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* 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:

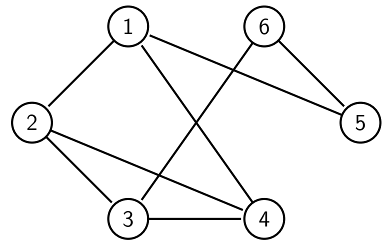

Figura 1.1: Un grafo indiretto.

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:

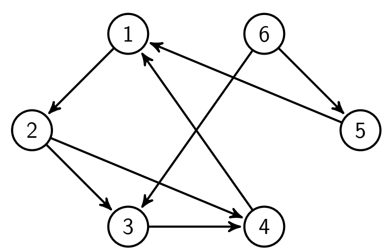

Figura 1.2: Un grafo diretto.

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 $\displaystyle\sum_{v \in V(G)} \deg(v) = 2|E(G)|$. □

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:

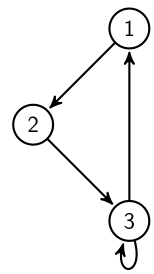

Figura 1.3: Un grafo diretto con cappio in 3.

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:

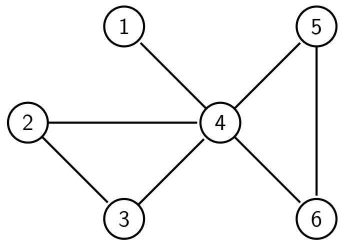

Figura 1.4: Un grafo indiretto.

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.

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:

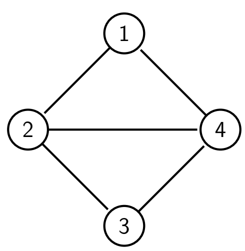

Figura 1.5: Un grafo indiretto.

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:

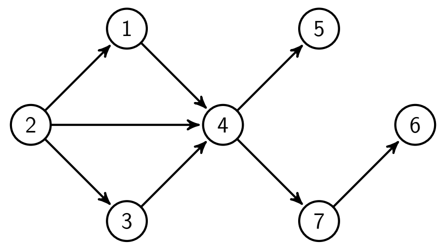

Figura 1.6: Un grafo diretto con ordinamenti topologici.

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 (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*. Si noti che, spesso, il sottografo generato dall'arborescenza di v si identifica con l'arborescenza stessa.

Esempio 1.1.2.4 (Arborescenza)**.** Si consideri il seguente grafo diretto:

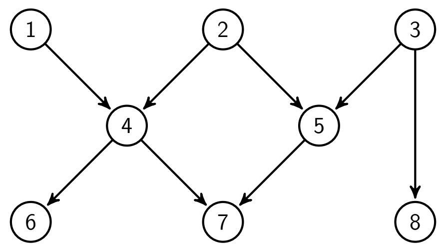

Figura 1.7: Un grafo diretto.

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

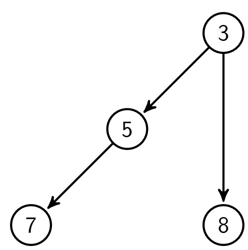

Figura 1.8: Arborescenza di 3.

Definizione 1.1.2.9 (Grafo connesso)**.** Un grafo è detto *connesso* se per ogni $v_1, v_2 \in V(G)$ esiste una passeggiata che li collega.

Esempio 1.1.2.5 (Grafo non connesso)**.** Ad esempio, si consideri questo grafo:

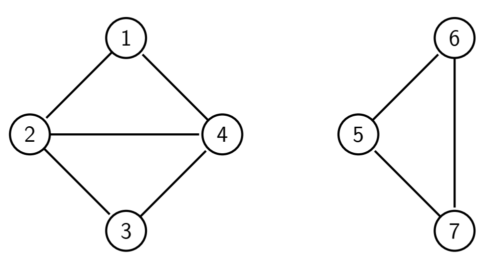

Figura 1.9: Un grafo non connesso.

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

Definizione 1.1.2.10 (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.

Esempio 1.1.2.6 (Grafo fortemente connesso)**.** Un esempio di grafo fortemente connesso è il seguente:

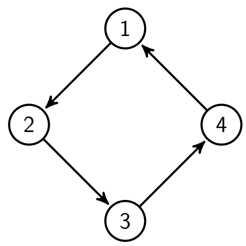

Figura 1.10: Un grafo fortemente connesso.

Definizione 1.1.2.11 (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 1.1.2.7 (Passeggiata euleriana)**.** Ad esempio, si consideri il seguente grafo indiretto:

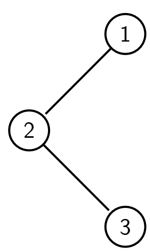

Figura 1.11: Un grafo indiretto.

in esso, l'unica passeggiata euleriana è

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

Teorema 1.1.2.2. *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. TODO

□

Definizione 1.1.2.12 (Passeggiata hamiltoniana)**.** Una passeggiata si dice *hamiltoniana* se TODO

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 (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 $O\left(\sum_{v \in V(G)} \deg(v)\right) = O(2|E(G)|) = O(2m) = O(m)$. 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)**.** 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, il costo per controllare se due vertici sono adiacenti, utilizzando tale rappresentazione, è $O(n)$.

Osservazione 1.2.2.3 (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:

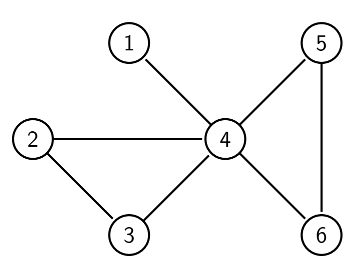

Figura 1.12: Un grafo indiretto.

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 Algoritmi

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 `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**
    `visited.add(`v'`)`
    $v' := v'' \in V(G) : \begin{cases} v' \sim v'' \\ v'' \neq \texttt{visited[visited.length()} - 2\texttt{]} \end{cases}$
  end
  return `visited[visited.indexOf(`v'`):visited.length()]`
end

Dimostrazione. 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 1.3.1.1. Si noti che `visited` contiene tutti i nodi visitati, dunque restituire interamente l'array potrebbe non fornire un ciclo, come nel seguente grafo

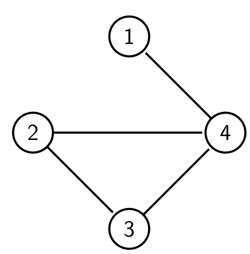

Figura 1.13: Un grafo diretto contenente un ciclo.

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 1.3.1.2 (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, tramite matrice l'algoritmo ha costo $O(n) \cdot O(n) = O(n^2)$, mentre tramite liste si ha $O(1) \cdot O(n) = O(n)$.

1.3.2 DFS (Depth-First Search)

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
13     end
14     return visited
15 end
```

Dimostrazione. Per assurdo, sia $\hat{v} \in V(G)$, raggiungibile da v attraverso cammino, che non sia stato raggiunto dall'algoritmo; allora, per definizione esiste un cammino $v, e_1, v_1, \ldots, v_{n-1}, v_n, \hat{v}$; TODO $\square$

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 `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 (Sottografo di un grafo indiretto)**.** 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.

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 tramite 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 tramite liste di adiacenza; v un vertice di G.

Output: i vertici raggiungibili da v.

```
1  Function findReachableNodes₂(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
14         end
15         if v_top == S.top() then
16             S.pop()
17         end
18     end
19     return visited
20 end
```

Osservazione 1.3.2.5. (Differenze con la prima versione) 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)**.** Si noti che, analogamente alla versione precedente, il ciclo `while` della riga 4 ha costo $O(n)$, poiché nel caso peggiore è necessario inserire e rimuovere dallo stack $2n - 1$ volte i vertici di G. Ma, a differenza del primo algoritmo, il `while` della riga 6 controllerà l'adiacenza di ogni vertice una sola volta, di conseguenza il costo complessivo dell'algoritmo dipende esclusivamente dalla dimensione delle liste di adiacenza, che per il Lemma 1.1.1.1 avrà dimensione $O(m)$. Allora, il costo complessivo equivale al maggiore tra n ed m, e dunque è pari a $O(n)+O(m) = O(n + m)$.

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 corrispondenti 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.

```
Function findTopologicalSorting(G, v)
    order := [v]
    while V(G) ≠ 0 do
        v ∈ V(G) : v.incoming_adjacent().length() = 0
        order.add(v)
        V(G).remove(v)
    end
    return order
end
```

Osservazione 1.3.3.1. (Costo dell'algoritmo) TODO

1.3.4 TODO

Definizione 1.3.4.1 (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 1.3.4.1 (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 1.3.4.1 (Proprietà degli intervalli)**.** *Sia G un grafo diretto, e $u, v \in V(G)$; 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, 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 $T(u) < T(v)$ ↯.

□