Progettazione di Algoritmi

Alessio Bandiera

Informatica, La Sapienza

Indice

Capitolo 1

Teoria dei Grafi

1.1 Grafi

1.1.1 Definizioni

Definizione 1.1.1.1 (Grafo)**.** Un grafo è una struttura matematica descritta da vertici, collegati da archi. Un grafo viene descritto formalmente come $G = (V, E)$, dove i $v \in V$ sono i *vertici* o *nodi* del grafo, mentre gli $e \in E$ sono gli *archi* (dall'inglese *edges*). In particolare, $V(G)$ è l'insieme dei vertici di G, comunemente indicato con n, mentre $E(G)$ è l'insieme degli archi di G, comunemente indicato con m. Presi due vertici $v_1, v_2 \in V(G)$, allora $(v_1, v_2) \in E(G)$ è l'arco che li collega.

Osservazione 1.1.1.1. $E(G) \subseteq V^2$.

Definizione 1.1.1.2 (Vertici adiacenti)**.** $v_1, v_2 \in V(G)$ sono detti *adiacenti* se $(v_1, v_2) \in E(G)$; in tal caso, si usa la notazione $v_1 \sim v_2$.

Definizione 1.1.1.3 (Sottografo)**.** Dato un grafo $G = (V, E)$, un sottografo G' di G è un grafo della forma $G' = (V', E') : \begin{cases} V' \subseteq V \\ E' \subseteq E \end{cases}$. Si noti che G è sottografo di sè stesso.

Definizione 1.1.1.4 (Grafo indiretto)**.** Un grafo è detto *indiretto* se gli archi non hanno direzione, o equivalentemente

$$\forall v_1, v_2 \in V(G) \quad (v_1, v_2) = (v_2, v_1) \in E(G)$$

Esempio 1.1.1.1 (Grafo indiretto)**.** Ad esempio, si consideri questo grafo indiretto:

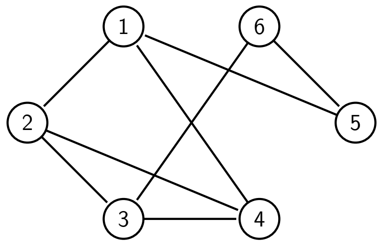

Figura 1.1: Un grafo indiretto.

in esso, si hanno

$$V(G) = \{1, 2, 3, 4, 5, 6\}$$

$$E(G) = \{(1, 2), (1, 4), (1, 5)(2, 3), (2, 4), (3, 4), (3, 6), (5, 6)\}$$

Definizione 1.1.1.5 (Grafo diretto)**.** Un grafo è detto *diretto* se gli archi hanno direzione, o equivalentemente

$$\forall v_1, v_2 \in V(G) \quad (v_1, v_2) \neq (v_2, v_1) \in E(G)$$

Esempio 1.1.1.2 (Grafo diretto)**.** Ad esempio, si consideri questo grafo diretto:

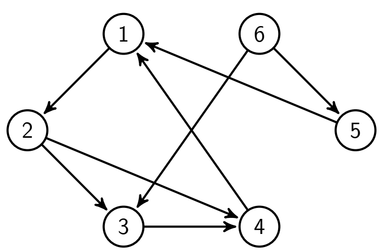

Figura 1.2: Un grafo diretto.

in esso, si hanno

$$V(G) = \{1, 2, 3, 4, 5, 6\}$$

$$E(G) = \{(1, 2), (2, 3), (2, 4), (3, 4), (4, 1), (5, 1), (6, 3), (6, 5)\}$$

Definizione 1.1.1.6 (Grado)**.** Il *grado* di un vertice $v \in V(G)$ è il numero di archi incidenti su v, indicato con $\deg(v)$.

Lemma 1.1.1.1 (Somma dei gradi)**.** *Dato un grafo G, la somma dei gradi dei vertici è pari a $2|E(G)|$.*

Dimostrazione. Sia G un grafo. Allora, ogni arco $e \in E(G)$ collega due vertici; allora necessariamente $\displaystyle\sum_{v \in V(G)} \deg(v) = 2|E(G)|$. $\square$

Definizione 1.1.1.7 (Cappio)**.** Un arco con estremi coincidenti è detto *cappio*.

Esempio 1.1.1.3 (Grafo con cappio)**.** Un esempio di grafo con cappio è il seguente:

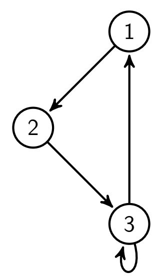

Figura 1.3: Un grafo diretto con cappio in 3.

Definizione 1.1.1.8 (Grafo semplice)**.** Un grafo è detto *semplice* se non contiene cappi, né lati multipli, ovvero più archi per due vertici.

1.1.2 Visite

Definizione 1.1.2.1 (Passeggiata)**.** Una *passeggiata* è una sequenza di vertici ed archi, della forma $\{v_0, e_1, v_1, e_2, \ldots, e_{k-1}, v_{k-1}, e_k, v_k\}$, dove $e_i = (v_{i-1}, v_i)$. È la visita di un grafo più generale, ed è possibile ripercorrere ogni arco ed ogni vertice.

Osservazione 1.1.2.1. La lunghezza massima di una passeggiata su un grafo è infinita.

Definizione 1.1.2.2 (Passeggiata chiusa)**.** Una passeggiata si dice *chiusa* se è della forma $\{v_0, e_1, v_1, e_2, \ldots, e_{k-1}, v_{k-1}, e_k, v_0\}$, dunque il primo e l'ultimo vertice coincidono.

Definizione 1.1.2.3 (Traccia)**.** Una *traccia* è una passeggiata aperta, in cui non è possibile ripercorrere gli archi, ma è possibile ripercorrere i vertici.

Esempio 1.1.2.1 (Traccia di un grafo)**.** Ad esempio, si consideri questo grafo indiretto:

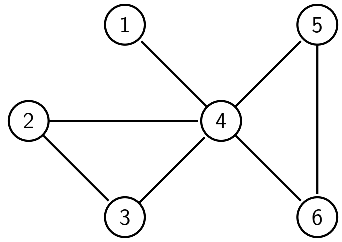

Figura 1.4: Un grafo indiretto.

in esso, si ha la traccia

$$\{5,(5,4),4,(4,3),3,(3,2),2,(2,4),4,(4,6),6\}$$

Definizione 1.1.2.4 (Circuito)**.** Un *circuito* è una traccia chiusa.

Definizione 1.1.2.5 (Cammino)**.** Un *cammino* è una traccia aperta, in cui non è possibile ripercurrere i vertici. In simboli, $\forall v, v' \in V(G) \quad v \to v'$ indica che è possibile raggiungere v' attraverso un cammino; inoltre, è possibile estendere tale sintassi anche agli archi.

Osservazione 1.1.2.2. In una passeggiata in cui non si ripercorrono i vertici, non è possibile ripercorrere gli archi

Definizione 1.1.2.6 (Ciclo)**.** Un *ciclo* è un cammino chiuso.

Esempio 1.1.2.2 (Cicli di un grafo)**.** Ad esempio, si consideri questo grafo indiretto:

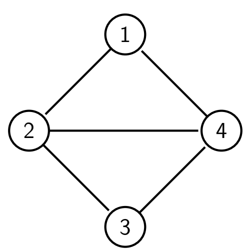

Figura 1.5: Un grafo indiretto.

in esso, si hanno tre cicli:

$$\{2,(2,4),4,(4,3),3,(3,2),2\}$$

$$\{2,(2,4),4,(4,1),1,(1,2),2\}$$

$$\{1,(1,2),2,(2,3),3,(3,4),4,(4,1),1\}$$

Definizione 1.1.2.7 (Ordinamento topologico)**.** I vertici di un grafo diretto si definiscono *ordinati topologicamente*, se disposti in modo tale che ogni vertice viene prima di tutti i vertici collegati ai suoi archi uscenti.

Esempio 1.1.2.3 (Ordinamento topologico)**.** Ad esempio, nel seguente grafo sono presenti vari ordinamenti topologici:

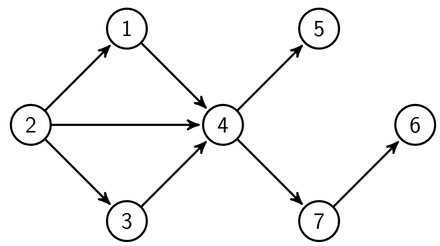

Figura 1.6: Un grafo diretto con ordinamenti topologici.

ad esempio, uno di questi è $\{2, 3, 1, 4, 5, 7, 6\}$.

Teorema 1.1.2.1 (Ordinamento topologico)**.** *Un grafo diretto G ha un ordinamento topologico se e solo se è aciclico.*

Dimostrazione.

Prima implicazione. Per assurdo, sia G un grafo diretto ciclico, avente dunque almeno un ciclo, con un ordinamento topologico, e siano $\{v_0, \ldots, v_{k-1}, v_0\}$ i vertici che costituiscono uno dei cicli di G; allora, si ha che $v_0 \to v_1 \to \ldots \to v_{k-1} \to v_0$, dunque all'interno dell'ordinamento topologico v_0 dovrebbe essere posto contemporaneamente prima e dopo $v_1, \ldots, v_{k-1}$ ↯.

Seconda implicazione. Sia G un grafo aciclico; allora per definizione, all'interno di esso non esistono cicli, ed è dunque possibile enumerare in sequenza ogni vertice G, senza creare dipendenze circloari, per poter trovare un ordinamento topologico del grafo.

□

Corollario 1.1.2.1 (Vertici particolari)**.** *In un grafo diretto aciclico, esiste almeno un vertice senza archi entranti, ed almeno un vertice senza archi uscenti.*

Dimostrazione. Per il teorema Teorema 1.1.2.1, è sufficiente considerare un ordinamento topologico del grafo, dove in esso il primo vertice non ha archi entranti, mentre l'ultimo non ha archi uscenti. □

Definizione 1.1.2.8 (Pozzo universale)**.** Sia G un grafo diretto; $v \in V(G)$ è detto *pozzo universale* se ha $n-1$ archi entranti, e nessun arco uscente.

Esempio 1.1.2.4 (Pozzo universale)**.** Ad esempio, il seguente grafo diretto presenta un pozzo universale in 3:

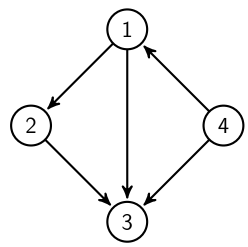

Figura 1.7: Un grafo con pozzo universale in 3.

Teorema 1.1.2.2 (Unicità del pozzo universale)**.** *Sia G un grafo diretto, e $p \in V(G)$ un suo pozzo universale; allora p è unico in G.*

Dimostrazione. Per assurdo, sia $p' \in V(G)$ un secondo pozzo universale in G; allora, per definizione, sia p che p' avrebbero $n-1$ archi entranti, e nessun arco uscente, ma questo non è possibile poiché l'unico modo per avere entrambe le condizioni verificate sarebbe attraverso un arco bidirezionale tra p e p' $\lightning$. □

Definizione 1.1.2.9 (Arborescenza)**.** Sia G un grafo diretto, e v un suo vertice; l'insieme degli archi raggiungibili da v formano l'*arborescenza di* v, e v prende il nome di *radice*. In simboli

$$A_v := \{(v', v'') \in E(G) : v \to (v', v'')\} \subseteq E(G)$$

è l'arborescenza di v. Si noti che, spesso, il sottografo generato dall'arborescenza di v viene identificato con l'arborescenza stessa.

Esempio 1.1.2.5 (Arborescenza)**.** Si consideri il seguente grafo diretto:

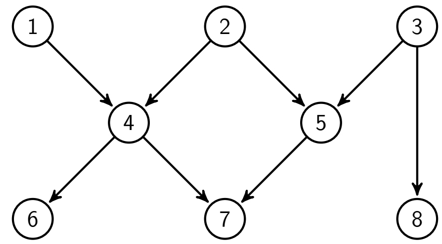

Figura 1.8: Un grafo diretto.

in esso, ad esempio il sottografo dell'arborescenza di 3 è

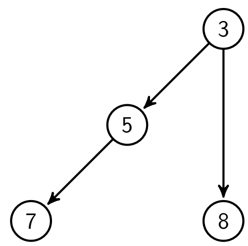

Figura 1.9: Arborescenza di 3.

Definizione 1.1.2.10 (Grafo connesso)**.** Un grafo è detto *connesso* se per ogni $v_1, v_2 \in V(G)$ esiste un cammino che li collega. Nel caso dei grafi diretti, è sufficiente avere $v_1 \to v_2$, oppure $v_2 \to v_1$.

Esempio 1.1.2.6 (Grafo non connesso)**.** Ad esempio, si consideri questo grafo:

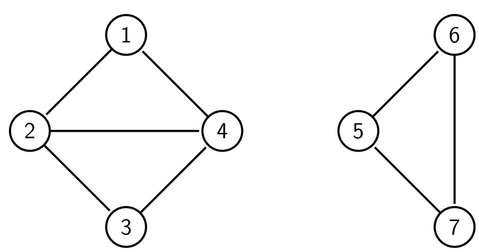

Figura 1.10: Un grafo non connesso.

Poiché non esiste cammino che possa collegare 4 e 5, il grafo non è connesso.

Definizione 1.1.2.11 (Albero)**.** Un grafo indiretto è detto *albero* se è connesso ed aciclico.

Esempio 1.1.2.7 (Albero)**.** Un esempio di albero è il seguente:

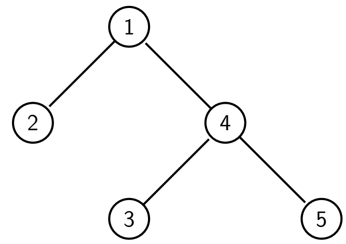

Figura 1.11: Un albero.

Definizione 1.1.2.12 (Grafo fortemente connesso)**.** Un grafo diretto è detto *fortemente connesso* se per ogni $v_1, v_2 \in V(G)$ esistono due cammini diretti, che li collegano in entrambe i versi; allora, è necessario avere $v_1 \to v_2$ e $v_2 \to v_1$. Si noti che ogni grafo indiretto connesso è anche fortemente connesso, poiché gli archi non hanno direzione.

Esempio 1.1.2.8 (Grafo fortemente connesso)**.** Un esempio di grafo fortemente connesso è il seguente:

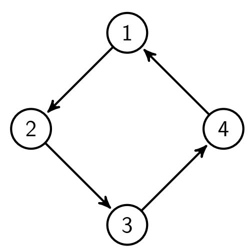

Figura 1.12: Un grafo fortemente connesso.

Definizione 1.1.2.13 (Ponte)**.** Sia G un grafo, e siano $u, v \in V(G)$ due suoi vertici; allora (u, v) è detto *ponte* se non è contenuto in nessun ciclo di G.

Esempio 1.1.2.9 (Ponte)**.** Un esempio di grafo che presenta un ponte è il seguente:

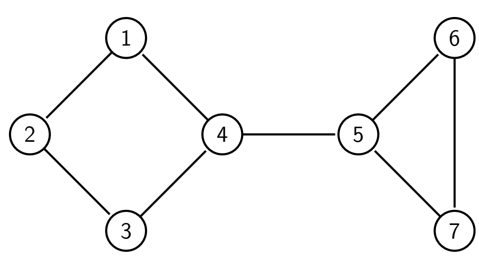

Figura 1.13: Un grafo con un ponte.

Infatti, $(4,5)$ è un ponte poiché gli unici due cicli di G sono

$$\{1,(1,2),2,(2,3),3,(3,4),4,(4,1),1\}$$
$$\{5,(5,6),6,(6,7),7,(7,5),5\}$$

Definizione 1.1.2.14 (Passeggiata euleriana)**.** Una passeggiata si dice *euleriana* se attraversa ogni arco del grafo, senza ripercorrerne nessuno.

Osservazione 1.1.2.3. Una passeggiata euleriana è una traccia passante per ogni arco del grafo.

Esempio 1.1.2.10 (Passeggiata euleriana)**.** Ad esempio, si consideri il seguente grafo indiretto:

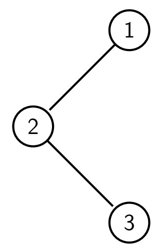

Figura 1.14: Un grafo indiretto.

in esso, l'unica passeggiata euleriana è

$$\{1,(1,2),2,(2,3),3\}$$

Teorema 1.1.2.3. *Dato un grafo G, esiste un circuito euleriano su G se e solo se G è connesso, ed ogni grado dei vertici di G è pari.*

Dimostrazione.

Prima implicazione. Sia G un grafo avente un circuito euleriano; per assurdo, sia $v \in V(G) : \deg(v)$ non sia pari. Allora, percorrendo G secondo il circuito euleriano, giungendo a v non si potrebbe più lasciare tale vertice senza riattraversare uno degli archi gia visitati $\lightning$. Inoltre, se G non fosse connesso, il circuito non potrebbe essere euleriano poiché non potrebbe attraversare tutti gli archi di G.

Seconda implicazione. Omessa.

$\square$

Definizione 1.1.2.15 (Passeggiata hamiltoniana)**.** Una passeggiata si dice *hamiltoniana* se attraversa ogni nodo del grafo, senza ripercorrerne nessuno.

Osservazione 1.1.2.4. Una passeggiata hamiltoniana è un cammino.

1.2 Rappresentazione

1.2.1 Matrici di adiacenza

Definizione 1.2.1.1 (Matrice di adiacenza)**.** Sia $G = (V, E)$ un grafo; allora, è possibile rappresentare G attraverso una matrice $M_G \in \mathrm{Mat}_{n\times n}(\{0,1\})$, dove

$$\forall m_{i,j} \in M_G \quad m_{i,j} = \begin{cases} 1 & i \sim j \\ 0 & i \nsim j \end{cases}$$

Osservazione 1.2.1.1 (Spazio di una matrice)**.** Lo spazio utilizzato da una matrice di adiacenza è pari a $O(n^2)$, poiché è necessario rappresentare l'adiacenza di ogni vertice con ogni altro.

Osservazione 1.2.1.2 (Aggiornamento di una matrice)**.** Per ogni grafo G indiretto, si ha che M_G è simmetrica; di conseguenza, il costo per aggiornare la corrispondente matrice di adiacenza è $2O(1) = O(2) = O(1)$, poiché per $v_i, v_j \in V(G)$ non coincidenti, sarà necessario aggiornare $M_G[i,j]$ e $M_G[j,i]$.

Osservazione 1.2.1.3 (Eliminazione di un nodo)**.** Per eliminare un nodo da un grafo indiretto, sarà necessario eliminare tutti i suoi collegamenti, e dunque il costo risulta essere $O(n)$, rimuovendo interamente la sua riga e la sua colonna.

Osservazione 1.2.1.4 (Controllo di adiacenza)**.** Per controllare che $v_i, v_j \in V(G)$ non coincidenti siano adiacenti, sarà sufficiente controllare $M_G[i,j] = M_G[j,i]$, e dunque il costo di un controllo è $O(1)$.

1.2.2 Liste di adiacenza

Definizione 1.2.2.1 (Liste di adiacenza)**.** Sia $G = (V, E)$ un grafo; allora, è possibile rappresentare G attraverso liste di adiacenza, salvando dunque una lista per ogni vertice, contenente i vertici ad esso adiacenti; in simboli

$$\forall v \in V(G) \quad v : [\hat{v} \in V(G) - \{v\} \mid \hat{v} \sim v]$$

Osservazione 1.2.2.1 (Spazio delle liste)**.** Dato un certo $v \in V(G)$, la lista di adiacenza corrispondente ha lunghezza $\deg(v)$; allora, il numero di elementi nelle liste di adiacenza, per il Lemma 1.1.1.1, è pari a $O\left(\sum_{v\in V(G)} \deg(v)\right) =$

$O\left(2|E(G)|\right) = O(2m) = O(m)$. Si noti inoltre che, per un grafo con pochi archi, nonostante si abbiano le liste poco riempite, è comunque necessario salvare i puntatori a tali liste, e dunque è necessario introdurre un $O(n)$ nel costo totale dello spazio, ottenendo allora $O(n) + O(m) = O(n + m)$.

Osservazione 1.2.2.2 (Controllo di adiacenza)**.** Per controllare che due nodi $v, v' \in V(G)$ siano adiacenti, è necessario controllare, ad esempio, se v' è contenuto nella lista di v, e dunque il costo per tale controllo è $O(\deg(v))$. Si noti che, nel caso peggiore, il grafo rappresentato da liste di adiacenza sarà composto da una sola lista per un certo $v \in V(G)$, contenente ogni altro vertice del grafo $\hat{v} \in V(G) - \{v\}$, e la lunghezza della lista di adiacenza di v sarà $n - 1$. Di conseguenza, nel caso peggiore, il costo per controllare se due vertici sono adiacenti è $O(n)$.

Osservazione 1.2.2.3 (Eliminazione di un nodo)**.** Per effettuare la rimozione di un nodo da un grafo, è necessario rimuoverlo da ogni lista di adiacenza in cui compare, e nel caso peggiore esso ha archi verso tutti gli altri nodi; allora, il costo di tale operazione è dato dal maggiore tra n ed m, e dunque $O(n) + O(m) = O(n + m)$.

Osservazione 1.2.2.4 (Grafo diretto)**.** Si noti che per grafi diretti è necessario effettuare una scelta di rappresentazione: all'interno delle liste è possibile salvare i vertici entranti, i vertici uscenti, o entrambi (assegnando due liste ad ogni vertice).

Esempio 1.2.2.1 (Rappresentazione di un grafo)**.** Ad esempio, si consideri il seguente grafo G:

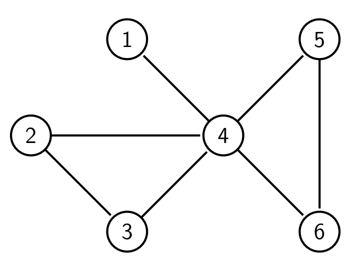

Figura 1.15: Un grafo indiretto.

allora, la sua corrispondente matrice di adiacenza è

$$M_G = \begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 \end{pmatrix}$$

mentre le corrispondenti liste di adiacenza sono

$$\begin{cases} 1 : [4] \\ 2 : [4, 3] \\ 3 : [2, 4] \\ 4 : [1, 3, 5, 6] \\ 5 : [4, 6] \\ 6 : [4, 5] \end{cases}$$

1.3 DFS (Depth-first Search)

1.3.1 Trovare un ciclo

Algoritmo 1.3.1.1 Dato un grafo indiretto G, con ogni vertice avente grado almeno pari a 2, l'algoritmo restiuisce un ciclo di G.
Input: G grafo indiretto, tale che $\forall v \in V(G) \quad \deg(v) \geq 2$.
Output: un ciclo di G.

function $\textsc{findCycle}(G)$
  $v \in V(G)$ ▷ un vertice qualsiasi di G
  `visited` $:= \{v\}$ ▷ conterrà i vertici visitati
  $v' \in V(G) : v \sim v'$
  while $v' \notin$ `visited` **do** ▷ tempo costante perché `visited` è un set
    `visited.add(`v'`)`
    $v' := v'' \in V(G) : \begin{cases} v' \sim v'' \\ v'' \neq \texttt{visited[visited.length()} - 2\texttt{]} \end{cases}$
  end while
  return `visited[visited.indexOf(`v'`):visited.length()]`
end function

Osservazione 1.3.1.1 (Correttezza dell'algoritmo)**.** L'algoritmo inizia scegliendo un qualsiasi vertice di G, denotato alla riga 2 con v; successivamente, alla riga 3 viene inizializzato un array `visited` che contterà tutti i vertici visitati attraverso l'algoritmo; inotre, alla riga 4 viene scelto un altro vertice v', che sia adiacente al v di partenza.

All'interno del ciclo `while`, alla riga 6 l'algoritmo salva v' all'interno dell'array di vertici visitati, mentre alla riga 7 viene rimpiazzato v', scegliendo un nuovo vertice, adiacente a v', che sia diverso dal penultimo vertice inserito all'interno di `visited`. Il motivo per cui quest'ultimo controllo è necessario, è che il penultimo vertice inserito sarà il vertice dal quale v' proveniva, di conseguenza si rischierebbe di ripercorrere uno stesso vertice più di una volta, e dunque non si formerebbe un ciclo. Si noti che è necessaria l'ipotesi per cui G abbia ogni vertice di grado almeno pari a 2, altrimenti non sarebbe possibile trovare un vertice differente dal penultimo di `visited`. Il ciclo termina nel momento in cui viene scelto un v' già presente all'interno di `visited`, in quanto, poiché non è possibile ripercorrere i propri passi, l'unica possibilità in cui si è giunti ad un vertice già visitato è se si è concluso un ciclo.

L'algoritmo termina restituendo uno slice dell'array, partendo dal primo indice di v' disponibile (si noti che alla fine dell'algoritmo anche l'ultimo elemento di `visited` sarà v'), fino alla fine.

Osservazione 1.3.1.2. Si noti che `visited` contiene tutti i nodi visitati, dunque restituire interamente l'array potrebbe non fornire un ciclo, come nel seguente grafo

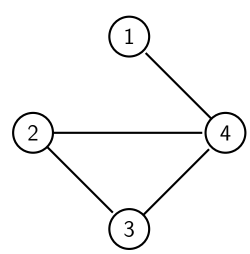

Figura 1.16: Un grafo diretto contenente un ciclo.

in cui, ad esempio partendo da $v = 4$, l'unico ciclo è

$$\{4, (4, 3), 3, (3, 2), 2, (2, 4), 4\}$$

nonostante al termine dell'algoritmo si avrebbe `visited` $= [1, 4, 3, 2, 4]$, che non costituisce un ciclo.

Osservazione 1.3.1.3 (Costo dell'algoritmo)**.** Il costo di questo algoritmo dipende dalla struttura dati utilizzata per rappresentare il grafo in input: nel caso in cui G è rappresentato attraverso una matrice di adiacenza, il costo del ciclo `while` è pari a $O(n)$, poiché la riga 7 richiede di trovare un $v'' \in V(G) : v' \sim v''$, il ché potrebbe portare a dover scorrere tutta la riga/colonna di v'', dunque nel caso peggiore $O(n)$; differentemente, rappresentando G attraverso liste di adiacenza, basta scegliere il primo vertice contenuto nella lista di v'', e se questo dovesse coincidere con il penultimo elemento di `visited`, sarà sufficiente scegliere il secondo elemento della lista (sicuramente presente per come G è scelto in ipotesi), dunque si ha $O(2) = O(1)$.

Infine, si noti che il ciclo `while` ha costo $O(n)$, poiché nel caso peggiore si ha un ciclo che percorre tutto il grafo.

Allora, attraverso una rappresentazione matriciale, l'algoritmo ha costo $O(n) \cdot O(n) = O(n^2)$, mentre attraverso la rappresentazione con liste di adiacenza, si ha $O(1) \cdot O(n) = O(n)$.

1.3.2 Visita in DFS

Definizione 1.3.2.1 (DFS)**.** Con DFS si indica il criterio di visita di un grafo; in particolare, DFS sta per *Depth-first Search*, dunque la visita del grafo avviene procendendo sempre più in profondità, retrocedendo esclusivamente se non è più possibile avanzare.

Algoritmo 1.3.2.1 Prima versione dell'algoritmo; dato un grafo indiretto G, e un suo vertice v, l'algoritmo restituisce tutti i vertici, raggiungibili attraverso cammini, partendo da v.
Input: G grafo indiretto; v un vertice di G.
Output: i vertici raggiungibili da v.

```
1:  function findReachableNodes_1(G, v)
2:      visited := [0] * n                          ▷ array di n zeri
3:      visited[v] = 1
4:      Stack S := [v]
5:      while !S.isEmpty() do
6:          v_top = S.top()
7:          if ∃z ∈ V(G) : { z ~ v_top ; visited[z] = 0 } then
8:              S.push(z)
9:              visited[z] = 1
10:         else
11:             S.pop()
12:         end if
13:     end while
14:     return visited
15: end function
```

Dimostrazione. Per assurdo, sia $\hat{v} \in V(G)$, raggiungibile da v attraverso un cammino, che non sia stato raggiunto dall'algoritmo; allora, per definizione esiste un cammino $v, e_1, v_1, \dots, v_{n-1}, v_n, \hat{v}$; inoltre, sia v_i il vertice con indice maggiore all'interno del cammino, raggiunto dall'algoritmo, e dunque avendo che $\begin{cases} v_i \sim v_{i+1} \\ v_{i+1} \notin \texttt{visited} \end{cases}$. Allora, per costruzione dell'algoritmo, v_i sarebbe stato rimosso dallo stack, alla riga 11, prima che v_{i+1} potesse essere visitato; ma poiché $v_i \sim v_{i+1}$, allora l'algoritmo dovrebbe aver sbagliato esecuzione, poiché v_{i+1} sarebbe stato raggiunto alla riga 7 inevitabilmente $\lightning$. $\square$

Osservazione 1.3.2.1 (Costo dell'algoritmo)**.** Si consideri G rappresentato attraverso matrice di adiacenza; allora, il costo della riga 7, nel caso peggiore, è $O(n)$, poiché è necessario controllare tutta la riga/colonna di v_{top} per trovare un vertice z tale che $\texttt{visited}[z] = 0$, dunque non sia stato ancora visitato. Per ragione analoga, rappresentando G attraverso liste di adiacenza, nel caso peggiore si ha una sola lista corrispondente ad un singolo vertice di G, e sarà dunque necessario effettuare $O(n-1) = O(n)$ controlli.

Inoltre, si noti che il caso peggiore dell'algoritmo si ha quando v può raggiungere ogni altro nodo di G, e dunque il ciclo `while` sarà ripetuto $O(2n-1) = O(2n) = O(n)$ volte, poiché ogni vertice verrà inserito e rimosso dallo stack, eccetto il primo, inserito alla riga 4.

Allora, il costo complessivo dell'algoritmo, indipendentemente dalla rappresentazione di G, è pari a $O(n) \cdot O(n) = O(n^2)$.

Osservazione 1.3.2.2 (Albero)**.** Sia G un grafo indiretto; considerando l'insieme di archi attraversati dall'algoritmo per trovare ogni vertice raggiungibile partendo da v, al termine della procedura si ottiene un sottografo indiretto di G connesso ed aciclico: connesso, poiché l'algoritmo procede per adiacenza di vertici, ed aciclico, poiché l'algoritmo non visita lo stesso vertice più di una volta. Allora, per definizione, tale sottografo è un albero.

Osservazione 1.3.2.3 (Grafo diretto)**.** Si noti che l'algoritmo è valido anche per grafi diretti.

Osservazione 1.3.2.4 (Arborescenza)**.** Sia G un grafo diretto; considerando l'insieme di archi attraversati dall'algoritmo per trovare ogni vertice raggiungibile partendo da v, al termine della procedura si ottiene un sottografo diretto di G connesso ed aciclico, per gli stessi motivi dell'Osservazione 1.3.2.2; tale sottografo è un arborescenza di v.

Algoritmo 1.3.2.2 Seconda versione dell'algoritmo; dato un grafo indiretto G, rappresentato attraverso liste di adiacenza, e un suo vertice v, l'algoritmo restituisce tutti i vertici, raggiungibili attraverso cammini, partendo da v.
Input: G grafo indiretto, rappresentato attraverso liste di adiacenza; v un vertice di G.
Output: i vertici raggiungibili da v.

```
1:  function findReachableNodes_2(G, v)
2:      visited := {v}
3:      Stack S := [v]
4:      while !S.isEmpty() do
5:          v_top = S.top()
6:          while !v_top.adjacent().isEmpty() do
7:              z := v_top.adjacent()[0]
8:              v_top.adjacent().remove(0)          ▷ fa la differenza
9:              if z ∉ visited then
10:                 visited.add(z)
11:                 S.push(z)
12:                 break
13:             end if
14:         end while
15:         if v_top == S.top() then
16:             S.pop()
17:         end if
18:     end while
19:     return visited
20: end function
```

Osservazione 1.3.2.5 (Correttezza dell'algoritmo)**.** Questa seconda versione dell'algoritmo presenta una miglioria sostanziale alla riga 8: infatti, attraverso questa riga si rimuovono di volta in volta i vertici adiacenti appena già visitati; di conseguenza, i vertici adiacenti da controllare saranno progressivamente sempre meno. Infatti, si noti che senza la riga 8, l'algoritmo si comporterebbe come la prima versione.

Il `break` alla riga 12 interrompe il ciclo `while` della riga 6, facendo sì che v_{top} della riga 5, all'iterazione successiva del `while` della riga 4, sia pari a z, dunque cambiando il vertice correntemente in esame. Di conseguenza, alla riga 15 il controllo sarà valutato a `True` esclusivamente se non è mai stata eseguita la riga 11 per tutta l'iterazione del ciclo `while` della riga 6, ovvero quando tutti i vertici adiacenti a v_{top} sono già stati visitati.

Osservazione 1.3.2.6 (Costo dell'algoritmo)**.** Poiché i nodi visitati vengono eliminati, il costo del ciclo `while` dipende da operazioni eseguite in tempo costante $O(1)$, e da quanti nodi vengono controllati per ogni iterazione del ciclo, ma poiché non si possono ricontrollare più volte gli stessi nodi, allora il costo del ciclo dipende solamente dalla dimensione delle liste di adiacenza, e dunque si ha $O\left(\sum_{v \in V(G)} O(1) + \deg(v)\right) = O\left(\sum_{v \in V(G)} O(1)\right) + O\left(\sum_{v \in V(G)} \deg(v)\right) = O(n) + O(m) = O(n+m)$, per il Lemma 1.1.1.1.

Osservazione 1.3.2.7 (Grafo diretto)**.** Per estendere questo algoritmo a grafi diretti, è necessario fornire in input un grafo rappresentato attraverso liste di adiacenza, le quali devono contenere esclusivamente i vertici uscenti, poichè sono gli unici archi percorribili.

1.3.3 Trovare un ordinamento topologico

Algoritmo 1.3.3.1 Dato un grafo diretto aciclico G, l'algoritmo restituisce un suo ordinamento topologico.
Input: G grafo diretto aciclico.
Output: un ordinamento topologico di G.

```
1: function findTopologicalSorting(G)
2:     order := []
3:     while V(G) ≠ 0 do
4:         v ∈ V(G) : v.incoming_adjacent().length() = 0
5:         order.append(v)
6:         V(G).remove(v)
7:     end while
8:     return order
9: end function
```

Osservazione 1.3.3.1 (Correttezza dell'algoritmo)**.** TODO

Osservazione 1.3.3.2 (Costo dell'algoritmo)**.** Il ciclo `while` della riga 3, indipendentemente dalla struttura di rappresentazione del grafo G, deve essere eseguito n volte, e dunque ha costo $O(n)$, poiché l'ordinamento topologico deve coinvolgere ogni nodo del grafo, e alla riga 6 i nodi controllati vengono progressivamente rimossi.

Rappresentando G attraverso matrice di adiacenza, indicando con 1 i vertici adiacenti entranti, il costo della riga 4 è pari a $O(n^2)$, poiché per trovare un vertice v che non abbia archi entranti, è necessario controllare tutta la sua riga/colonna, e dunque nel caso peggiore, per trovarlo sarà necessario controllare l'intera matrice; inoltre, per effettuare la rimozione di v alla riga 6, il costo è $O(n)$, Allora, il costo complessivo dell'algoritmo risulta essere $O(n) \cdot [O(n^2) + O(n)] = O(n) \cdot O(n^2) = O(n^3)$.

Differentemente, rappresentando G attraverso liste di adiacenza, salvando solamente i vertici adiacenti entranti per ogni nodo, alla riga 4 per trovare un nodo senza archi entranti è sufficiente controllare il numero di elementi della lista di ogni vertice, operazione a costo $O(1)$, e dunque il costo, nel caso peggiore, è $O(n-1) = O(n)$; inoltre, per rimuovere v alla riga 6, il costo è pari a $O(n+m)$. Allora, il costo complessivo dell'algoritmo risulta essere $O(n) \cdot [O(n) + O(n+m)] = O(n) \cdot [O(2n+m)] = O(n) \cdot O(n+m) = O(n \cdot (n+m))$.

1.4 Tempi di visita e di chiusura

1.4.1 Definizioni

Definizione 1.4.1.1 (Tempo di visita e di chiusura)**.** All'interno degli algoritmi che visitano grafi secondo DFS, è possibile introdurre un `counter` inizializzato ad 1, ed incrementato ogni volta che viene attraversato un *nuovo* vertice.

Allora, per ogni vertice v del grafo diretto in input, si definiscono $t(v)$, detto *tempo di visita di* v, pari al valore del `counter` la prima volta che v viene visitato, e $T(v)$, detto *tempo di chiusura di* v, pari al valore del `counter` nel momento in cui v viene rimosso dallo stack.

Inoltre, si definisce $\mathrm{Int}(v) := [t(v), T(v)]$.

Osservazione 1.4.1.1 (Intervalli delle foglie)**.** Si noti che per ogni foglia v del grafo, ovvero i vertici per i quali non è più possibile scendere di profondità, si ha $t(v) = T(v)$, per definizione stessa dei tempi.

Lemma 1.4.1.1 (Proprietà degli intervalli)**.** *Sia G un grafo diretto, e $u, v \in V(G)$ adiacenti; allora solo una delle seguenti proposizioni è vera:*

i) $\mathrm{Int}(u) \subseteq \mathrm{Int}(v)$

ii) $\mathrm{Int}(v) \subseteq \mathrm{Int}(u)$

iii) $\mathrm{Int}(u) \cap \mathrm{Int}(v) = \varnothing$

Dunque, gli intervalli o sono l'uno interamente contenuto nell'altro, o non si intersecano.

Dimostrazione. La tesi equivale a dimostrare che non può verificarsi il caso in cui c'è intersezione non vuota tra i due intervalli, e dunque non è possibile che $\mathrm{Int}(u) \cap \mathrm{Int}(v) \neq \varnothing$, ovvero $t(u) < t(v) < T(u) < T(v)$, allora:

- $t(u) < t(v) \implies u$ inserito nello stack prima di v
- $t(v) < T(u) \implies u$ viene rimosso dallo stack dopo aver visitato v, ma poiché u era sotto a v all'interno dello stack, necessariamente v deve essere stato rimosso dallo stack prima di u, e allora non è possibile che $T(u) < T(v)$ ↯.

Si noti che, avendo ad esempio $(u, v) \in E(G)$, e dunque u incidente su v, allora necessariamente $\mathrm{Int}(u) \cap \mathrm{Int}(v) \implies t(v) < T(v) < t(u) \leq T(u)$. □

Lemma 1.4.1.2 (Proprietà degli intervalli). *Sia G un grafo indiretto, e $u, v \in V(G)$ adiacenti; allora si verifica una sola tipologia di inclusione, in cui* $\mathrm{Int}(u) \subseteq \mathrm{Int}(v)$*, oppure* $\mathrm{Int}(v) \subseteq \mathrm{Int}(u)$*, e poiché gli archi non sono orientati perde di significato la distinzione tra i due casi.*

Dimostrazione. La tesi equivale a dimostrare che non può verificarsi il caso in cui c'è intersezione vuota tra i due intervalli, e dunque non è possibile che $\mathrm{Int}(u) \cap \mathrm{Int}(v) = \varnothing$, ovvero $t(u) \leq T(u) < t(v) \leq T(v)$, poiché $T(u) < t(v)$ implicherebbe che u verrebbe rimosso dallo stack prima che v possa essere inserito, e questo non è possibile per costruzione della visita DFS, poiché $u \sim v$. □

1.4.2 Categorie di archi

Osservazione 1.4.2.1 (Categorie di archi). Sia $G = (V, E)$ un grafo diretto, $\hat{v} \in V(G)$, e sia $A_{\hat{v}}$ la sua arborescenza; allora, è possibile classificare ogni arco $(u, v) \in E(G) - E(A_{\hat{v}})$, mediante $\mathrm{Int}(u)$ e $\mathrm{Int}(v)$:

- $\text{Int}(u) \subseteq \text{Int}(v)$, allora l'arco (u, v) è un *backward edge*, ovvero in avanti: sono gli archi che congiungono due nodi dello stesso ramo di $A_{\hat{v}}$, nel caso in cui v è più in profondità di u nella visita DFS

- $\text{Int}(v) \subseteq \text{Int}(u)$, allora l'arco (u, v) è un *forward edge*, ovvero all'indietro: sono gli archi che congiungono due nodi dello stesso ramo di $A_{\hat{v}}$, nel caso in cui u è più in profondità di v nella visita DFS

- $\text{Int}(v) \cap \text{Int}(u) = \varnothing$, allora l'arco (u, v) è un *cross edge*, detto *arco di attraversamento*: sono gli archi che congiungono due nodi di rami differenti in $A_{\hat{v}}$

Osservazione 1.4.2.2 (Categorie di archi)**.** Sia $G = (V, E)$ un grafo indiretto, $\hat{v} \in V(G)$, e sia $T_{\hat{v}}$ il suo albero; allora, ogni arco $(u, v) \in E(G) - E(T_{\hat{v}})$ viene classificato come *backward edge*.

Esempio 1.4.2.1. TODO

Teorema 1.4.2.1 (Presenza di cicli)**.** *Sia G un grafo indiretto connesso; allora G ha un ciclo se e solo se in esso esiste un backward edge in ogni albero.*

Dimostrazione.

> *Prima implicazione.* Per assurdo, sia G un grafo indiretto, in cui è presente almeno un ciclo, e non sono presenti backward edge; inoltre, sia $\hat{v} \in V(G)$, e sia $T_{\hat{v}}$ il suo albero. Allora, poiché G non ha backward edge, necessariamente gli unici suoi archi sono quelli che compongono $T_{\hat{v}}$, e dunque $E(G) = E(T_{\hat{v}}) \implies G = T_{\hat{v}} \implies G$ è un albero, e dunque G non ha cicli $\lightning$.
>
> *Seconda implicazione.* Sia G un grafo indiretto connesso, sia $\hat{v} \in V(G)$, $T_{\hat{v}}$ il suo albero, e sia $(u, v) \in E(G) - E(T_{\hat{v}})$ un backward edge. Allora, poiché $u, v \in V(T_{\hat{v}})$, è sufficiente considerare il cammino tale che $u \to v$, che esiste poiché $T_{\hat{v}}$ è un albero, e dunque $\{u \to v\} \cup (u, v)$ è un ciclo di G.

□

Algoritmo 1.4.2.1 Dato un grafo G, rappresentato attraverso liste di adiacenza (nel caso di G diretto, l'adiacenza è dei nodi uscenti), e un suo vertice r, l'algoritmo restituisce i tempi di visita e di chiusura dei nodi di G, relativi alla visita dell'albero, o dell'arborescenza, di r.
Input: G grafo, rappresentato attraverso liste di adiacenza; r un vertice di G.
Output: tempi di visita e di chiusura dei $v \in V(G)$, relativi all'albero, o all'arborescenza, di r.

```
function DFS(G, v, visited, c, t, T)
    for u ∈ V(G) : (v, u) ∈ E(G) do           ▷ u deve essere uscente da v
        if u ∉ visited then
            c += 1                             ▷ c viene trattato come oggetto
            t[u] = c
            visited.add(u)
            DFS(G, u, visited, c, t, T)
        end if
    end for
    T[v] = c
end function

function findTimes(G, r)
    visited := {r}
    t := [0] ∗ n
    T := [0] ∗ n
    t[r] = 1
    c := 1                                     ▷ è un oggetto
    DFS(G, r, visited, c, t, T)
    return t, T
end function
```

Osservazione 1.4.2.3 (Correttezza dell'algoritmo)**.** TODO

Osservazione 1.4.2.4 (Costo dell'algoritmo)**.** TODO

Algoritmo 1.4.2.2 Dato un grafo diretto G, rappresentato attraverso liste di adiacenza (per ogni vertice sono salvate due liste, dei vertici entranti e dei vertici uscenti), e un suo vertice v, l'algoritmo restituisce gli archi non facenti parti dell'arborescenza di v, categorizzati in base ai loro intervalli di apertura e chiusura.
Input: G grafo diretto, rappresentato attraverso liste di adiacenza; v un vertice di G.
Output: archi non dell'arborescenza, categorizzati per intervalli.

```
function categorizeEdges(G, v)
    visited := [0] * n
    visited[v] = 1
    Stack S := [v]
    c := 1
    t := [0] * n                                  ▷ tempi di visita
    T := [0] * n                                  ▷ tempi di chiusura
    t[v] = c
    parents := [0] * n
    parents[v] = v                                ▷ per riconoscere la radice
    while !S.isEmpty() do
        v_top := S.top()
        while !v_top.outgoing_adjacent().isEmpty() do
            z := v_top.outgoing_adjacent()[0]
            v_top.outgoing_adjacent().remove(0)
            if visited[z] == 0 then
                visited[z] = 1
                S.push(z)
                parents[z] = v_top
                c += 1
                t[z] = c
                break
            end if
        end while
        if v_top == S.top() then
            S.pop()
            T[v_top] = c
        end if
    end while
```

```
    forward := []
    backward := []
    cross := []
    for v ∈ V(G) do
        for u ∈ v.incoming_adjacent() do
            if parents[v] == u then
                continue                    ▷ faceva parte dell'arborescenza di v
            else if T[u] < t[v] or T[v] < t[u] then
                cross.append((u, v))
            else if T[u] ≤ T[v] then
                backward.append((u, v))
            else
                forward.append((u, v))
            end if
        end for
    end for
    return forward, backward, cross
end function
```

Osservazione 1.4.2.5 (Correttezza dell'algoritmo)**.** TODO O(n +m)

Osservazione 1.4.2.6 (Costo dell'algoritmo)**.** TODO O(n +m)

1.4.3 Trovare un ordinamento topologico

Teorema 1.4.3.1 (Presenza di cicli)**.** *Sia G un grafo diretto connesso; allora G ha un ciclo se e solo se in esso esiste un backward edge in almeno un'arborescenza.*

Dimostrazione.

Prima implicazione. TODO

Seconda implicazione. Sia G un grafo diretto fortemente connesso, e sia $\hat{v} \in V(G)$ tale che la sua arborescenza $A_{\hat{v}}$ contenga un backward edge $(u, v) \in E(G) - E(A_{\hat{v}})$. Allora, poiché $u, v \in V(T_{\hat{v}})$, e (u, v) è un backward edge, è sufficiente considerare il cammino tale che $u \to v$, che esiste poiché $A_{\hat{v}}$ è un arborescenza, e dunque $\{u \to v\} \cup (u, v)$ è un ciclo di G. Si noti che il fatto che G sia fortemente connesso garantisce di poter considerare un $\hat{v} \in V(G)$ qualsiasi.

$\square$

Corollario 1.4.3.1. *Sia G un grafo diretto aciclico connesso, sia $\hat{v} \in V(G)$ un suo vertice, sia $A_{\hat{v}}$ la relativa arborescenza, e sia $(u, v) \in E(G)$ un arco; allora $t(v) \leq T(u)$.*

Dimostrazione. Si noti che ogni arco $(u, v) \in A_{\hat{v}}$ è un forward edge per costruzione della visita DFS; allora si consideri il caso in cui $(u, v) \in E(G) - A_{\hat{v}}$. Allora per il teorema precedente, (u, v) non è un backward edge, e dunque per il Lemma 1.4.1.1 si può verificare solo una delle seguenti:

- $\mathrm{Int}(v) \subseteq \mathrm{Int}(u) \implies t(u) < t(v) \leq T(v) < T(u)$, e in particolare $t(v) \leq T(u)$
- $\mathrm{Int}(u) \cap \mathrm{Int}(v) = \varnothing \implies t(v) < T(v) < t(u) < T(u)$, e in particolare $t(v) \leq T(u)$.

$\square$

Corollario 1.4.3.2. *Sia G un grafo diretto aciclico connesso, sia $\hat{v} \in V(G)$ un suo vertice, sia $A_{\hat{v}}$ la relativa arborescenza, e sia $(u, v) \in E(G)$ un arco; allora $T(v) \leq T(u)$.*

Dimostrazione. Per il corollario precedente, per ogni arco di un grafo indiretto aciclico connesso $t(v) \leq T(u)$, allora:

- $t(v) < t(u)$: se $T(u) \leq T(v)$, allora (u, v) sarebbe un backward edge, che non è possibile avere per il Teorema 1.4.3.1; allora necessariamente $T(v) < T(u)$; ma se $t(u) < T(v)$ allora si avrebbe intersezione non vuota tra gli intervalli, impossibile per il Lemma 1.4.1.1; allora necessariamente $t(v) < T(v) < t(u) < T(u)$, e dunque (u, v) è un cross edge
- $t(u) < t(v)$: se $T(u) < T(v)$, allora gli intervalli avrebbero intersezione non vuota, e ciò non si può verificare per il Lemma 1.4.1.1; allora necessariamente $t(u) < t(v) \leq T(v) \leq T(u)$, e dunque (u, v) è un forward edge.

In particolare, si ha che $T(v) \leq T(u)$ in entrambe i casi. $\square$

Teorema 1.4.3.2 (Ordinamento topologico attraverso i tempi)**.** *Sia G un grafo diretto aciclico connesso, sia $\hat{v} \in V(G)$ un suo vertice, e sia $A_{\hat{v}}$ la sua arborescenza; allora, ordinando i vertici attraverso i loro tempi di chiusura T in ordine decrescente, si ottiene un ordinamento topologico del grafo.*

Dimostrazione. Per definizione, un'ordinamento è detto topologico se ogni vertice è posto prima dei suoi archi uscenti; inoltre, per il corollario precedente, per ogni $(u, v) \in E(G)$ si ha $T(v) \leq T(u)$, e dunque se si ordinassero i vertici di G utilizzando i tempi di chiusura T, in ordine crescente, come criterio, allora v verrebbe prima di u, e (u, v) è un arco diretto in cui u è incidente su v; allora, segue che ordinando i vertici in ordine decrescente di T, si ha un ordinamento topologico di G. □

Algoritmo 1.4.3.1 Dato un grafo diretto aciclico connesso G, rappresentato attraverso liste di adiacenza in cui vengono salvati gli archi adiacenti uscenti, l'algoritmo restituisce un ordinamento topologico di G.
Input: G grafo diretto, rappresentato attraverso liste di adiacenza.
Output: un ordinamento topologico di G.

```
1:  function DFS(G, v, visited, order)
2:      visited.add(v)
3:      for u ∈ V(G) : (v, u) ∈ E(G) do          ▷ u deve essere uscente da v
4:          if u ∉ visited then
5:              DFS(G, u, visited, order)
6:          end if
7:      end for
8:      order.append(v)                          ▷ l'ordinamento risulterà invertito
9:  end function
10:
11: function findTopologicalSortingDFS(G)
12:     order := []
13:     visited := {}
14:     for v ∈ V(G) do
15:         if v ∉ visited then
16:             DFS(G, v, visited, order)
17:         end if
18:     end for
19:     order.reverse()                          ▷ viene invertita la lista
20:     return order
21: end function
```

Osservazione 1.4.3.1 (Correttezza dell'algoritmo)**.** Si noti che l'algoritmo non salva i tempi di chiusura dei vari vertici per ordinarli, ma non è necessario grazie alla ricorsione: infatti, inserendo il vertice alla riga 8, dunque dopo il loop `for`, l'inserimento avviene in post-order rispetto alla visita del grafo, e dunque è equivalente a rispettare l'ordinamento crescente dei tempi di chiusura di ogni nodo. Dunque, è sufficiente invertire la lista ottenuta, alla riga 19, per ottenere un ordinamento topologico.

Osservazione 1.4.3.2 (Costo dell'algoritmo)**.** Si noti che l'algoritmo ha costo $O(n+m)$, poiché attua una sola visita in DFS del grafo, rappresentato attraverso liste di adiacenza, ricorsivamente.

1.4.4 Trovare un pozzo universale

Algoritmo 1.4.4.1 Dato un grafo diretto G, rappresentato attraverso matrice di adiacenza, l'algoritmo restituisce, se presente, il pozzo universale di G.
Input: G grafo diretto, rappresentato attraverso matrice di adiacenza.
Output: il pozzo universale di G, se presente.

```
function findUniversalSink(M_G)
    p ∈ V(G)                    ▷ un vertice qualsiasi, possibile pozzo universale
    for v ∈ V(G) do
        if M_G[p, v] == 1 then
            p = v
        end if
    end for
    for v ∈ V(G) − {p} do
        if M_G[p, v] == 1 then
            return None
        end if
        if M_G[v, p] == 0 then
            return None
        end if
    end for
    return p
end function
```

Osservazione 1.4.4.1 (Correttezza dell'algoritmo)**.** Per definizione stessa di pozzo universale, indipendentemente dalla scelta del vertice di partenza del grafo in input, percorrendo una qualsiasi sequenza di archi, inevitabilmente, si deve giungere al pozzo universale, poiché esso ha esattamente $n-1$ archi entranti. Allora, alla riga 2 viene arbitrariamente scelto un possibile pozzo universale, e il ciclo `for` della riga 3 percorre la catena di archi possibili, e ha costo $O(n)$. Si noti però che il vertice a cui si è giunti potrebbe non essere un pozzo universale, ad esempio si consideri questo grafo:

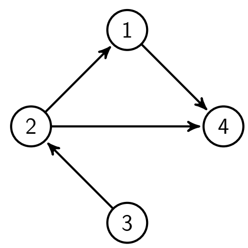

Figura 1.17: Un grafo che contiene un possibile pozzo universale.

si noti che ndipendentemente dalla scelta iniziale di $p \in V(G)$, al termine del ciclo `for` si avrà $p = 4$, pur non essendo 4 un pozzo universale, poiché $(3, 4) \notin E(G)$. Risulta dunque necessario accertarsi che p sia realmente un pozzo universale, andando dunque a controllare, all'interno del ciclo `for` della riga 8, se p non ha archi uscenti, ed ogni altro arco è incidente su p. Tali controlli vengono effettuati rispettivamente alla riga 9, in cui il ciclo termina restituendo `None` se esiste un vertice uscente da p, e alla riga 12, in cui il ciclo termina analogamente se esiste un vertice non adiacente entrante a p.

Osservazione 1.4.4.2 (Costo dell'algoritmo)**.** Poiché i due cicli `for` percorrono ogni vertice di G, una ed una sola volta, il costo dell'algoritmo è pari a $O(n) + O(n) = O(n)$.

1.4.5 Trovare i ponti

Lemma 1.4.5.1. *Sia G un grafo indiretto connesso, $x \in V(G)$, e sia T_x l'albero di visita in DFS di x in G; sia $y \in T_x$, e sia $T_y \subseteq T_x$ il suo sottoalbero in T_x; se esiste un arco $(u, v) \in E(G)$, tale che $u \in T_y$ e $v \in T_x - T_y$, allora esiste un ciclo in G, contenente (x, y).*

Dimostrazione. Si noti che, poiché G è indiretto e connesso, per qualsiasi $x \in V(G)$, $V(T_x) = V(G)$, dunque la visita in DFS è sempre in grado di raggiungere ogni nodo del grafo G. Per definizione, T_x è connesso ed aciclico; poiché T_x è radicato in x, allora il cammino $u \to v$ all'interno di T_x, che esiste poiché T_x è connesso, deve necessariamente passare per (x, y), poiché T_x è aciclico, e $v \in T_y$; allora, $\{u \to v\} \cup (u, v)$ è necessariamente un ciclo. $\square$

Lemma 1.4.5.2 (Presenza di ponti)**.** *Sia G un grafo indiretto connesso, $x \in V(G)$, e sia T_x l'albero di visita in DFS di x in G; sia $y \in T_x$, e sia $T_y \subseteq T_x$ il suo sottoalbero in T_x; se non esistono archi $(u, v) \in E(G)$, tali che $u \in T_y$ e $v \in T_x - T_y$, allora (x, y) è un ponte.*

Dimostrazione. Per assurdo, (x, y) non è un ponte; allora, per definizione, (x, y) è incluso in almeno un ciclo di G, e dunque esiste necessariamente un cammino $x \to y$ non passante per (x, y); allora, tale cammino deve necessariamente contenere un arco (u, v), in cui $u \in T_y$, e $v \in T_x - T_y$ $\lightning$. $\square$

Algoritmo 1.4.5.1 Dato un grafo indiretto G, rappresentato attraverso liste di adiacenza, l'algoritmo restituisce i ponti di G.
Input: G grafo indiretto, rappresentato attraverso liste di adiacenza.
Output: i ponti di G.

```
function DFS(G, y, c, back, t, parents)
    c += 1
    t[y] = c
    back[y] = t[y]
    for u ∈ V(G) : (v, u) ∈ E(G) do
        if u ∉ visited then
            DFS(G, u, visited, order)
        end if
    end for
    order.append(v)                         ▷ l'ordinamento risulterà invertito
end function

function findBridges(G)
    v ∈ V(G)                                ▷ un vertice qualsiasi di G
    t := [0] ∗ n
    parents := [0] ∗ n
    parents[v] = v
    back := [0] ∗ n
    c := 0
    DFS(G, v, c, back, t, parents)
    bridges := {}
    for u ∈ V(G) do
        if { back[u] = t[u]
           { u ≠ parents[u]   then
            bridges.add((u, parents[u]))
        end if
    end for
    return bridges
end function
```

Osservazione 1.4.5.1 (Correttezza dell'algoritmo).

Osservazione 1.4.5.2 (Costo dell'algoritmo). Il costo dell'algoritmo è $O(n+m)$, poiché è costituito semplicemente da una visita in DFS ricorsiva del grafo in input, e da un ciclo `for`, alla riga $TODO$, su tutti i nodi del grafo, e dunque si ha $O(n+m)+O(n)=O(n+m)$.