algoritmo, si presta in maniera piuttosto naturale ad essere implementato in maniera ricorsiva, permettendo l'omissione dell'utilizzo di uno stack.

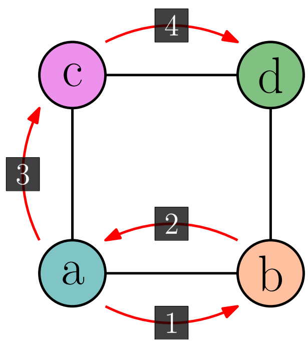
Pseudocodice

```
DFSRec(graph G, vert x,int[n] Vis){
    Vis[x]=1
    for each y∈x.adiacenti{      // per ogni adiacente di x
        if(Vis[y]==0){
            DFSRec(G,y,Vis)
        }
    }
}
```

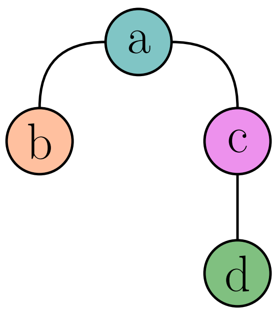
Il ciclo `for each y∈x.adiacenti` considera ogni adiacente di x una volta sola, facendo lo stesso lavoro di "cancellazione" dei vicini già controllati, la complessità rimane la medesima.

Si considera la figura seguente, rappresentante una visita *DFS* su un grafo :

L'esecuzione del DFS è rappresentata dalle frecce rosse, numerate con i rispettivi passi sequenziali.



genera l'albero



Dal nodo di partenza, si inizia a visitare diversi nodi seguendo diversi percorsi, definiamo **albero di visita**, il sottografo generato, o composto dagli archi che utilizziamo per raggiungere i nuovi vertici non ancora visitati. In generale, un albero è un grafo connesso ed aciclico. Essendo che non si ritorna mai in un nodo già visitato due volte, nell'albero di visita non si creeranno cicli (rendendolo appunto un albero).

Possiamo applicare lo stesso algoritmo ai grafi diretti, l'unica considerazione da fare, è il controllo dell'ordine di ogni arco. Consideriamo l'implementazione non ricorsiva.

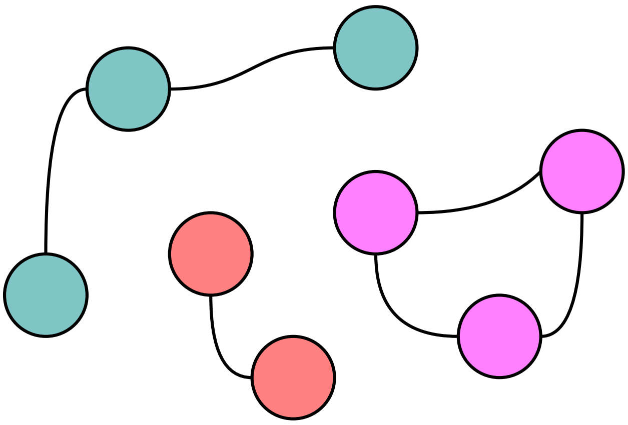
Pseudocodice

```
DFSdiretto(graph G, vert x,){
    S : stack = {x}
    Vis :  int[n] = [0,0...0]
    Vis[x]=1
    while(S≠ ∅){
        y=S.top()
        if(∃z|(y,z) ∈ E(G)∧ Vis[z]==0){      // l'arco ha la giusta orientazione
            S.push(z)
            Vis[z]=1
        }
        else{
            S.pop()
        }
    }
    return Vis
}
```

Anche questo algoritmo genera l'albero di visita, solo che avrà tutti gli archi, ordinati "verso il basso", ossia seguiranno l'orientazione che va dalla radice verso le foglie, tale albero è detto **arborescenza**.

1.4.2 Componenti di un Grafo

Se G è un grafo connesso, è ovvio che la DFS, qualsi voglia sia il vertice iniziale, restituirà sempre tutti i vertici del grafo. Se esso non dovesse essere connesso, restituirà un sottografo, precisamente il sottografo **componente** connesso che contiene il nodo input, i diversi sottografi componenti costituiscono una *partizione* del grafo originale.



Si noti come in questo grafo non connesso vi sono diversi sottografi connessi, costituiti da vertici ed archi ovviamente disgiunti (indicati con colori diversi).

Saper riconoscere le componenti di un grafo è un problema noto, che trova applicazione in svariati ambiti, ad esempio, nell'identificazione delle reti di amicizia in un social network, per capire se ci sono grandi gruppi di persone per i quali non vi è nemmeno 1 collegamento.

Il problema è il seguente, si vuole scrivere un algoritmo che identifichi tutte le componenti di un grafo, associando ad ogni vertice, un indice che ne indica la componente, dato un grafo G, e due vertici x, y, si vuole costruire un array Comp tale che :

$$\text{Comp}[x]=\text{Comp}[y] \iff x \text{ ed } y \text{ sono nella stessa componente}$$

Utilizziamo la versione ricorsiva del DFS, modificandola a dovere, sono necessarie 2 funzioni :

Pseudocodice

```
DFSRecComp(graph G, vert x,int[n] Comp, int index){     // funzione di supporto
    Comp[x]=index
    for each y∈x.adiacenti{     // per ogni adiacente di x
        if(Comp[y]==0){
            DFSRec(G,y,Comp,index)
        }
    }
}

Comp(graph G){     // funzione principale da eseguire
    Comp :  int[n] = [0,0...0]
    index = 0
    for each x∈V(G){     // per ogni vertice del grafo
        index++
        DFSRecComp(G,x,Comp,index)
    }
    return Comp
}
```

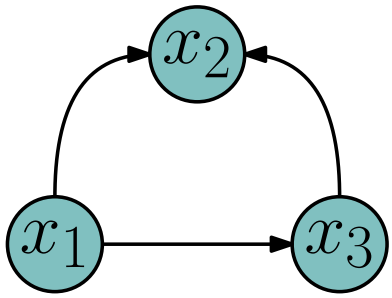
1.5 Ordinamento Topologico

Supponiamo che vi sia un progetto da completare, che viene diviso in n piccoli processi $x_1, x_2 \dots x_n$, e supponiamo che fra essi, vi siano delle dipendenze sull'ordine di completamento, ad esempio :

- Per essere completato x_1, ha bisogno che siano completati x_2, x_3
- Per essere completato x_3, ha bisogno che sia completato x_2

Dobbiamo pensare ad una programmazione dei processi che rispetti le dipendenze allo scopo di completare il progetto. Nell'esempio dato, l'ordine corretto sarebbe x_2, x_3, x_1. Utilizziamo un grafo diretto per modellizzare il problema : i processi saranno i vertici del grafo, e vi sarà un arco da x_i a x_j se x_i dipende da x_j.

In questo modello, una programmazione dei processi non è altro che un ordine dei vertici del grafo, con la proprietà che tutti i vertici siano orientati "da destra verso sinistra".



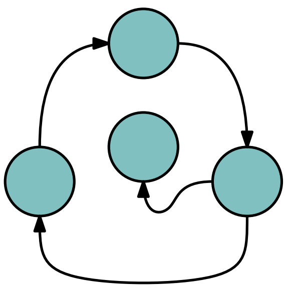
Osservazione : Se in un grafo diretto vi è un ciclo, allora il grafo non ha tutti gli archi che vanno da destra verso sinistra.

Dimostrazione : Presumiamo che esista tale ordine, allora esiste un vertice x che è l'ultimo vertice di tale ordinamento, esiste quindi un arco (y, x) per qualche y, però, nonostante sia l'ultimo, data la presenza di un ciclo, deve esistere un arco uscente (x, y), ma quindi l'ordine iniziale non è rispettato, causando una contraddizione. ∎

Se in un grafo diretto vi è un ciclo, tutto il grafo non ammette la proprietà dell'orientazione degli archi. Tale proprietà è nota con il nome di **ordine topologico**, e l'assenza di un ciclo, è condizione necessaria e sufficiente per garantirla.

Proposizione : Se ogni singolo vertice di un grafo diretto ha almeno un arco uscente, allora esiste un ciclo.

Dimostrazione : Se esiste sempre un arco uscente, è sempre possibile, partendo da un vertice x spostarsi in un suo vertice adiacente, ciò significa che è possibile "camminare" all'infinito sul grafo, il fatto è che il numero di vertici è finito, quindi prima o poi si visiterà un vertice per una seconda volta, trovandosi in un ciclo.



L'implicazione inversa non è verificata, infatti è possibile che esista un grafo in cui vi è un nodo senza archi uscenti, ed anche un ciclo.

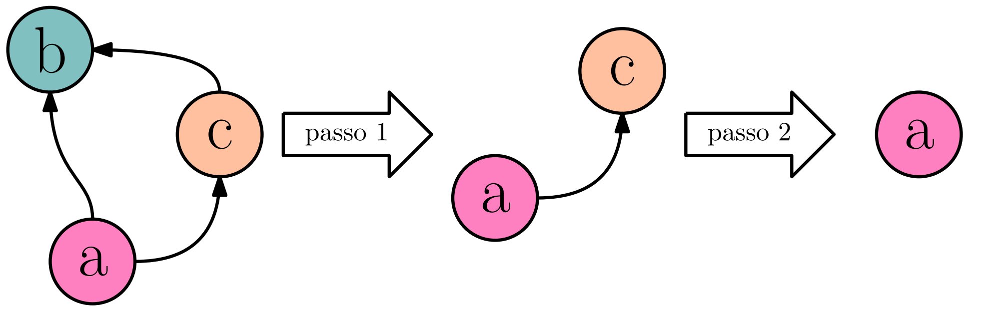
Corollario : Se non esiste alcun ciclo in un grafo, allora esiste almeno un vertice che non ha archi uscenti.

Per ottenere un cosiddetto **ordinamento topologico**, posso considerare il seguente algoritmo : Si ha un grafo diretto G, sprovvisto di cicli, si sceglie un qualsiasi vertice privo di archi uscenti, si inserisce in una lista per poi eliminarlo dal grafo (insieme a tutti i suoi archi associati), dopo ciò, si ri-esegue l'operazione, inserendo ogni volta il vertice nella prima posizione della lista.

Tale algoritmo risulta parecchio utile, si pensi all'ordinamento topologico applicato al grafo di serializzazione nell'ambito del controllo della concorrenza (trattato nel corso di Basi di Dati 1).

Pseudocodice

```
OrdinamentoTopologico(graph G){      // il grafo è diretto
    L : list    // una lista vuota, sarà l'output dell'algoritmo
    while(G≠ ∅){
        x=v∈V(G)|v.adiacentiOut=∅      // un vertice senza archi uscenti
        L.insert(x)
        G.delete(x)
    }
    return L
}
```



Il *problema* di questo algoritmo è il suo costo computazionale, di fatto è troppo dispendioso : Per controllare se un vertice non ha archi uscenti, si è in $O(n)$, inoltre il ciclo `while` controlla tutti i vertici, quindi si è nuovamente in $O(n)$.

La cancellazione di un vertice risulta dispendiosa, in quanto bisona eliminare anche tutti gli archi associati, ossia, eliminare il vertice da tutte le liste di adiacenza degli altri vertici, il numero di controlli dipende dal grado di ogni vertice, quindi costa $O(m)$. In totale, l'intero algoritmo ha una complessità $O(n \cdot (n+m))$, vorremmo riuscire ad ottenere lo stesso output in tempo lineare.

1.5.1 Contatore nel DFS e Relazioni sull'Arborescenza

Vogliamo considerare un estensione del normale DFS, consideriamo un contatore, denotato `cc`, tale contatore, verrà incrementato ogni qual volta verrà visitato per la prima volta un nuovo nodo.

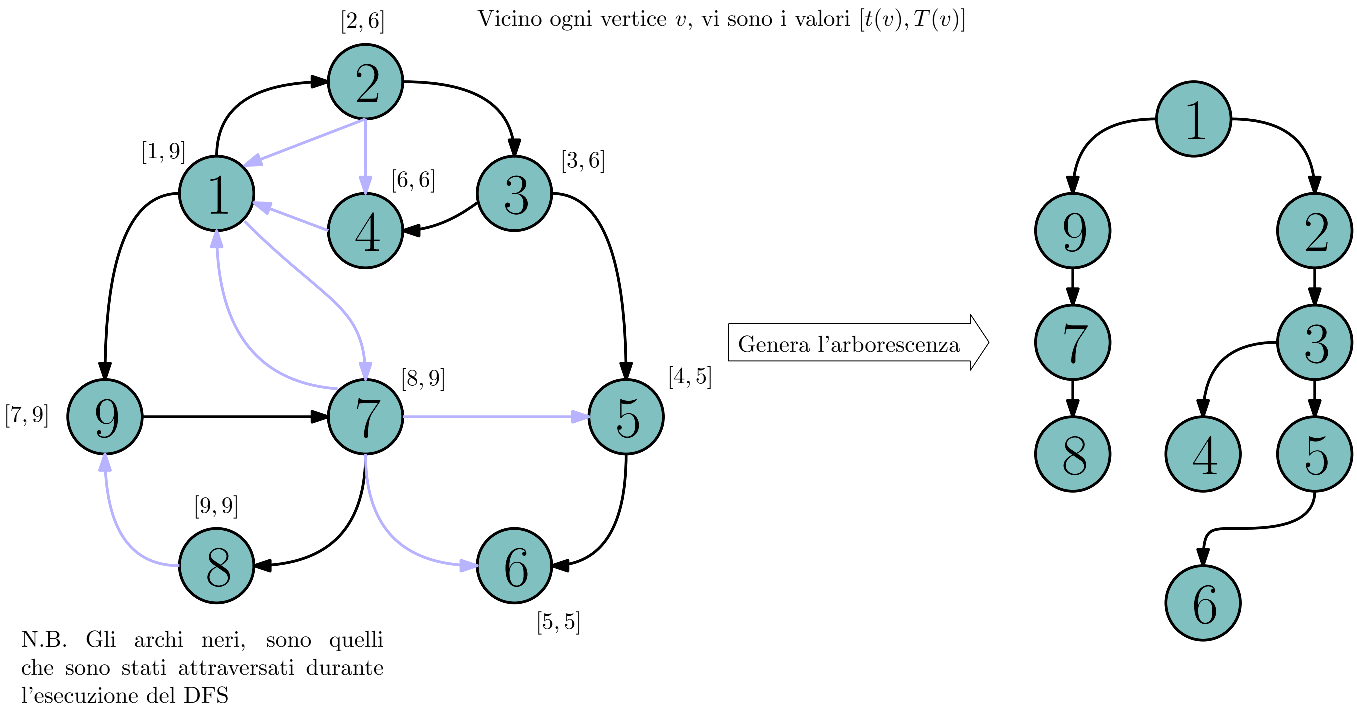
Consideriamo inoltre, due nuove funzioni $t : V(G) \to \mathbb{N}$ e $T : V(G) \to \mathbb{N}$, sia v un vertice, $t(v)$ sarà uguale al valore del contatore `cc` nel momento in cui v viene visitato per la prima volta, invece $T(v)$ sarà uguale al valore del contatore `cc` nel momento in cui v viene visitato per l'ultima volta, ossia quando esso viene rimosso dallo stack.

Osservazione :

- Per ogni coppia di vertici v, u, si ha che $t(v) \neq t(u)$
- Per ogni vertice v, si ha che $t(v) \leq T(V)$
- Sia v un vertice, se $t(v) = T(V)$, allora v, è una foglia nell'albero di visita derivante dall'applicazione del DFS.

- Sia n il numero di vertici e v_0 la radice dell'albero di visita, si ha che $t(v_0) = 1 \wedge T(v_0) = n$.

Esempio di applicazione dell'algoritmo (si parte dal vertice 1) :



Ad ogni vertice v, è associato un *intervallo* $[t(v), T(v)]$, gli intervalli di vertici diversi possono essere confrontati, e si ricade sempre in uno dei seguenti casi.

Osservazione : Siano v e u due vertici distinti del grafo, uno dei seguenti punti è sempre vero:

- *i)* $[t(v), T(v)] \subseteq [t(u), T(u)]$
- *ii)* $[t(v), T(v)] \supseteq [t(u), T(u)]$
- *iii)* $[t(v), T(v)] \cap [t(u), T(u)] = \emptyset$

Dimostrazione : Il quarto ed ultimo caso possibile, sarebbe un intersezione del tipo:

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

Basta dimostrare che questa casistica non può verificarsi. Se u è stato inserito nello stack prima di v, si avrà che $T(u) \geq t(v)$, questo implica che u era già nello stack quando v è stato inserito, ma allora è impossibile togliere u prima di v, e necessariamente $T(u) > T(v)$. ■

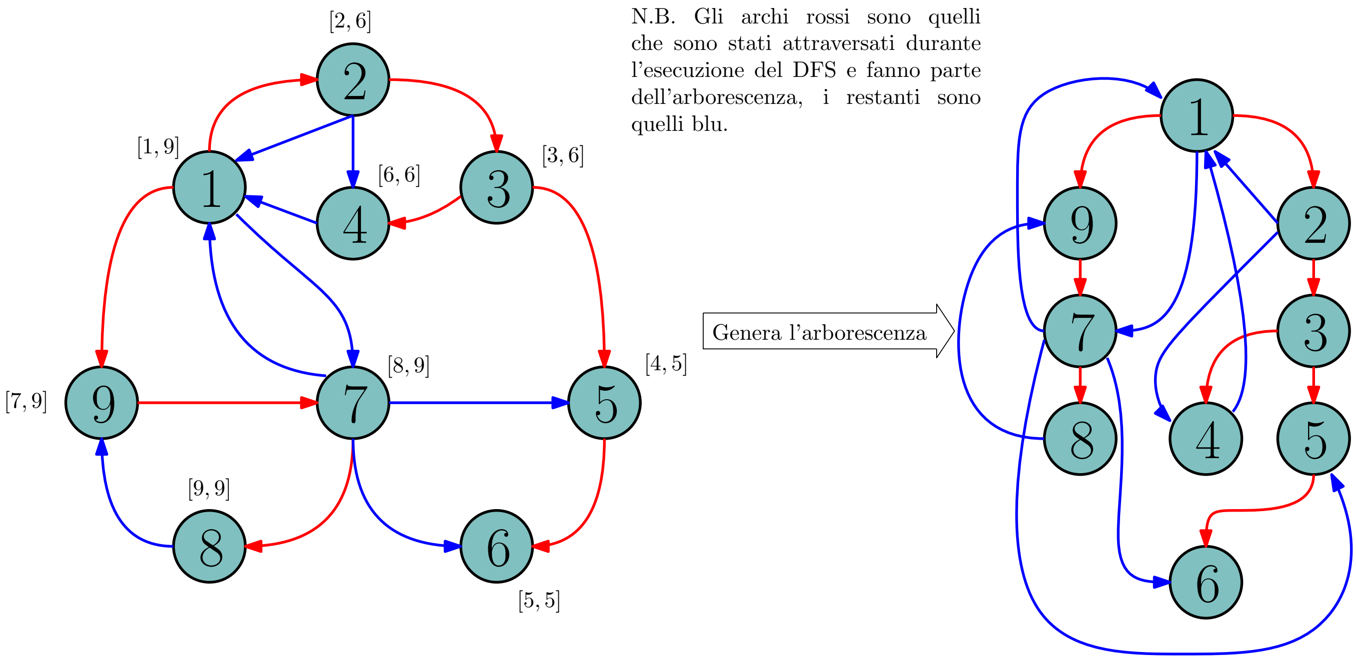
Adesso, consideriamo il grafo sulla quale è stato applicato il nuovo DFS con contatore, e consideriamo gli archi che *non appartengono* all'arborescenza, ossia gli archi che non sono stati attraversati durante il DFS (nell'immagine esplicativa precedente, quelli colorati in azzurro).

Vi è un fatto interessante, consideriamo tutti un qualsiasi arco non facente parte dell'arborescenza, esso indica due vertici (v, u), e tali vertici posseggono gli intervalli che possono essere messi in relazione, ricadendo in uno dei 3 casi prima citati.

Gli archi non facenti parte dell'arborescenza, se considerati nell'arborescenza, potranno essere di 3 tipi, o partire da un vertice ed andare verso un suo antenato, o partire da un vertice ed andare verso un suo successore, oppure attraversare due vertici di due diramazioni differenti, in effetti, riguardo la relazione di intervalli prima citata, si ha che :

- Se i due vertici dell'arco ricadono nel punto (i), allora l'arco va da un antenato ad un discendente (**arco in avanti**).
- Se i due vertici dell'arco ricadono nel punto (ii), allora l'arco va da un discendente ad un antenato (**arco all'indietro**).
- Se i due vertici dell'arco ricadono nel punto (iii), allora l'arco attraversa due diramazioni differenti (**arco di attraversamento**).

Riguardo il grafo del precedente esempio :



Si noti come l'arco che va dal vertice 8 al vertice 9, è un *arco all'indietro*, infatti gli intervalli dei due vertici ricadono nel secondo caso : $[9, 9] \supseteq [7, 9]$.

Si noti come l'arco che va dal vertice 2 al vertice 4, è un *arco in avanti*, infatti gli intervalli dei due vertici ricadono nel primo caso : $[2, 6] \subseteq [6, 6]$.

Si noti come l'arco che va dal vertice 7 al vertice 5, è un *arco di attraversamento*, infatti gli intervalli dei due vertici ricadono nel terzo caso : $[8, 9] \cap [4, 5] = \emptyset$.

Se dovessi applicare lo stesso algoritmo ai grafi non diretti, non si potrebbe definire una relazione di antenato-discendente, in quanto ogni arco è percorribile per entrambe le direzioni, quindi i casi (i) e (ii) indicherebbero la stessa situazione.

Inoltre, è impossibile che, per due nodi u, v si verifichi che $[t(v), T(v)] \cap [t(u), T(u)] = \emptyset$, quindi il caso (iii) è impossibile.

Esercizio : Si vuole dare lo pseudocodice di una modifica del DFS, che restituisca in output 3 liste, una contenente gli archi in avanti, una quelli all'indietro, ed una gli archi di attraversamento.

Pseudocodice

```
DFSconArchi(graph G, vert x,){      // il grafo è diretto
    int cc=1
    t :  int[n]      // array lungo n inizializzato a zero
    T : int[n]      // array lungo n inizializzato a zero
    t[x]=1
    T[x]=|V(G)|
    S : stack = {x}
    Vis :  int[n] = [0,0...0]
    Vis[x]=1
    while(S≠ ∅){
        y=S.top()
        if(∃z|(y, z) ∈ E(G)∧ Vis[z]==0){      // l'arco ha la giusta orientazione
            S.push(z)
            c++
            t[z]=cc
            Vis[z]=1
        }
        else{
            S.pop()
            T[z]=cc
        }
    }
    A : graph = arborescenza generata dal DFS
    A' : graph = G-A      // il complementare dell'arborescenza
    av :  list
    ind :  list
    att :  list
    for each (x,y)∈E(A'){
        switch(t[x],T[x],t[y],T[y]){
            [t(v), T(v)] ⊆ [t(u), T(u)]  :  sv.append((x,y))      // si ricade nel primo caso
            [t(v), T(v)] ⊇ [t(u), T(u)]  :  ind.append((x,y))      // si ricade nel secondo caso
            [t(v), T(v)] ∩ [t(u), T(u)] = ∅  :  att.append((x,y))      // si ricade nel terzo caso
        }
    }
    return av,ind,att
}
```

La domanda da porsi adesso è, la presenza di questi archi *in avanti*, *indietro* e di *attraversamento*, quali informazioni fornisce riguardo le proprietà del grafo?

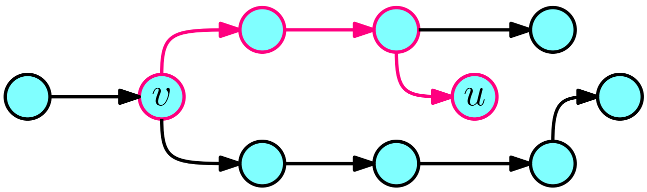
Consideriamo un grafo G non diretto e connesso, vuol dire che per ogni coppia di vertici x ed y esiste un cammino da x ad y, se dovesse esistere un'arco $(x, y) \in E(G)$, allora vi sarà un ciclo.

Proposizione : Sia G un grafo connesso non diretto, se esiste un ciclo, allora, per una *qualsiasi applicazione* del DFS, esisterà un arco all'indietro (che è identico all'arco in avanti, essendo il grafo non diretto).

Dimostrazione : Se in G c'è un ciclo, allora esisterà un arco che non sarà presente nell'albero di visita generato dal DFS (essendo un albero, non ha cicli), quindi esiste un arco esterno a tale albero che collega due nodi, ed è necessariamente un arco all'indietro. ■

$$\textit{Conclusione} : \text{DFS genera arco all'indietro} \iff G \text{ ha un ciclo}$$

Consideriamo ora il caso in cui il grafo è diretto, sia v un vertice, ed u un suo discendente nell'arborescenza generata da una qualsiasi applicazione del DFS, esiste un cammino diretto da v ad u.



Se u è un discendente di v, allora u è stato visitato la prima volta dopo di v, allora è stato rimosso dallo stack prima di v

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

Se esistesse un arco (u, v), allora sarebbe un arco all'indietro. Sappiamo che per ogni coppia di vertici u, v, se u è un discendente di v, allora esiste un cammino diretto da v ad u nell'arborescenza.

Osservazione : Se esistesse un arco all'indietro nell'arborescenza generata dal DFS, allora il grafo avrebbe un ciclo.

Proposizione 1 : Se G è un grado diretto, e tutti i suoi vertici sono raggiungibili da un vertice di partenza x, allor, una qualsiasi applicazione del DFS partendo da x genera un arco all'indietro nell'arborescenza *se e solo se* esiste un ciclo in G.

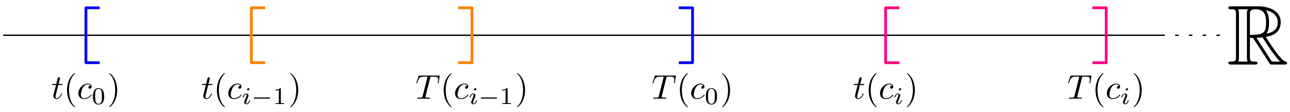
Assumendo che in G ci sia un ciclo, consideriamo i vertici che compongono il ciclo : $c_0, c_1, c_2 \ldots, c_k$, elencati in ordine di visita nel DFS, quindi c_0 è il primo vertice del ciclo visitato durante una qualsiasi applicazione del DFS.

Proposizione 2 : Tutti i vertici del ciclo (escluso c_0), verranno visitati per la prima volta *prima* che c_0 venga rimosso dallo stack.

Dimostrazione della prop. 2 : So che $\forall i \in \{1 \dots k\}, t(c_i) > t(c_0)$, assumiamo che esista un c_i fissato che non rispetti la condizione della proposizione, ossia

$$t(c_i) > T(c_0)$$

Tale c_i potrebbe non essere l'unico, sia però il vertice visitato per primo fra quelli che non rispettano la condizione, considero ora il vertice visitato appena prima di c_i, ossia c_{i-1}, so che $t(c_{i-1}) > t(c_0)$ e che $t(c_{i-1}) \leq T(c_0)$, ovviamente non può essere superiore perchè il primo vertice che viola la condizione, è il suo successivo c_i. Si verifica la condizione :



Sappiamo però che c_{i-1} viene prima di c_i nel ciclo, esiste quindi un arco (c_{i-1}, c_i), quindi è impossibile che c_{i-1} venga rimosso dallo stack prima di c_i, è quindi una contraddizione, e necessariamente la proposizione è vera. ■

Dimostrazione della prop. 1 : Data la *proposizione 2*, necessariamente l'arco del ciclo (c_k, c_0) è un arco all'indietro.

$[t(c_k), T(c_k)] \subseteq [t(c_0), T(c_0)] \implies$ arco all'indietro

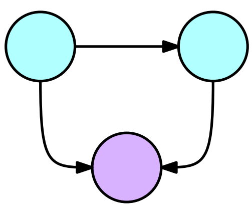
■

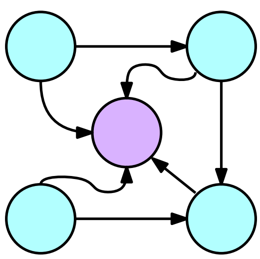
1.5.2 Pozzo Universale

Si consideri ora un vertice x di un generico grafo diretto G, che rispetti le seguenti proprietà :

- $\forall y \in V(G), \nexists (x, y) \in E(G)$
- $\forall y \in V(G), \exists (y, x) \in E(G)$

È un vertice che non ha archi uscenti, e tutti gli altri vertici del grafo hanno un arco che diretto verso di esso, tale vertice prende il nome di *pozzo universale*.





Esercizio : Si dia lo pseudocodice di un algoritmo che in $O(n)$, dove n è il numero di vertici, stabilisca se il grafo in input ha o non ha un pozzo universale, il grafo è dato sottoforma di matrice di adiacenza.

La costrizione più grande è la richiesta del costo computazionale, è chiaro che non è possibile controllare ogni vertice in maniera dettagliata, vedendo se è o non è un pozzo universale in base ai valori che assumono le entrate nella matrice.

Una possibile idea è di controllare in coppia tutti i vertici, escludendo i possibili che sicuramente non sono un pozzo universale : Si comincia controllando due vertici a caso x, y, se l'entrata della matrice $m(x, y)$ è 1, vuol dire che esiste un arco che va da x ad y, dovremmo quindi escludere x dato che ha archi uscenti, e continuare con y, altrimenti continueremo con x.

Alla fine, avremo un vertice candidato ad essere un pozzo, e controlleremo in maniera esplicita se lo è o no.

Pseudocodice

```
PozzoUniversale(m){      // l'input è la matrice di adiacenza
    candidato = 1
    n = m[1].lenght()     // n è il numero di vertici
    for(i=2;i≤n;i++){
        if(m[candidato,i]==1){
            candidato=i
        }
    }
    for(i=1;i≤n;i++){
        if(m[candidato,i]==1){
            return false     // il candidato ha un arco uscente
        }
    }
    for(i=1;i≤n;i++){
        if(m[i,candidato]==0 ∧ i≠candidato){
            return false     // un nodo non ha un arco verso il candidato
        }
    }
    return true
}
```

1.5.3 Ordine Topologico in Tempo Lineare

Tornando al DFS con il contatore, esiste ovviamente anche una versione ricorsiva, composta da due funzioni, una "globale" che inizializza il processo, ed una ricorsiva che opera.

Pseudocodice

```
DFSglobal(G,x){
    Vis :  int[n] = [0,0...,0]
    t :  int[n] = [0,0...,0]
    T : int[n] = [0,0...,0]
    c :  int = 1
    DFSrecursive(x,Vis,c,t,T)     // prima chiamata della funzione ricorsiva
    return Vis
}
```

L'algoritmo rimane in $O(n+m)$, passiamo ora alla funzione ricorsiva.

Pseudocodice

```
DFSrecursive(x,Vis,c,t,T){
    while(∃y∈V(G) | Vis[y]=0 ∧ (x,y)∈E(G)){
        Vis[y]=1
        c++
        t[y]=c
        DFSrecursive(y,Vis,c,t,T)
    }
    T[x]=c
}
```

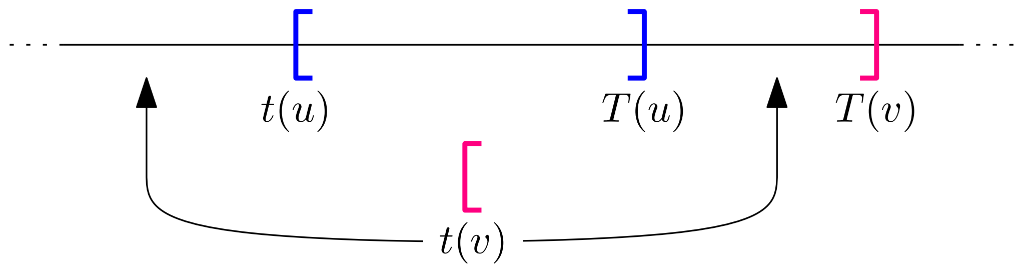
Proposizione : Per ogni arco (v,u) in un grafo diretto, si ha che $t(u) \leq T(v)$.

Dimostrazione : Se così non fosse, vorrebbe dire che $t(u) > T(v)$, significherebbe che avremmo chiuso (tolto dallo stack) v quando vi era ancora possibilità di continuare su u, quindi è impossibile che ciò accada. ■

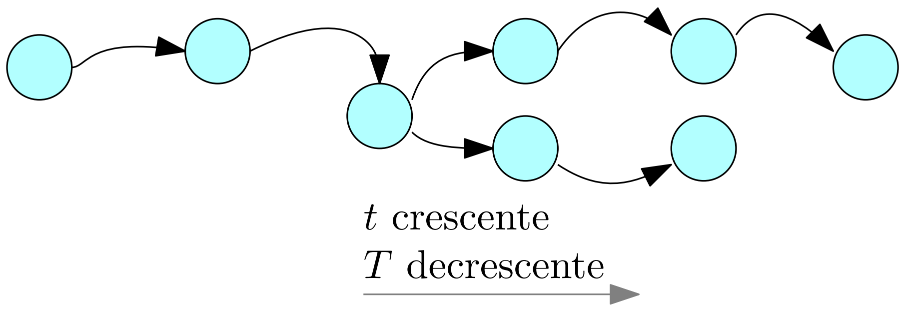
Torniamo adesso all'*ordinamento topologico*, nei capitoli precedenti, si è visto che, se il grafo è ciclico allora esistono degli archi all'indietro. In caso contrario, ci sono due restanti possibilità per ogni arco (v,u) :

- *ii)* $[t(v),T(v)] \supseteq [t(u),T(u)]$
- *iii)* $[t(v),T(v)] \cap [t(u),T(u)] = \emptyset$

Sicuramente $t(u) \leq T(u) \leq T(v)$, $t(v)$ può trovarsi in uno dei due seguenti intervalli :



Il fatto, è che $T(u) \leq T(v)$, e ciò vale per ogni arco del grafo (v,u), nel corrispettivo ordine topologico in cui tutti gli archi andranno da "sinistra verso destra", si avrà che, seguento quest'ordine, i valori di T per i vertici coinvolti saranno *decrescenti*.



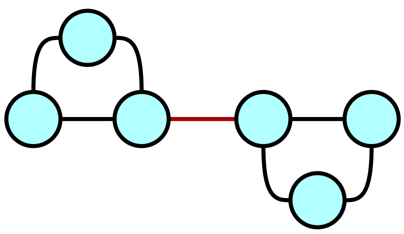
La dove si causerà una situazione di "tie break", ossia in cui i valori di T sono coincidenti per due vertici, si avrà che essi differiranno per i valori di t, che secondo l'ordine prima menzionato saranno strettamente crescenti, nel risultante ordine topologico, i vertici chiusi (tolti dallo stack) per ultimi, saranno quelli a sinistra.

Pseudocodice

```
ORDtopologico(G){      // funzione globale
    L : list     // l'output, conterrà i vertici dell'ordine topologico
    Vis :  int[n] = [0,0...,0]
    for each (v∈V(G)){
        if(Vis[v]==0){
            DFSord(G,v,Vis,L)
        }
    }
}

DFSord(G,v,Vis,L){      // funzione ricorsiva
    Vis[v]=1
    for each (w adiacente di v){
        if(Vis[w]==0){
            DFSord(G,w,Vis,L)
        }
    }
    L.insert(v,0)      // inserisci il vertice nella prima posizione della lista
}
```

1.6 Ponti sui Grafi non Diretti



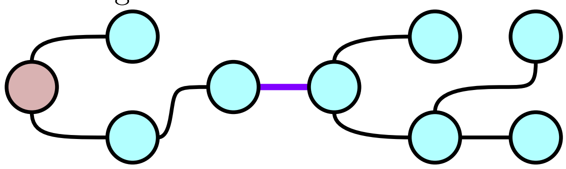
Definizione : Un *ponte* in un grafo connesso non diretto, è un arco che, se eliminato dal grafo, lo rende *non* connesso.

Come si può procedere per verificare che un arco (u, v) sia o no un ponte? Posso rimuovere l'arco, e controllare con il DFS se esiste ancora un cammino fra i due vertici coinvolti, se esiste, allora quell'arco non era un ponte. Se volessi trovare tutti i ponti di un grafo, questa operazione risulterebbe poco efficiente, e l'algoritmo avrebbe complessità $O(m \cdot (n + m))$.

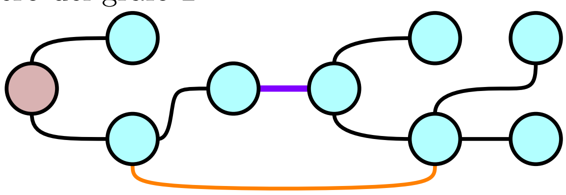
Osservazione : Qualsiasi arco coinvolto in un ciclo, non è un ponte, viceversa, se un arco è un ponte, allora non fa parte di un ciclo.

Ne consegue che, qualsiasi arco che non fa parte dell'albero di visita derivante da una qualsiasi applicazione del DFS (gli archi all'indietro), sicuramente non è un ponte. Quindi un ponte fa parte dell'albero di visita, non è necessario controllare tutti gli archi. Si considerino i due seguenti alberi di visita di due grafi che differiscono esclusivamente per un arco.

albero del grafo 1



albero del grafo 2



Il vertice rosso è la radice, ossia il punto da cui è partito il DFS, l'arco colorato di viola invece, è l'arco di cui vogliamo capire se sia un ponte o no. L'arco arancione è l'unico elemento che è presente nel grafo 2, ma non è presente nel grafo 1.

Osservando la seguente immagine, si noti che, l'arco viola del primo grafo, è un ponte, invece l'arco viola del secondo grafo, non lo è, rimane infatti connesso grazie all'arco arancione, che non fa parte dell'albero di visita, da tali considerazioni, si giunge alla seguente proposizione.

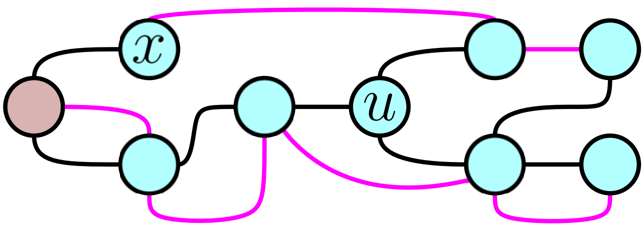
Proposizione : Sia T un albero di visita derivante da una qualsiasi applicazione del DFS su un grafo connesso e non diretto, e sia $(u, v) \in E(T)$, un arco dell'albero, dove v è il padre di u, si ha che, l'arco (u, v) è un ponte *se e solo se* non esiste alcun arco all'indietro da un qualsiasi vertice discendente di u, ad un qualsiasi vertice antenato di v (v compreso).

Dimostrazione : $\boxed{(1) \implies (2)}$: Sia T_u l'insieme dei discendenti di u. Se esistesse un arco da T_u all'indietro, allora (u, v) sarebbe parte di un ciclo, e sicuramente non sarebbe un ponte.
$\boxed{(2) \implies (1)}$: Assumiamo che (u, v) non sia un ponte, allora esiste un cammino $u \to v$ che non fa uso dell'arco in questione. Esiste sicuramente un punto nel cammino, in cui si passa da un vertice x tale che $x \notin T_u$, ad un vertice y tale che $y \in T_u$, ma sappiamo che non esistono archi all'indietro, ciò porta ad una contraddizione. ■

Scriviamo adesso lo pseudocodice di un algoritmo che restituisce tutti i ponti di un grafo in tempo lineare, come prima, diamo la definizione di *punto di back*, o semplicemente *back*.

Definizione : Il *back* di un vertice u in un albero di visita, non è altro che il vertice più vicino alla radice che è possibile raggiungere con un arco da u o da uno dei suoi discendenti.

in rosso la radice. Gli archi rosa non fanno parte dell'albero di visita.



In questo albero di visita, il back di u risulta essere x.

Vogliamo quindi un algoritmo che, per un arco (u, v), dove v è il padre di u, si controlli il back di u, se esso è presente fra v ed i suoi antenati, allora l'arco non è un ponte, altrimenti lo è.

Pseudocodice

```
Ponti(G : grafo connesso non diretto){     // funzione globale
    t :  int[n] = [0,0...,0]
    c :  int = 0
    Ponti :  list     // l'output
    z = un vertice a caso di G
    DFSponte(G,z,z,t,c,Ponti)
    return Ponti
}

DFSponte(G, v, z, t, c, Ponti){     // funzione ricorsiva, z è il padre di v
    back=t[v]
    c++
    t[v]=c
    for each (w adiacente di v){
        if(t[w]==0){
            b=DFSponte(G,w,v,t,c,Ponti)
            back = min(b,back)
        }
        else if(w≠z){
            back = min(t[w],back)
        }
    }
    if(back==t[v]){
        Ponti.add((v,z))
    }
    return back
}
```

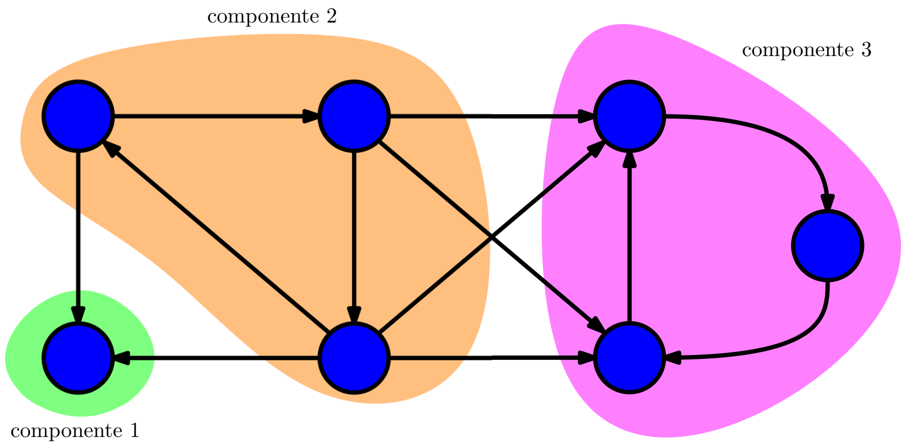
1.7 Componenti Fortemente Connesse

Abbiamo già dato la definizione di fortemente connesso per un grafo diretto, ossia un grafo G di cui, per ogni coppia di vertici (u, v) esiste un cammino da u a v e viceversa. Quando un grafo è non diretto, risulta facile trovare le componenti connesse, in quanto è facilmente visualizzabile come un "pezzo" di grafo connesso distaccato dal resto.

In un grafo diretto, una componente è un sottografo fortemente connesso *massimale*, ossia, che non è contenuto in un sottografo più grande fortemente connesso.

Osservazione : Ogni vertice di un grafo diretto è contenuto in un componente fortemente connesso, dato che al minimo esiste il componente costituito dall'unico vertice.

Osservazione : Non esistono più componenti che hanno vertici in comune, ogni vertice appartiene ad un solo componente.

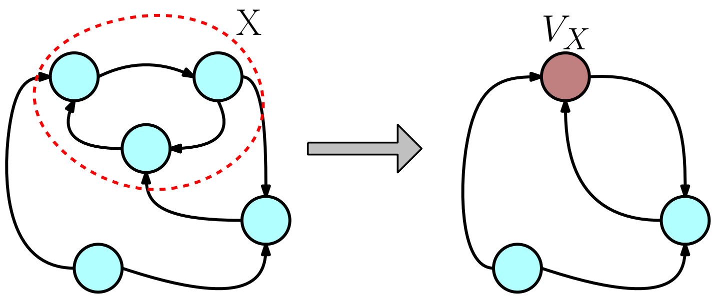


Si vuole un algoritmo capace di trovare le componenti fortemente connesse di un grafo diretto.

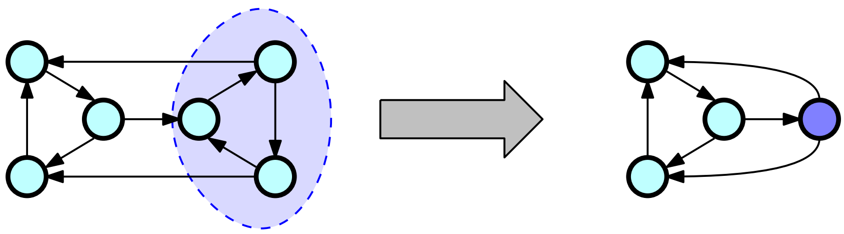
1.7.1 Contrazione di Vertici

Definizione : Sia G un grafo diretto, e sia $H \in V(G)$ un insieme di vertici, è possibile *contrarre* i vertici, facendoli "collassare" in un unico vertice, ottenendo il grafo G contratto H, denotato ${}^{G}/_{H}$. Si denota con V_H il nuovo vertice contratto.

- $V({}^{G}/_{H}) := (V(G)\backslash H) \cup \{V_H\}$
- $E({}^{G}/_{H}) := \{(x, y) \in E(G)|x, y \notin H\} \cup$
  $\{(w, V_H) \text{ se } \exists(w, y)|w \notin V_H \wedge y \in V_H\} \cup$
  $\{(V_H, w) \text{ se } \exists(y, w)|w \notin V_H \wedge y \in V_H\}$



Proposizione : Se G è un grafo fortemente connesso ed H è un sottografo connesso, allora ${}^{G}/_{H}$ è ancora un grafo fortemente connesso.



Dimostrazione : Nel grafo originale G esiste un cammino P da un qualsiasi nodo x ad un qualsiasi nodo contenuto in H, ed esiste un cammino Q da un qualsiasi nodo in H ad un qualsiasi nodo x, siano questi cammini quelli più corti possibile, essi per definizione di ${}^{G}/_{H}$

saranno anche in ${}^{G}/_{H}$, quindi esisterà un cammino da da x a V_H e da V_H ad x, quindi il grafo ${}^{G}/_{H}$ è fortemente connesso. ■

Osservazione : Se G è fortemente connesso e non è banale, allora contiene sicuramente un ciclo, quindi esiste un arco (x, y) per cui esiste anche un cammino da y ad x, che insieme all'arco precedente compone il ciclo. Un ciclo inoltre è un sottografo fortemente connesso, se applichiamo la contrazione ricorsivamente sui cicli, otterremo le componenti connesse.

Se $u_1, u_2 \ldots, u_k$ sono fortemente connessi in ${}^{G}/_{C}$, con C un insieme e V_C il vertice contratto, si ha che le componenti in G sono $u'_1, u'_2 \ldots, u'_k$ con:

$$u'_i = \begin{cases} u_i \text{ se } V_C \notin u_i \\ (u_i \backslash \{V_C\}) \cup \{V(C)\} \text{ se } V_C \in u_i \end{cases}$$

Se il grafo non ha cicli, ogni vertice è un componente connesso. Vediamo ora l'algoritmo non lineare.

Pseudocodice

```
Fort(G graph){

    C = un ciclo in G
    if (C non esiste){      // non ci sono cicli nel grafo
        return {{v}|v ∈ V(G)}
    }
    G=G/C
    V_C = vertice contratto
    (u_1, u_2 ..., u_k)=Fort(G)
    for (i in 1...k){
        if (V_C ∉ u_i){
            u'_i = u_i
        }
        else {
            u'_i = (u_i\V_C) ∪ {V(C)}
        }
    }
    return (u'_1, u'_2 ..., u'_k)
}
```