Progettazione di Algoritmi

Alessio Bandiera

Informatica, La Sapienza

Indice

Capitolo 1

Elementi di 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:

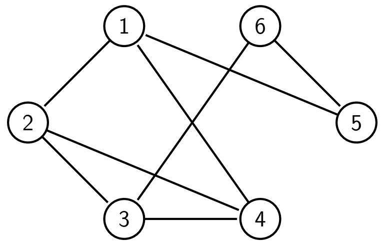


Figura 1.1: Un grafo indiretto.

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:

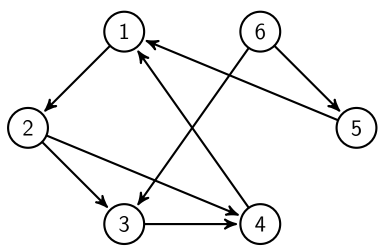


Figura 1.2: Un grafo diretto.

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

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:

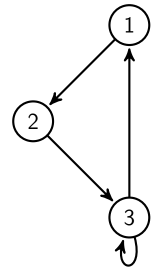


Figura 1.3: Un grafo diretto con cappio in 3.

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.

All'interno di questi appunti, a meno di esplicita specifica, si assume che ogni grafo trattato sia semplice.

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:

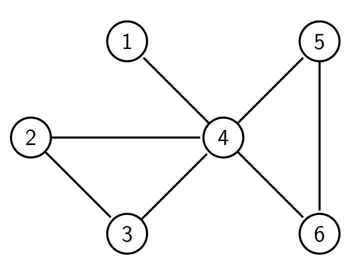


Figura 1.4: Un grafo indiretto.

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 ripercorrere i vertici. In simboli, dati $v, v' \in V(G)$, con $v \to v'$ si indica che è possibile raggiungere v', partendo da 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.

Teorema 1.1.2.1 (Cammini e passeggiate)**.** *Sia G un grafo, e $u, v \in V(G)$ due suoi vertici; allora, in G esiste una passeggiata $u \to v$, se e solo se esiste un cammino $u \to v$.*

Dimostrazione.

Prima implicazione. Sia $u \to v$ una passeggiata da u a v; allora, per trovare il cammino $u \to v$, è sufficiente considerare il sottoinsieme minore, di vertici ed archi, della passeggiata, tale che u e v siano connessi.

Seconda implicazione. Per definizione, una passeggiata è un qualsiasi percorso tra due vertici di un grafo, e in particolare un cammino è una passeggiata, e dunque il cammino $u \to v$ è anche un passeggiata.

□

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:

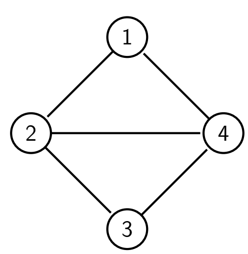


Figura 1.5: Un grafo indiretto.

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:

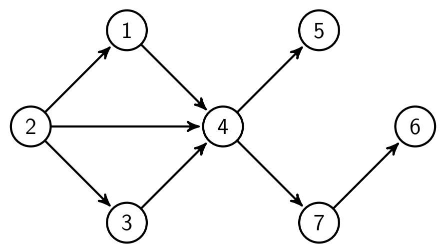


Figura 1.6: Un grafo diretto con ordinamenti topologici.

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

Teorema 1.1.2.2 (Ordinamento topologico)**.** *Un grafo diretto ha un ordinamento topologico, se e solo se è aciclico.*

Dimostrazione.

Prima implicazione. Per assurdo, sia G un grafo diretto, con un ordinamento topologico, e ciclico, avente dunque almeno un ciclo, e siano $\{v_0, \ldots, v_{k-1}, v_0\}$ i vertici che costituiscono uno dei cicli di G; allora, si ha che

$$v_0 \to v_1 \to \ldots \to v_{k-1} \to v_0$$

e dunque all'interno dell'ordinamento topologico v_0 dovrebbe essere posto contemporaneamente prima e dopo $v_1, \ldots, v_{k-1}$ $\lightning$.

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 precedente, 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 r un suo vertice; G è detto *arborescenza* se e solo se, per ogni vertice $v \in V(G) - \{r\}$, esiste uno ed un solo cammino diretto $r \to v$; in tal caso, r prende il nome di *radice.*

Osservazione 1.1.2.3 (Arborescenza)**.** Sia G un grafo diretto, e v un suo vertice; allora, l'insieme degli archi raggiungibili da v formano l'*arborescenza di* v *in* G, 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 in G. Si noti che, spesso, il sottografo generato dall'arborescenza di v viene identificato con l'arborescenza stessa.

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

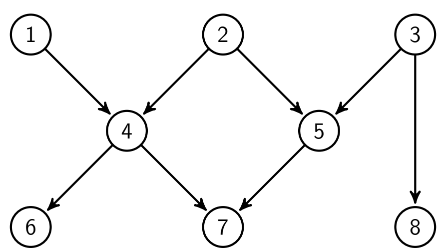


Figura 1.7: Un grafo diretto.

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

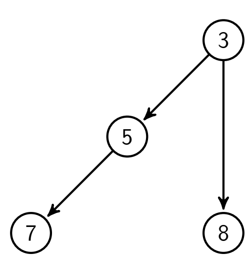


Figura 1.8: Arborescenza di 3.

Definizione 1.1.2.9 (Grafo connesso)**.** Un grafo è detto *connesso* se per ogni $v_1, v_2 \in V(G)$ esiste 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 1.1.2.5 (Grafo non connesso)**.** Ad esempio, si consideri questo grafo:

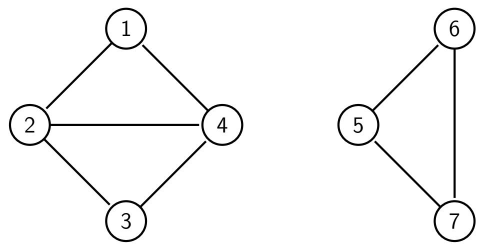


Figura 1.9: Un grafo non connesso.

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

Definizione 1.1.2.10 (Albero)**.** Un grafo indiretto è detto *albero* se è connesso ed aciclico.

Esempio 1.1.2.6 (Albero)**.** Un esempio di albero è il seguente:

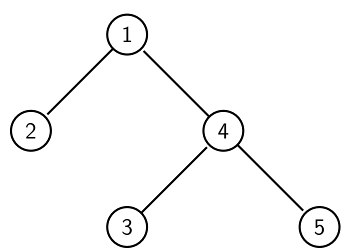


Figura 1.10: Un albero.

Definizione 1.1.2.11 (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 1.1.2.7 (Grafo fortemente connesso)**.** Un esempio di grafo fortemente connesso è il seguente:

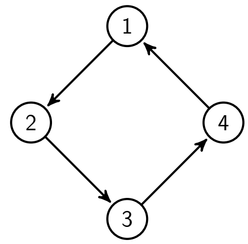


Figura 1.11: Un grafo fortemente connesso.

Lemma 1.1.2.1 (Grafi fortemente connessi)**.** *Sia G un grafo diretto, e $u \in V(G)$ un suo vertice; allora, G è fortemente connesso, se e solo se $\forall v \in V(G) \quad v \to u$ e $u \to v$.*

Dimostrazione.

> *Prima implicazione.* L'implicazione è vera per definizione di grafo diretto fortemente connesso.
>
> *Seconda implicazione.* Siano $x, y \in V(G)$; allora, per ipotesi, esistono cammini $x \to u$, $u \to x$, $y \to u$ e $u \to y$; inoltre, per definizione, tali cammini sono anche passeggiate. In particolare, poiché le passeggiate non hanno vincoli di attraversamento di vertici ed archi, allora è possibile utilizzare la proprietà transitiva, e dunque esistono passeggiate $x \to u \to y \implies x \to y$, e $y \to u \to x \implies y \to x$; allora, per il Teorema 1.1.2.1, esistono anche dei cammini $x \to y$ e $y \to x$. Allora, per definizione, per ogni coppia di vertici esistono due cammini in entrambe le direzioni, e dunque G è fortemente connesso.

□

Definizione 1.1.2.12 (Passeggiata euleriana)**.** Una passeggiata si dice *euleriana* se attraversa ogni arco del grafo, senza ripercorrerne nessuno.

Osservazione 1.1.2.4. Una passeggiata euleriana è una traccia passante per ogni arco del grafo.

Esempio 1.1.2.8 (Passeggiata euleriana)**.** Ad esempio, si consideri il seguente grafo indiretto:

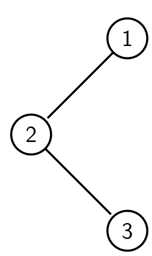


Figura 1.12: Un grafo indiretto.

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.

□

Definizione 1.1.2.13 (Passeggiata hamiltoniana)**.** Una passeggiata si dice *hamiltoniana* se attraversa ogni nodo del grafo, senza ripercorrerne nessuno.

Osservazione 1.1.2.5. 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 $O\left(\sum_{v \in V(G)} \deg(v)\right) = O(2|E(G)|) = O(2m) = O(m)$. 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)**.** 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:

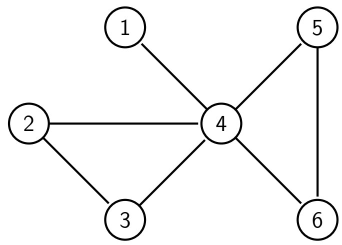


Figura 1.13: Un grafo indiretto.

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 restituisce 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.

1: **function** $\textsc{findCycle}(G)$
2: $\quad v \in V(G)$ ▷ un vertice qualsiasi di G
3: $\quad$ `visited` $:= \{v\}$ ▷ conterrà i vertici visitati
4: $\quad v' \in V(G) : v \sim v'$
5: $\quad$ **while** $v' \notin$ `visited` **do** ▷ tempo costante perché `visited` è un set
6: $\qquad$ `visited.add(`v'`)`
7: $\qquad v' := v'' \in V(G) : \begin{cases} v' \sim v'' \\ v'' \neq \texttt{visited[visited.length()} - \texttt{2]} \end{cases}$
8: $\quad$ **end while**
9: $\quad$ **return** `visited[visited.indexOf(`v'`):visited.length()]`
10: **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 insieme `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'insieme 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'insieme, 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.

Si noti che, nella maggior parte dei linguaggi di programmazione, gli insiemi non hanno ordine, dunque non sarebbe possibile restituire uno slice di `visited`; allora, per semplicità, all'interno dello pseudocodice presentato, si assume si stia utilizzando una struttura dotata di hashing per l'univocità degli elementi, e di ordine per restituirne uno slice, ad esempio un `IndexSet`).

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

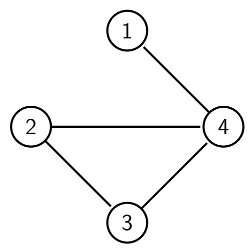


Figura 1.14: Un grafo diretto contenente un ciclo.

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 un 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 $v, e_1, v_1, \ldots, v_{n-1}, v_n, \hat{v}$; 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 `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 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 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)**.** L'algoritmo inizia definendo una lista vuota `order`, all'interno della quale verrà salvato l'ordinamento topologico; successivamente, alla riga 3, viene inizializzato un ciclo `while` che, in ogni iterazione, trova un vertice v il cui numero di vertici adiacenti entranti è 0, e lo inserisce in `order`; questo garantisce che ogni vertice inserito venga necessariamente inserito prima di ogni suo arco uscente, e ne esiste sempre almeno uno grazie al Corollario 1.1.2.1. Si noti inoltre che, poiché G è aciclico, è garantito che rimuovendo un vertice senza archi entranti, il grafo risultante sarà ancora aciclico, ed è possibile dunque ripetere il ragionamento induttivamente per poter dimostrare la correttezza dell'algoritmo.

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 $T(u) < T(v)$ $\lightning$

$\square$

Osservazione 1.4.1.2 (Intervalli disgiunti)**.** Si noti che, avendo un G grafo diretto, e un arco $(u, v) \in E(G)$, dunque con u incidente su v, si ha che

$$\mathrm{Int}(u) \cap \mathrm{Int}(v) = \varnothing \implies t(v) < T(v) < t(u) \leq T(u)$$

e non $t(u) \leq T(u) < t(v) < T(v)$, poiché $T(u) < t(v)$ implicherebbe che la visita in DFS avrebbe sbagliato a rimuovere u dallo stack prima che v potesse essere visitato.

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 $u \sim v$. $\square$

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 $\mathrm{Int}(v)$:

- $\mathrm{Int}(u) \subseteq \mathrm{Int}(v)$, 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
- $\mathrm{Int}(v) \subseteq \mathrm{Int}(u)$, 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
- $\mathrm{Int}(v) \cap \mathrm{Int}(u) = \varnothing$, 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.

$\square$

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.

```
1:  function DFS(G, v, visited, c, t, T)
2:      for u ∈ V(G) : (v, u) ∈ E(G) do                ▷ u deve essere uscente da v
3:          if u ∉ visited then
4:              c.increment()
5:              t[u] = c
6:              visited.add(u)
7:              DFS(G, u, visited, c, t, T)
8:          end if
9:      end for
10:     T[v] = c
11: end function
12:
13: function findTimes(G, r)
14:     visited := {r}
15:     t := [0] ∗ n
16:     T := [0] ∗ n
17:     t[r] = 1
18:     Counter c := 1                   ▷ questo contatore deve essere un oggetto
19:     DFS(G, r, visited, c, t, T)
20:     return t, T
21: end function
```

Osservazione 1.4.2.3 (Correttezza dell'algoritmo)**.** L'algoritmo inizia salvando la radice r all'interno di un insieme `visited`, ed inizializzando gli array `t` e `T` con 0; inoltre, il tempo di visita di r viene inizializzato a 1, alla riga 17; infine, viene istanziato un contatore, partendo da 1 (poiché la radice è già stata inizializzata).

All'interno della funzione ricorsiva, per ogni livello della ricorsione, viene esplorato ogni vertice adiacente a v in ingresso; in particolare, se gia non visitato, viene scelto un u tale che $u \sim v$. Successivamente, viene aggiornato il contatore, e viene salvato il tempo di visita di u; infine, viene aggiunto il vertice a `visited`. Alla riga 7, la funzione ricorsiva viene eseguita nuovamente, utilizzando come nuovo nodo di partenza u. Infine, per ogni livello di ricorsione, dopo aver terminato i vertici adiacenti, il ciclo `for` della riga 2 termina, e alla riga 10 viene aggiornato il tempo di chiusura del vertice v.

Allora, il codice è in grado di visitare il grafo intermaente, senza ripercorrere vertici già visitati, utilizzando una visita in DFS, poiché vengono esplorati tutti i vertici adiacenti ricorsivamente, prima di tornare al vertice precedente.

Si noti che l'algoritmo funziona correttamente, solamente se `c` è un oggetto e non una variabile; infatti, il contatore si deve comportare come se fosse globale per ogni livello di ricorsione, altrimenti i tempi sarebbero tutti errati; in particolare, senza trattare il contatore come oggetto, il contatore, ritornando indietro con i livelli ricorsivi, decrementerebbe.

Osservazione 1.4.2.4 (Costo dell'algoritmo)**.** Si noti che l'algoritmo controlla ogni singolo vertice una ed una sola volta, e la vista avviene in DFS; inoltre, poiché si ha un ciclo `for` all'interno di ogni livello della ricorsione, per costruzione stessa, l'algoritmo si comporta esattamente come l'Algoritmo 1.3.2.2. Allora, poiché il grafo è rappresentato tramite liste di adiacenza, per ragionamento analogo all'Osservazione 1.3.2.6, il costo dell'algoritmo è pari a $O(n + m)$.

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
```

```
30:     forward := []
31:     backward := []
32:     cross := []
33:     for v ∈ V(G) do
34:         for u ∈ v.incoming_adjacent() do
35:             if parents[v] == u then
36:                 continue                    ▷ faceva parte dell'arborescenza di v
37:             else if T[u] < t[v] or T[v] < t[u] then
38:                 cross.append((u, v))
39:             else if T[u] ≤ T[v] then
40:                 backward.append((u, v))
41:             else
42:                 forward.append((u, v))
43:             end if
44:         end for
45:     end for
46:     return forward, backward, cross
47: end function
```

Osservazione 1.4.2.5 (Correttezza dell'algoritmo)**.** TODO

Osservazione 1.4.2.6 (Costo dell'algoritmo)**.** Si noti che il ciclo `while` della riga 11, e termina alla riga 29, ha lo stesso costo computazionale dell'Algoritmo 1.3.2.2, e dunque il suo costo è pari a $O(n + m)$.

Si noti inoltre che, il ciclo `for` della riga 33, effettua un'iterazione per ogni singolo vertice del grafo, ma all'interno di esso è presente un ulteriore ciclo `for`, alla riga 34, che itera sui rispettivi vertici adiacenti entranti; allora, il costo di questi due cicli equivale al solo spazio di rappresentazione delle liste di adiacenza di G, ovvero $O(n + m)$, poiché al loro interno vengono eseguite esclusivamente operazioni in tempo costante.

In conclusione, il costo computazionale è pari a $O(n + m) + O(n + m) = O(n + m)$.

Algoritmo 1.4.2.3 Dato un'array di padri `parents`, che rappresenta un'arborescenza di visita in DFS di un grafo diretto, e un arco (x, y) del grafo, l'algoritmo restituisce il tipo di arco.
Input: `parents` array di padri di un'arborescenza di visita in DFS di un grafo diretto; (x, y) un arco del grafo.
Output: la categoria di (x, y).

```
function CATEGORIZEEDGE(parents, (x, y))
    if parents[y] == x then
        return NodeType::Arborescence
    end if
    z := y
    while parents[z] ≠ z do
        z = parents[z]
        if z == x then
            return NodeType::Forward
        end if
    end while
    while parents[z] ≠ z do
        z = parents[z]
        if z == y then
            return NodeType::Backward
        end if
    end while
    return NodeType::Cross
end function
```

Osservazione 1.4.2.7 (Correttezza dell'algoritmo)**.** TODO

Osservazione 1.4.2.8 (Costo dell'algoritmo)**.** TODO O(n)

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 $u, v \in V(A_{\hat{v}})$, 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.

```
function DFS(G, v, visited, order)
    visited.add(v)
    for u ∈ V(G) : (v, u) ∈ E(G) do                 ▷ u deve essere uscente da v
        if u ∉ visited then
            DFS(G, u, visited, order)
        end if
    end for
    order.append(v)                                 ▷ l'ordinamento risulterà invertito
end function

function findTopologicalSortingDFS(G)
    order := []
    visited := {}
    for v ∈ V(G) do
        if v ∉ visited then
            DFS(G, v, visited, order)
        end if
    end for
    order.reverse()                                 ▷ viene invertita la lista
    return order
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

Definizione 1.4.4.1 (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 1.4.4.1 (Pozzo universale)**.** Ad esempio, il seguente grafo diretto presenta un pozzo universale in 3:

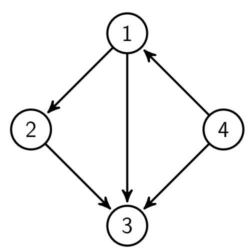


Figura 1.15: Un grafo con pozzo universale in 3.

Teorema 1.4.4.1 (Unicità del pozzo universale)**.** *Sia G un grafo diretto, e $p \in V(G)$ un suo pozzo universale; allora, tale pozzo universale 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' $\lightning$. □

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.

```
1:  function findUniversalSink(M_G)
2:      p ∈ V(G)                    ▷ un vertice qualsiasi, possibile pozzo universale
3:      for v ∈ V(G) do
4:          if M_G[p, v] == 1 then
5:              p = v
6:          end if
7:      end for
8:      for v ∈ V(G) − {p} do
9:          if M_G[p, v] == 1 then
10:             return None
11:         end if
12:         if M_G[v, p] == 0 then
13:             return None
14:         end if
15:     end for
16:     return p
17: 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:

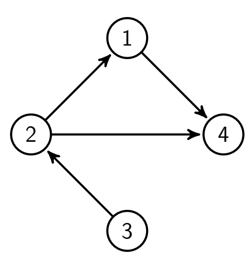


Figura 1.16: Un grafo che contiene un possibile pozzo universale.

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

Definizione 1.4.5.1 (Ponte)**.** Sia G un grafo, e sia $(u, v) \in E(G)$ un suo arco; allora, (u, v) è detto *ponte* se e solo se non è contenuto in nessun ciclo di G.

Esempio 1.4.5.1 (Ponte)**.** Un esempio di grafo che presenta un ponte è il seguente:

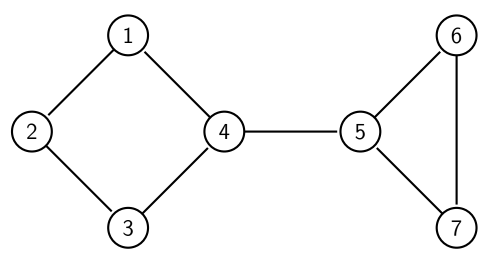


Figura 1.17: Un grafo con un ponte.

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

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

$$\{5, (5,6), 6, (6,7), 7, (7,5), 5\}$$

Teorema 1.4.5.1 (Presenza di ponti)**.** *Sia G un grafo indiretto connesso, $r \in V(G)$, e sia T_r l'albero di visita in DFS di r in G; sia $x \in V(T_r)$, e sia $T_x \subseteq T_r$ il suo sottoalbero in T_r; sia inoltre $y \in V(T_x)$ tale che $(x, y) \in E(G)$, e sia $T_y \subseteq T_x$ il suo sottoalbero in T_x; allora, esiste un arco $(u, v) \in E(G)$, tale che $u \in V(T_y)$ e $v \in V(T - T_x)$, se e solo se esiste un ciclo in G contenente (x, y).*

Dimostrazione.

Prima implicazione. Si noti che, poiché G è indiretto e connesso, per qualsiasi $r \in V(G)$, $V(T_r) = V(G)$, dunque la visita in DFS è sempre in grado di raggiungere ogni nodo del grafo G. Per definizione T_y è connesso ed aciclico, e dunque esiste uno ed un solo cammino $y \to u$; inoltre, per ragionamento analogo, esiste uno ed un solo cammino $v \to x$. Allora, se esiste un arco $(u, v) \in E(G)$, viene creato un ciclo in G della forma $v \to x \to y \to u \to v$, poiché esiste l'arco $(x, y) \in E(G)$ in ipotesi, che sarà dunque necessariamente contenuto in tale ciclo.

Seconda implicazione. Se (x, y) è incluso in almeno un ciclo di G, deve necessariamente esistere un cammino $x \to y$ non passante per (x, y), il quale deve contenere un arco (u, v), in cui $u \in V(T_y)$, e $v \in V(T - T_x)$, poiché $(x, y) \in E(G)$ è un arco.

□

Lemma 1.4.5.1 (Alberi con ponti)**.** *Sia G un grafo indiretto, e $(u, v) \in E(G)$ un suo ponte; allora, per ogni possibile T_u, albero di visita di G in DFS, radicato in u, si ha che $(u, v) \in E(T_u)$.*

Dimostrazione. Poiché (u, v) è un ponte, per definizione non appartiente a nessun ciclo di G, allora preso un qualsiasi T_u, albero di visita di G in DFS, radicato in u, l'unico modo per raggiungere v è attraverso (u, v) stesso, in quanto G è indiretto, e dunque necessariamente $(u, v) \in E(T_u)$ □

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.increment()
    t[y] = c
    back[y] = t[y]
    for z ∈ V(G) : z ∼ y do
        if t[z] == 0 then                          ▷ z non deve essere già stato visitato
            parents[z] = y
            DFS(G, z, c, back, t, parents)
            if back[z] < back[y] then
                back[y] = back[z]
            end if
        else if { z ≠ parents[y]
                { t[z] < back[y]    then
            back[y] = back[z]
        end if
    end for
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
    Counter c := 0                                 ▷ è un oggetto
    DFS(G, v, c, back, t, parents)
    bridges := {}
    for u ∈ V(G) do
        if { back[u] = t[u]
           { u ≠ parents[u]    then
            bridges.add((parents[u], u))
        end if
    end for
    return bridges
end function
```

Osservazione 1.4.5.1 (Correttezza dell'algoritmo)**.** L'algoritmo inizia scegliendo un vertice v casuale del grafo G in input; successivamente, vengono definiti gli array `t`, all'interno del quale verranno salvati i tempi di visita dei vertici, `parents`, utilizzato per salvare i nodi padri di ogni vertice visitato, e `back`.

Sia T l'albero di visita in DFS della visita correntemente in esecuzione dall'algoritmo; allora, per un certo vertice $y \in V(T)$, all'interno di $\texttt{back}[y]$, verrà salvato il tempo di visita del vertice $v \in V(T - T_y)$ più lontano, raggiungibile da un discendente z di y (incluso y stesso), attraverso un cammino passante per un arco $(z, v) \notin E(T)$. L'array `back` verrà sfruttato congiuntamente al Teorema 1.4.5.1, in quanto l'arco $(\texttt{parents}[y], y)$ è un ponte, se e solo se non esiste un tale arco (z, v).

Infine, prima di iniziare una visita in DFS ricorsiva dell'albero, viene istanziato un oggetto contatore `c`, utilizzato per salvare i tempi di visita in `t`.

L'esecuzione procede all'interno della funzione ricorsiva `DFS`, che ha lo scopo di popolare i valori degli array precedentemente definiti. La funzione inizia aggiornando il contatore, e inserendo il relativo valore del tempo di visita del vertice corrente; inoltre, di quest'ultimo viene inizializzato il valore corrisopndente in `back`, pari al suo stesso tempo di visita. Tale valore servirà come *sentinel value*, e verrà utilizzato successivamente dall'algoritmo per stabilire quali archi del grafo sono ponti.

Successivamente, viene istanziato un ciclo `for`, alla riga 5, in cui per ogni vertice z, adiacente al vertice y corrente:

- se tale vertice z ha tempo di visita pari a 0, e dunque non è ancora stato visitato, ne viene inizialmente aggiornato il valore del padre, alla riga 7, che sarà proprio y, e viene poi effettuata una chiamata ricorsiva, radicata in z; al termine di quest'ultima, viene aggiornato il valore di $\texttt{back}[y] = \min(\texttt{back}[y], \texttt{back}[z])$, e dunque sarà sempre il minore tra i suoi discendenti;

- se invece tale vertice z è già stato visitato, va aggiornato il valore di $\texttt{back}[y]$ con $\min(\texttt{t}[z], \texttt{back}[y])$, esclusivamente se z non è il padre di y.

Al termine della visita in DFS ricorsiva, l'algoritmo conclude cercando tutti i ponti del grafo, a partire dalle informazioni all'interno di `back`: infatti, alla riga 27, per ogni vertice $u \in V(G)$ del grafo, viene inserito all'interno dell'insieme `bridges` l'arco tra u e il suo vertice padre, se e solo se $u \neq \texttt{parents}[u]$ (ovvero, u non è la radice), e $\texttt{back}[u] = \texttt{t}[u]$, dunque il valore di $\texttt{back}[u]$ è rimasto invariato dalla riga 4. L'algoritmo termina restituendo `bridges`, che conterrà l'insieme dei ponti di G.

Ad esempio, si consideri il seguente grafo:

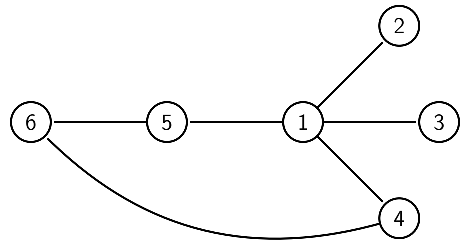


Figura 1.18: Un grafo indiretto.

inoltre, sia

$$T_6 = (\{6, 5, 1, 2, 3, 4\}, \{(6, 5), (5, 1), (1, 2), (1, 3), (1, 4)\})$$

l'albero di visita in DFS del grafo, partendo dal vertice 6; allora, si avranno

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

$$\texttt{parents} = [5, 1, 1, 1, 6, 6]$$

$$\texttt{back} = [1, 4, 5, 1, 1, 1]$$

allora, al termine del ciclo `for` della riga 27, si avrà

$$\texttt{bridges} = \{(2, \texttt{parents[2]}), (3, \texttt{parents[3]})\}$$

ovvero

$$\texttt{bridges} = \{(2, 1), (3, 1)\}$$

poiché `back[2]` $=$ `t[2]` $= 4$ e `back[3]` $=$ `t[3]` $= 5$; infine, si noti che `back[6]` $=$ `back[6]` $= 1$, ma $6 \notin$ `bridges` poiché `parents[6]` $= 6$, e infatti T_6 è proprio radicato in 6.

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 27, su tutti i nodi del grafo, e dunque si ha $O(n + m) + O(n) = O(n + m)$.

1.4.6 Trovare le componenti

Definizione 1.4.6.1 (Componenti indirette)**.** Sia G un grafo indiretto, non necessariamente connesso; si definisce *componente* un sottografo di G, connesso, non ulteriormente estendibile. Più rigorosamente, sia $H \subseteq G$ un sottografo di G; esso è una componente di G, se e solo se non esiste $H' \subseteq G$, sottografo connesso di G, tale che $H \subsetneq H'$. Le componenti sono anche definite come sottografi *massimalmente connessi*.

In simboli, dato un vertice $v \in V(G)$, $\text{comp}(v)$ è la componente contenente v.

Esempio 1.4.6.1 (Componenti indirette)**.** Ad esempio, si consideri questo grafo:

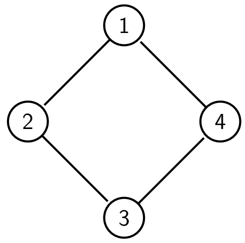


Figura 1.19: Un grafo indiretto.

esso, presenta 2 componenti:

$$H_1 := (\{1, 2, 3, 4\}, \{(1, 2), (2, 3), (3, 4), (4, 1)\})$$

$$H_2 := (\{5, 6, 7\}, \{(5, 6), (6, 7), (7, 5)\})$$

Definizione 1.4.6.2 (Componenti dirette)**.** Sia G un grafo diretto, non necessariamente connesso; allora, si definisce *componente* un sottografo di G, fortemente e massimalmente connesso.

Esempio 1.4.6.2 (Componenti dirette)**.** Ad esempio, si consideri questo grafo:

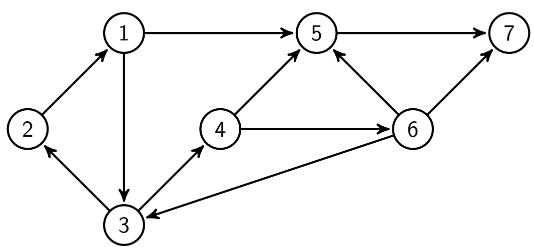


Figura 1.20: Un grafo diretto.

esso, presenta 3 componenti:

$$H_1 := (\{1, 2, 3, 4, 6\}, \{(1, 3), (3, 4), (4, 6), (6, 3), (3, 2), (2, 1)\})$$

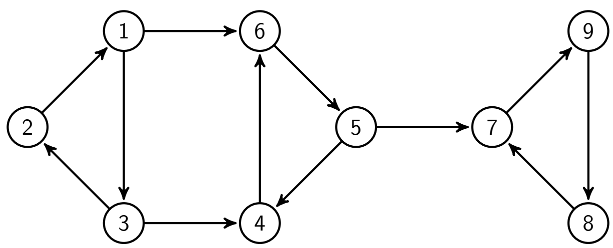
$$H_2 := (\{5\}, \varnothing)$$
$$H_3 := (\{7\}, \varnothing)$$

Lemma 1.4.6.1 (Digiunzione delle componenti)**.** *Sia G un grafo; allora, le sue componenti sono disgiunte.*

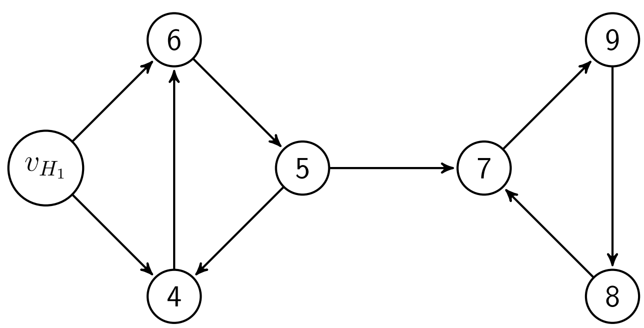
Dimostrazione. Per assurdo, sia G un grafo, diretto o indiretto, e $H_1, H_2 \subseteq G$ due sue componenti tali che $H_1 \cap H_2 \neq \varnothing$; se tali componenti esistessero, allora esisterebbe almeno un vertice nella loro intersezione, raggiungibile da entrambe le componenti (nel caso di G diretto, i cammini sarebbero in entrambe le direzioni, poiché tale vertice sarebbe parte sia di H_1 che di H_2, entrambe fortemente connesse per definizione); allora, H_1 e H_2 non sarebbero massimalmente connesse $\lightning$. $\square$

Definizione 1.4.6.3 (Contrazione)**.** Sia G un grafo, e $H \subseteq G$ un suo sottografo; si definisce *contrazione di H in G*, l'operazione che rimuove vertici (ed archi) di H da G, e al suo posto inserisce un vertice, generalmente denotato con v_H, che viene connesso con $G - H$ dagli archi che precedentemente connettevano H con G. In simboli, il grafo G, contratto su H, verrà indicato con $\operatorname{contr}(G, H)$.

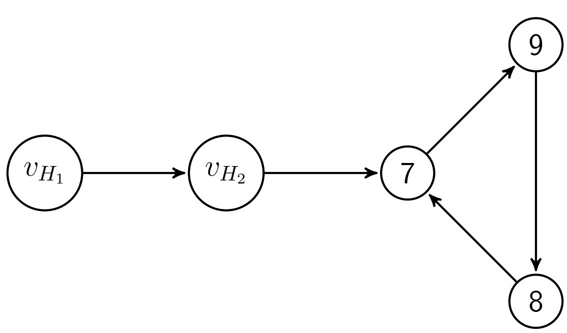
Esempio 1.4.6.3 (Contrazione di un grafo diretto)**.** Si consideri il seguente grafo diretto:



contraendo la componente $H_1 = (\{1, 2, 3\}, \{(1, 3), (3, 2), (2, 1)\})$ in v_{H_1}, si ottiene:



contraendo la componente $H_2 = (\{4, 5, 6\}, \{(4, 6), (6, 5), (5, 4)\})$ in v_{H_2}, si ottiene:



infine, contraendo la componente $H_3 = (\{7,8,9\}, \{(7,9),(9,8),(8,7)\})$ in v_{H_3}, si ottiene il seguente grafo massimalmente contratto:

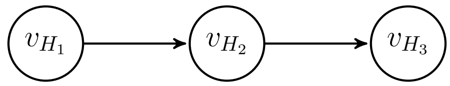


Figura 1.21: Il grafo iniziale massimalmente contratto.

Teorema 1.4.6.1 (Contrazioni fortemente connesse)**.** *Sia G un grafo diretto fortemente connesso, e $H \subseteq G$ un suo sottografo fortemente connesso; allora,* $\mathrm{contr}(G, H)$ *è ancora fortemente connesso.*

Dimostrazione. Sia v_H il vertice del grafo contratto $\mathrm{contr}(G, H)$, e $y \in V(G - H)$; poiché G è fortemente connesso, devono necessariamente esistere due vertici $v', v'' \in V(H)$ tali che $y \to v'$ e $v'' \to y$; siano i vertici v' e v'' per i quali tali cammini siano di minor lunghezza possibile. Allora, effettuando la contrazione di H, in $\mathrm{contr}(G, H)$ sarà necessario rimpiazzare gli archi che permettevano tali cammini, e dunque verranno inseriti due nuovi archi, uno entrante uno uscente, verso v_H. Allora, necessariamente $\mathrm{contr}(G, H)$ è ancora fortemente connesso. □

Teorema 1.4.6.2 (Presenza di cicli)**.** *Sia G un grafo diretto fortemente connesso, non composto da un singolo vertice; allora, esso presenta almeno un ciclo.*

Dimostrazione. Per definizione $\forall u, v \in V(G) \quad u \to v$ e $v \to u$; allora, presi due vertici $u, v \in V(G)$, in G deve necessariamente esistere un ciclo della forma $u \to v \to u$. □

Algoritmo 1.4.6.1 Dato un grafo diretto G, rappresentato attraverso liste di adiacenza, con liste di archi sia entranti che uscenti per ogni vertice, l'algoritmo restituisce le componenti di G.
Input: G grafo diretto, rappresentato attraverso liste di adiacenza, con liste di archi sia entranti che uscenti per ogni vertice.
Output: le componenti di G.

```
function findCycle(G)
    TODO deve funzionare in O(n + m)
end function

function findComponents_1(G)
    C := findCycle(G)
    if C == None then
        return {{v} : v ∈ V(G)}            ▷ le componenti sono i singoli vertici
    else
        v_C, G = contr(G, C)               ▷ v_C è il vertice della contrazione
        {H_1, ..., H_k} = findComponents(G)
        new_components := {}               ▷ conterrà le nuove componenti
        for i ∈ [1, k] do
            if v_C ∉ H_i then
                new_components.add(H_i)
            else
                new_components.add((H_i − {v_C}) ∪ V(C))
            end if
        end for
    end if
    return new_components
end function
```

Osservazione 1.4.6.1 (Correttezza dell'algoritmo)**.** Sia G un grafo diretto, e $H \subseteq G$ una sua componente; allora, se H non è composto da un singolo vertice, poiché è fortemente connesso per definizione, per il Teorema 1.4.6.2, in H è presente un ciclo C; allora, contraendo tale ciclo, si otterrà un sottografo $\mathrm{contr}(H, C)$, ancora fortemente connesso, per il Teorema 1.4.6.1. Allora, questo garantisce di poter trovare tutte le componenti di un grafo, contraendone ricorsivamente i cicli.

TODO parla della funzione che trova i cicli in O(n + m)

L'algoritmo inizia cercando un ciclo C all'interno del grafo G; se non ne viene trovato alcuno, viene raggiunto il caso base della ricorsione, e l'algoritmo restituisce un insieme di insiemi dei singoli vertici di G; si noti che tale caso si presenta esclusivamente quando il grafo è massimalmente contratto.

Differentemente, se è presente un ciclo C all'interno del grafo, l'algoritmo procede contraendo C in v_C, e rimpiazzando G, alla riga 10; successivamente, viene effettuata una chiamata ricorsiva sul grafo contratto, al termine della quale viene restituito l'insieme di componenti di G corrente.

Dopo aver contratto ricorsivamente ogni possibile ciclo, l'algoritmo istanzia un loop `for` alla riga 13, all'interno del quale viene trovata la componente H_i in cui v_C faceva parte: le componenti che non contengono v_C vengono inserite all'interno dell'insieme `new_components`, senza subire modifiche; al contrario, dall'unica componente H_i che contiene v_C, viene rimosso v_C stesso, e ad essa verrà aggiunto tutto il ciclo C che era stato contratto, ripristinando dunque il grafo di partenza di ricorsione in ricorsione.

Osservazione 1.4.6.2 (Costo dell'algoritmo)**.** TODO O(n(n + m))

Definizione 1.4.6.4 (C-radici)**.** Sia G un grafo diretto, sia $\hat{v} \in V(G)$ un suo vertice, e sia $A_{\hat{v}}$ l'arborescenza di $\hat{v}$ in G; $v \in A_{\hat{v}}$ è detto *c-radice* di $\text{comp}(v)$ in $A_{\hat{v}}$, se e solo se è il primo vertice visitato in $\text{comp}(v)$.

Esempio 1.4.6.4 (C-radici)**.** Si consideri il seguente G grafo diretto:

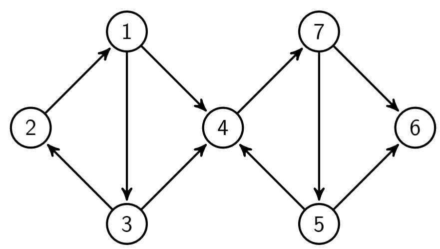


Figura 1.22: Un grafo diretto.

esso presenta 3 componenti:

$$H_1 = (\{1, 2, 3\}, \{(1, 3), (3, 2), (2, 1)\})$$
$$H_2 = (\{4, 5, 7\}, \{(4, 7), (7, 5), (5, 4)\})$$
$$H_3 = (\{6\}, \varnothing)$$

sia $\hat{v} = 2$, dunque la sua arborescenza $A_{\hat{v}}$ è la seguente:

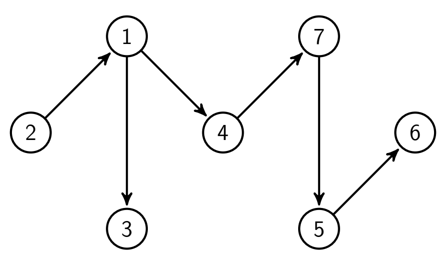


Figura 1.23: Arborescenza di 2.

allora, in H_1 il primo vertice visitato da $A_{\hat{v}}$ è 2, in H_2 è 4, e in H_3 è 6; allora le c-radici di $A_{\hat{v}}$ sono $\{2, 4, 6\}$.

Teorema 1.4.6.3 (C-radici)**.** *Sia G un grafo diretto, sia $\hat{v} \in V(G)$ un suo vertice, e sia $A_{\hat{v}}$ l'arborescenza di $\hat{v}$ in G; sia u la c-radice di* $\mathrm{comp}(u)$ *in $A_{\hat{v}}$; allora si verificano le seguenti proposizioni:*

i) *sia $A_u \subseteq A_{\hat{v}}$ è l'arborescenza radicata in u; allora*

$$V(\mathrm{comp}(u)) \subseteq V(A_u)$$

ii) *siano $u_1, \ldots, u_k$ le c-radici di* $\mathrm{comp}(u_1), \ldots, \mathrm{comp}(u_k)$ *in $A_u \subseteq A_{\hat{v}}$; allora*

$$V(\mathrm{comp}(u)) \cup \bigcup_{i=1}^{k} V(\mathrm{comp}(u_i)) = V(A_u)$$

Dimostrazione.

i) Per definizione, $\mathrm{comp}(u)$ è fortemente connesso, e dunque per ogni $v \in \mathrm{comp}(u)$ esistono due cammini $u \to v$ e $v \to u$; allora, partendo da u, necessariamente $v \in V(A_{\hat{v}})$, poiché la visita in DFS di $\hat{v}$ deve aver raggiunto v. Infine, v non potrebbe essere stato visitato prima di u, poiché u è stato scelto come c-radice di $\mathrm{comp}(u)$; allora necessariamente $v \in V(A_u)$, e dunque $V(\mathrm{comp}(u)) \subseteq V(A_u)$.

ii) Si noti che $u_1, \ldots, u_k \in V(A_u)$, e dunque per definizione

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

Inoltre, per la proposizione precedente, si ha che

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

e anche che

$$V(\mathrm{comp}(u)) \subseteq V(A_u)$$

Allora, necessariamente

$$V(\mathrm{comp}(u)) \cup \bigcup_{i=1}^{k} V(\mathrm{comp}(u_i)) \subseteq V(A_u)$$

Sia $w \in V(A_u)$, e dunque esiste un cammino $u \to w$:

- se esiste un cammino $w \to u$, poiché $\text{comp}(u)$ è massimalmente fortemente connesso per definizione, necessariamente $w \in \text{comp}(u)$; allora $V(A_u) \subseteq \text{comp}(u)$, e dunque segue la tesi;
- allora, si supponga non esista un tale cammino $w \to u$, e dunque $\text{comp}(u) \neq \text{comp}(w)$, per il Lemma 1.4.6.1; allora, sia $z \in V(\text{comp}(w))$ la c-radice di $\text{comp}(w) = \text{comp}(z)$; si noti che $z \in \text{comp}(w)$, implica che è presente in G un cammino $w \to z$, e dunque si ha un cammino $u \to w \to z$ in G; allora, la visita in DFS di $\hat{v}$ deve aver necessariamente raggiunto z, e dunque $z \in V(A_{\hat{v}})$;
- per assurdo, sia $z \notin V(A_u)$; poiché in G si ha un cammino $u \to w \to z$ per osservazione precedente, allora segue che $t(z) < t(u)$, e dunque z deve essere stato visitato prima di u, altrimenti sarebbe stato raggiunto a partire da u; allora:
  - sia $\text{Int}(z) \cap \text{Int}(u) = \varnothing$: si noti che $z \in \text{comp}(w)$ è c-radice di $\text{comp}(w)$, e dunque si ha il cammino $z \to w$; allora, poiché $\text{Int}(w) \subseteq \text{Int}(u)$, la visita in DFS avrebbe sbagliato a non controllare w prima di rimuovere z dallo stack $\lightning$
  - sia $\text{Int}(z) \supset \text{Int}(u) \supseteq \text{Int}(w)$: allora si avrebbe che $A_u \subset A_z$, e dunque esiste un cammino $z \to u$, ma per il cammino $u \to w \to z$, allora si avrebbe $\text{comp}(z) = \text{comp}(u)$, contraddicendo l'ipotesi per cui $\text{comp}(u) \neq \text{comp}(w) = \text{comp}(z)$ $\lightning$
- allora necessariamente $z \in V(A_u)$, e poiché è stato scelto come c-radice di una delle componenti di G in A_u, allora deve verificarsi che $z \in \{u_1, \ldots, u_k\}$;
- dunque $\exists i \in [1, k] \mid z = u_i$, e poiché $\forall i \in [1, k] \quad u_i \in V(\text{comp}(u_i))$, allora
$$z \in V(A_u) \implies \exists i \in [1, k] \mid z = u_i \in V(\text{comp}(u_i))$$
dunque $V(A_u) \subseteq V(\text{comp}(u_i))$ per qualche $i \in [1, k]$, e quindi segue la tesi.

$\square$

Algoritmo 1.4.6.2 Dato un grafo diretto G, rappresentato attraverso liste di adiacenza, l'algoritmo restituisce le componenti di G.
Input: G grafo diretto, rappresentato attraverso liste di adiacenza.
Output: le componenti di G.

```
function DFS(G, u, components, S, c, cc)
    c.increment()
    components = −c                    ▷ i valori negativi indicano i tempi di visita
    S.push(u)
    b := c
    for v ∈ V(G) : v ∼ u do
        if components[v] == 0 then
            b' := DFS(G, u, components, S, c, cc)
            b = min(b, b')
        else if components[v] < 0 then
            b = min(b, −components[v])
        end if
    end for
    if b < −components[u] then                                    ▷ u è c-radice
        cc.increment()
        do
            w := S.pop()
            components[w] = cc
        while u ≠ w
    end if
end function

function findComponents_2(G)
    components := [0] ∗ n
    Counter cc := 0
    Counter c := 0
    Stack s := []
    for u ∈ V(G) do
        if components[u] == 0 then
            DFS(G, u, components, S, c, cc)
        end if
    end for
    return components
end function
```

Osservazione 1.4.6.3 (Correttezza dell'algoritmo)**.** Sia G un grafo diretto, sia

$\hat{v} \in V(G)$ un suo vertice, e sia $A_{\hat{v}}$ l'arborescenza di $\hat{v}$ in G; allora, un vertice $u \in V(G)$ è una c-radice in $A_{\hat{v}}$, se e solo se TODO

Osservazione 1.4.6.4 (Costo dell'algoritmo)**.** TODO $O(n + m)$

Algoritmo 1.4.6.3 Dato un grafo indiretto G, rappresentato attraverso liste di adiacenza, l'algoritmo restituisce un suo ciclo, se presente.
Input: G grafo indiretto, rappresentato attraverso liste di adiacenza.
Output: un ciclo di G, se presente.

```
function DFS(G, u, parents, visited)
    visited[u] = 1
    for v ∈ V(G) : v ∼ u do
        if visited[v] == 0 then
            visited[v] = 1
            parents[v] = u
            output = DFS(G, v, parents, visited)
            if output ≠ None then
                return output
            end if
        else if { parents[u] ≠ v
                { parents[v] ≠ u   then                ▷ l'array parents è diretto
            output := {}
            while u ≠ v do
                output.add(u)
                u = parents[u]
            end while
            return output
        end if
    end for
    return None
end function

function findCycle(G)
    visited := [0] * n
    for v ∈ V(G) do
        if visited[v] == 0 then
            output = DFS(G, v, parents, visited)
            if output ≠ None then
                return output
            end if
        end if
    end for
    return None
end function
```

Osservazione 1.4.6.5 (Correttezza dell'algoritmo)**.** TODO DA RILEGGERE

L'algoritmo inizia effettuando un ciclo `for`, alla riga 25, su ogni vertice $v \in V(G)$; in particolare, se v non è stato ancora visitato (riga 26), viene effettuata una visita ricorsiva in DFS di G, partendo da v.

La funzione che effettua la visita ricorsiva in DFS, inizia con un ciclo `for`, alla riga 3, il quale, per ogni vertice $v \in V(G)$ adiacente a u in input, se non visitato, viene marcato come visitato (riga 5), ne viene aggiornato il nodo padre (riga 6), e viene effettuata una chiamata ricorsiva radicata su esso, per far si che la visita sia in DFS.

Al termine della chiamata ricorsiva, viene esaminato il valore ritornato da `DFS`, che può essere un ciclo di G, oppure `None`; se si è giunti al primo caso, e dunque si è trovato l'output richiesto dall'algoritmo, sarà sufficiente riportare il risultato di ricorsione in ricorsione a ritroso, ritornando `output` stesso, alla riga 9.

Se invece il vertice v, in `DFS` TODO

Osservazione 1.4.6.6 (Costo dell'algoritmo)**.** Poiché l'algoritmo effettua una visita ricorsiva in DFS del grafo G in input, il costo dovrebbe essere $O(n+m)$, ma l'unico caso in cui viene effettuata una visita completa del grafo è nel caso peggiore, ovvero quando il grafo è aciclico. Si noti però che, se G è aciclico, si verifica che $m := |E(G)| = n-1$, e dunque $O(n+m) = O(n+n-1) = O(2n-1) = O(n)$. Allora, il costo dell'algoritmo è $O(n)$.

1.5 BFS (Breadth-first Search)

1.5.1 Distanza

Definizione 1.5.1.1 (Distanza)**.** Sia G un grafo, e siano $x, y \in V(G)$ due suoi vertici; si definisce *distanza tra x e y*, il numero minimo di archi che costituiscono un cammino $x \to y$. In simboli, sia $\mathcal{C}$ l'insieme dei cammini $x \to y$ in G; allora

$$\operatorname{dist}(x, y) := \min_{c \in \mathcal{C}} |E(c)|$$

Algoritmo 1.5.1.1 Dato un grafo G, rappresentato attraverso liste di adiacenza, e un vettore di padri di un albero/arborescenza di visita in DFS di G, l'algoritmo restituisce la distanza di ogni vertice dalla radice dell'albero/arborescenza.
Input: G grafo diretto, rappresentato attraverso liste di adiacenza; `parents` un array di padri di un albero/arborescenza radicato in un certo $r \in V(G)$.
Output: $\forall v \in V(G) \quad \text{dist}(r, v)$.

```
1: function updateDistances(v, parents, distances)
2:     if distances[parents[v]] == −1 then          ▷ se non ancora visitato
3:         updateDistances(parents[v], parents, distances)
4:     end if
5:     distances[v] = distances[parents[v]] + 1
6: end function
7:
8: function findRoot(G, parents)
9:     for v ∈ V(G) do
10:        if parents[v] == v then
11:            return v
12:        end if
13:    end for
14: end function
15:
16: function findDistances(G, parents)
17:    r := findRoot(G, parents)
18:    distances := [ − 1] ∗ n
19:    distances[r] = 0
20:    for v ∈ V(G) do
21:        if distances[v] == −1 then                ▷ se non ancora visitato
22:            updateDistances(v, parents, distances)
23:        end if
24:    end for
25:    return distances
26: end function
```

Osservazione 1.5.1.1 (Correttezza dell'algoritmo)**.** L'algoritmo inizia istanziando un array `distances`, posto a -1 per ogni elemento, valore sentinella che verrà sfruttato dall'algoritmo per determinare se un dato vertice è stato visitato o meno.

Come primo passo, alla riga 17 viene trovata la radice r dell'array `parents`, il quale rappresenta un albero/arborescenza di visita in DFS del grafo G in input, grazie alla funzione `findRoot`; quest'ultima, per trovare la radice, esegue un ciclo `for` sull'array `parents`, restituendo l'unico vertice $v \in V(G)$: `parents[`v`]` $= v$.

Infine, viene salvata la distanza di r dalla radice in `distances`, pari naturalmente a 0, alla riga 19.

Successivamente, viene istanziato un ciclo `for`, alla riga 20, che per ogni vertice $v \in V(G)$ non ancora visitato dall'algoritmo (contrassegnato con $\texttt{distances[}v\texttt{]} = -1$), effettua una chiamata ricorsiva alla funzione `updateDistances`.

All'interno di `updateDistances`, alla riga 2, viene controllato se la distanza del padre del vertice v corrente, dalla radice di `parents`, non sia nota, e in tal caso viene effettuata una chiamata ricorsiva radicata proprio nel padre; questo, di fatto, ha l'effetto di "risalire" l'albero rappresentato da `parents`, di padre in padre, fintanto che non viene trovato un vertice la cui distanza dalla radice è nota; si noti che, inizialmente, l'unico vertice la cui distanza è nota è la radice stessa, e dunque la riga 19 funge da *caso base* della ricorsione. Infine, una volta risalito l'albero fino alla radice, vengono rimosse le chiamate ricorsive dallo stack-call, aggiornando i valori delle distanze, ora note, dalla radice, secondo la legge

$$\texttt{distances[}v\texttt{]} = \texttt{distances[parents[}v\texttt{]]} + 1$$

alla riga 5, poiché naturalmente ogni figlio v sarà ad una distanza, dalla radice r, pari alla distanza del proprio padre + 1, quest'ultimo dato dall'arco $(\texttt{parents[}v\texttt{]}, v)$ stesso.

Osservazione 1.5.1.2 (Costo dell'algoritmo)**.** L'algoritmo esegue inizialmente una chiamata alla funzione `findRoot`, la quale effettua un singolo ciclo `for`, che nel caso peggiore ha costo $O(n)$.

Successivamente, viene effettuato un ciclo `for` per ogni vertice del grafo G, il quale, per ogni nodo non ancora visitato, chiama `updateDistances`; all'interno di quest'ultima, si noti che la ricorsione viene effettuata esclusivamente se il padre del vertice corrente non è stato ancora visitato (riga 2), e dunque l'intero ciclo `for` della riga 20 ha costo $O(n)$, poiché è garantito che ogni vertice $v \in V(G)$ viene visitato una ed una sola volta.

Allora, il costo dell'algoritmo è pari a $O(n) + O(n) = O(n)$.

1.5.2 Visita in BFS

Definizione 1.5.2.1 (BFS)**.** Con BFS si indica un criterio di visita di un grafo; in particolare, BFS sta per *Breadth-first Search*, dunque la visita del grafo avviene controllando, prima di procedere al prossimo livello, tutti i vertici adiacenti al nodo del livello corrente.

Algoritmo 1.5.2.1 Dato un grafo G, rappresentato attraverso liste di adiacenza (nel caso di grafo diretto, è sufficiente memorizzare gli archi uscenti per ogni vertice), ed un suo vertice $u \in V(G)$, l'algoritmo restituisce le distanze dei vertici di G da u.
Input: G grafo, rappresentato attraverso liste di adiacenza; $u \in V(G)$ un vertice di G.
Output: $\forall v \in V(G) \quad \text{dist}(u, v)$.

```
1:  function findDistances(G, u)
2:      parents := [0] * n
3:      parents[u] = u
4:      distances := [0] * n
5:      Queue Q := [u]
6:      while !Q.isEmpty() do
7:          v = Q.deque()
8:          for x ∈ V(G) : x ∼ v do
9:              if { distances[x] == 0
                   { parents[x] ≠ x        then
10:                 parents[x] = v
11:                 distances[x] = distances[v] + 1
12:                 Q.enqueue(x)
13:             end if
14:         end for
15:     end while
16:     return distances
17: end function
```

Osservazione 1.5.2.1 (Correttezza dell'algoritmo)**.** L'algoritmo inizia definendo un array di nodi padri `parents`, un array di distanze `distances` (che conterrà le distanze dei vertici di G dalla radice u di partenza), e una coda `Q`, contenente u inizialmente (riga 5).

Successivamente, fintanto che la coda `Q` non è vuota, viene eseguito il ciclo `while` della riga 6, all'interno del quale viene rimosso il primo nodo v della coda `Q`, alla riga 7; inoltre, per ognuno dei suoi nodi adiacenti $x \sim v$, se non ancora visitato (la sua distanza da u è ancora 0), e non è la radice, ne viene aggiornato il padre, alla riga 10, e viene aggiornata la sua distanza da u, sommando 1 alla distanza del padre, ossia v, dalla radice; infine, x viene posizionato in coda. In particolare, quest'ultima riga permette all'algoritmo di effettuare una visita in BFS del grafo, poiché ogni vertice adiacente a v trovato viene inserito all'interno della coda non appena viene visitato.

Ad esempio, si consideri il seguente grafo:

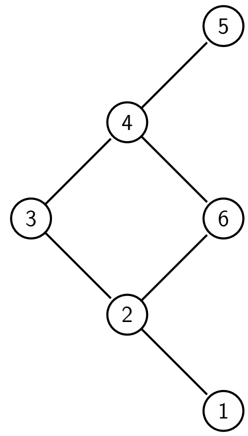


Figura 1.24: Un grafo indiretto.

partendo dal vertice $u = 3$, l'ordine di visita in BFS dei vertici è il seguente:

$$\{3, 4, 2, 5, 6, 1\}$$

Osservazione 1.5.2.2 (Costo dell'algoritmo)**.** Il ciclo `for` della riga 8 viene eseguito per ognuno dei vertici adiacenti a v, e dunque ha costo $O(\deg(v))$; allora, per ragionamento analogo all'Osservazione 1.3.2.6, il costo dell'algoritmo è pari a $O(n + m)$.

1.5.3 Trovare il numero di cammini minimi

Algoritmo 1.5.3.1 Dato un grafo G, rappresentato attraverso liste di adiacenza, ed un suo vertice u, per ogni $v \in V(G)$, l'algoritmo restituisce il numero di cammini minimi della forma $u \to v$.
Input: G grafo, rappresentato attraverso liste di adiacenza; $u \in V(G)$ un vertice di G.
Output: per ogni vertice $v \in V(G)$, il numero di cammini minimi della forma $u \to v$.

```
1:  function countShortestPaths(G, u)
2:      parents := [−1] ∗ n
3:      parents[u] = u
4:      count := [0] ∗ n
5:      count[u] = 1                       ▷ solo un modo per giungere ad u stesso
6:      distances := [0] ∗ n
7:      Queue Q := [u]
8:      while !Q.isEmpty() do
9:          v = Q.deque()
10:         for y ∈ V(G) : y ∼ v do
11:             if parents[y] == −1 then       ▷ se non è stato ancora visitato
12:                 parents[y] = v
13:                 distances[y] = distances[v] + 1
14:                 count[y] = count[v]
15:                 Q.enqueue(y)
16:             else if distances[y] == distances[v] + 1 then
17:                 count[y] + = count[v]
18:             end if
19:         end for
20:     end while
21:     return count
22: end function
```

Osservazione 1.5.3.1 (Correttezza dell'algoritmo)**.** L'algoritmo presenta una semplice modifica dell'Algoritmo 1.5.2.1, inserendo le righe 14, 16 e 17:

- alla riga 14, viene posto il numero di cammini minimi per raggiungere y partendo da u, pari al numero di cammini minimi di v, ovvero il suo nodo padre; ciò, in quanto alla riga 11 viene controllato che il nodo non sia stato ancora visitato, e dunque il suo valore in `count` sarà ancora pari a 0, e con la riga 14 viene dunque semplicemente rimpiazzato; inoltre, il numero di cammini minimi per raggiungere un figlio, sara sicuramente almeno pari a quello del padre;

- alla riga 16, viene controllato che il valore della distanza sia stato aggiornato correttamente, ed in tal caso, alla riga 17 viene accumulato al conteggio di y, il conteggio di v, ovvero il suo nodo padre.

Infatti, si noti che avendo un vertice $y \in V(G)$, figlio di $v_1, \ldots, v_k \in V(G)$, è sempre vero che

$$\texttt{count}[y] = \texttt{count}[v_1] + \ldots + \texttt{count}[v_k]$$

Osservazione 1.5.3.2 (Costo dell'algoritmo). L'algoritmo effettua esclusivamente una visita in BFS del grafo G in input, e dunque il suo costo è pari a $O(n + m)$.

1.5.4 Distanza tra insiemi di vertici

Definizione 1.5.4.1 (Distanza tra insiemi di vertici). Sia G un grafo, e $X, Y \subseteq V(G)$ due sottoinsiemi di vertici di G; allora, si definisce *distanza tra* X *e* Y la distanza minima tra due vertici di X e Y; in simboli

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

Algoritmo 1.5.4.1 Dato un grafo G, rappresentato attraverso liste di adiacenza, e due suoi sottoinsiemi di vertici $X, Y \subseteq V(G)$, l'algoritmo restituisce $\mathrm{dist}(X, Y)$.
Input: G grafo, rappresentato attraverso liste di adiacenza; $X, Y \subseteq V(G)$ sottoinsiemi di vertici di G.
Output: $\mathrm{dist}(X, Y)$.

```
1:  function setDistance(G, X, Y)
2:      distances := [-1] * n
3:      Queue Q := []
4:      for x ∈ X do
5:          Q.enqueue(x)
6:          distances[x] = 0                ▷ i vertici x ∈ X sono a distanza 0 da X
7:      end for
8:      while !Q.isEmpty() do
9:          v = Q.deque()
10:         for u ∈ V(G) : u ∼ v do
11:             if distances[u] == -1 then  ▷ se non è stato ancora visitato
12:                 distances[u] = distances[v] + 1
13:                 Q.enqueue(u)
14:             end if
15:         end for
16:     end while
17:     d := +∞
18:     for y ∈ Y do
19:         if distances[y] < d then
20:             d = distances[y]
21:         end if
22:     end for
23:     return count
24: end function
```

Osservazione 1.5.4.1 (Correttezza dell'algoritmo)**.** L'algoritmo inizia inserendo ogni vertice $x \in X$ in `Q`, all'interno del ciclo `for` della riga 4; successivamente, viene effettuata una visita in BFS di G, salvando le distanze trovate all'interno dell'array `distances`.

Si noti che, per dimostrazione precedente, l'algoritmo di visita in BFS trova la distranza tra vertici, naturalmente relativa al vertice di partenza; tale caratteristica può essere sfruttata, ad esempio in questo algoritmo, per calcolare tutte le distanze tra i vertici di X e di Y, semplicemente ponendo ogni vertice di X all'interno della coda `Q` all'inizio dell'algoritmo: così facendo, la visita in BFS calcolerà le distanze a partire dai vertici $x \in X$, che avranno `distances[`x`]` $= 0$, e per trovare $\text{dist}(X, Y)$ cercata, è sufficiente un ciclo `for`, alla riga 18, con il quale viene semplicemente trovata

$$d := \min_{y \in Y} \texttt{distances[}y\texttt{]} = \min_{x \in X, y \in Y} \text{dist}(x, y) =: \text{dist}(X, Y)$$

Osservazione 1.5.4.2 (Costo dell'algoritmo)**.** L'algoritmo è costituito da una visita in BFS del grafo G in input, partendo dai vertici in X, che ha costo $O(n+m)$ per osservazioni precedenti, seguita da un ciclo `for` su ogni vertice $y \in Y$, che ha dunque costo $O(|Y|)$; di conseguenza, il costo dell'algoritmo è pari a $O(n+m) + O(|Y|)$, ma si noti che $Y \subseteq V(G) \implies |Y| \leq |V(G)| =: n \implies O(n+m) + O(|Y|) = O(n+m)$.

1.5.5 Archi pesati

Definizione 1.5.5.1 (Archi pesati)**.** Sia G un grafo; su esso, è possibile definire una funzione

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

che, ad ogni arco, associa un valore reale positivo detto *peso*.

Esempio 1.5.5.1 (Grafo indiretto pesato)**.** Ad esempio, si consideri il seguente grafo indiretto:

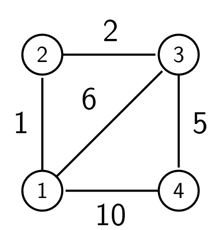


Figura 1.25: Un grafo indiretto pesato.

su di esso, sono stati inseriti dei valori sugli archi, che ne indicano il peso, secondo la funzione w che associa

$$w(1,2) = 1;\ w(2,3) = 2;\ w(3,4) = 5;\ w(4,1) = 10;\ w(1,3) = 6$$

Definizione 1.5.5.2 (Peso di un cammino)**.** Sia G un grafo, e c un cammino in G; si definisce *peso di* c la somma dei pesi degli archi che lo compongono. In simboli, sia $\mathcal{C}$ l'insieme dei cammini su G; allora, è possibile definire la funzione seguente

$$w_p : \mathcal{C} \to \mathbb{R}^+ : c \to \sum_{e \in E(c)} w(e)$$

che associa un cammino c al suo peso $w_p(c)$.

Definizione 1.5.5.3 (Distanza pesata)**.** Sia G un grafo pesato, e $x, y \in V(G)$ due suoi vertici; si definisce *distanza pesata tra x e y* il peso del cammino $x \to y$ di peso minimo. In simboli, sia $\mathcal{C}$ l'insieme dei cammini $x \to y$ in G; allora

$$\mathrm{dist}_w(x, y) := \min_{c \in \mathcal{C}} w_p(c)$$

Esempio 1.5.5.2 (Grafo indiretto pesato)**.** Ad esempio, si consideri il seguente grafo indiretto:

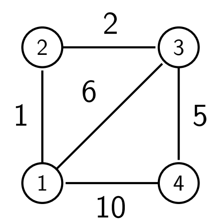


Figura 1.26: Un grafo indiretto pesato.

si noti che, in esso, si ha che

$$\mathrm{dist}(1, 4) = 10$$

attraversando l'arco $(1, 4)$, mentre

$$\mathrm{dist}_w(1, 4) = w(1, 2) + w(2, 3) + w(3, 4) = 1 + 2 + 5 = 8$$

attraversando gli archi $(1, 2), (2, 3)$ e $(3, 4)$, poiché

$$w(1, 4) = 10;\ w(1, 3) + w(3, 4) = 6 + 5 = 11$$

entrambi maggiori di 8.

Lemma 1.5.5.1 (Caratteristiche delle distanze pesate)**.** *Sia G un grafo; le seguenti proposizioni sono vere:*

i) $\forall x \in V(G) \quad \mathrm{dist}_w(x, x) = 0$

ii) $\forall x, y \in V(G) \quad \mathrm{dist}_w(x, y) \geq 0$

iii) $\forall x, y, z \in V(G) \quad \mathrm{dist}_w(x, y) \leq \mathrm{dist}_w(x, z) + \mathrm{dist}_w(z, y)$, *detta* disuguaglianza triangolare*.*

Dimostrazione.

i) La distanza da un nodo in sé stesso deve necessariamente essere nulla, poiché composta da 0 archi.

ii) Per definizione di w, la somma di numeri in $\mathbb{R}^+$ deve ancora essere in $\mathbb{R}^+$.

iii) Siano Q_1 e Q_2 cammini della forma $x \to z$ e $z \to y$ rispettivamente; si noti che il peso della passeggiata $Q_1 \cup Q_2$, della forma $x \to y$, è pari a

$$w_p(Q_1 \cup Q_2) = w(Q_1) + w(Q_2)$$

e inoltre, per il Teorema 1.1.2.1, è possibile trovare un cammino all'interno di tale passeggiata che, al più, attraversa lo stesso numero di archi di $Q_1 \cup Q_2$; allora segue che tale cammino deve avere peso minore o uguale al peso della passeggiata che lo contiene, e dunque esiste un cammino tale che

$$\text{dist}_w(x, y) \leq \text{dist}_w(x, z) + \text{dist}_w(z, y)$$

□

Osservazione 1.5.5.1 (Distanze pesate di un grafo diretto)**.** Sia G un grafo diretto, e $x, y \in V(G)$ due suoi vertici; si noti che, in un grafo diretto, dist_w non è necessariamente simmetrica, e potrebbe verificarsi che

$$\text{dist}_w(x, y) \neq \text{dist}_w(y, x)$$

Lemma 1.5.5.2 (Distanze pesate dei vicini)**.** *Sia G un grafo indiretto pesato attraverso w e $v \in V(G)$ un suo vertice; sia $N(v)$ l'insieme dei vertici adiacenti a v, e sia*

$$u := \arg\min_{x \in N(v)} w(v, x)$$

allora necessariamente

$$\text{dist}_w(v, u) = w(v, u)$$

Dimostrazione. Per ipotesi, u è il vertice adiacente v attraverso un arco $(v, u) \in E(G)$ di peso minimo tra gli adiacenti; allora, considerando qualsiasi altro cammino c' della forma $v \to u$, non passante per l'arco (v, u), si deve necessariamente avere

$$w_p(c') > w_p(\{v, (v, u), u\})$$

poiché il primo arco di c' avrà peso maggiore di $w(v, u)$, per come u è stato scelto in ipotesi. □

Teorema 1.5.5.1 (Estensioni pesate di insiemi di vertici). *Sia G un grafo indiretto, e sia $R \subseteq V(G)$ un sottoinsieme dei suoi vertici; sia $v \in R$ un vertice in R, e sia $(x, u) \in E(G)$ un arco con $x \in R$ e $u \in V(G) - R$, tale che*

$$(x, u) := \operatorname*{arg\,min}_{\substack{(a,b)\in E(G) \\ a \in R \\ b \in V(G)-R}} (\mathrm{dist}_w(v, x) + w(a, b))$$

allora necessariamente

$$\mathrm{dist}_w(v, u) = \mathrm{dist}_w(v, x) + w(x, u)$$

Dimostrazione. Per ipotesi, (x, u) è l'arco che minimizza $\mathrm{dist}_w(v, x) + w(x, u)$; si noti che, affinché sia verificata la tesi, è necessario dimostrare che esista un cammino $v \to u$, passante per (x, u), tale da minimizzare $\mathrm{dist}_w(v, x) + w(x, u)$, e che non esistano cammini $v \to u$, non passanati per (x, u), di peso inferiore:

- per la prima parte, è sufficiente considerare il cammino $v \to x$ che definisce $\mathrm{dist}_w(v, x)$, al quale è possibile aggiungere l'arco (x, u) stesso, per ottenere il cammino che minimizzi $\mathrm{dist}_w(v, x) + w(x, u)$, grazie al Lemma 1.5.5.2;
- per la seconda parte, sia Q un cammino $v \to u$, non passante per (x, u); allora Q deve necessariamente passare per un altro arco (x', u'), con $x' \in R$ e $u' \in V(G) - R$; ma poiché (x, u) è stato scelto tale da minimizzare la somma $\mathrm{dist}_w(v, x) + w(x, u)$, si ha che la porzione di cammino $v \to (x', u')$ deve essere pari o superiore a $w_p(v \to (x, u))$, e dunque $w_p(Q) \geq w_p(v \to (x, u))$; allora, il cammino cercato deve necessariamente passare per (x, u).

□

Capitolo 2

Algoritmi greedy

2.1 TODO

2.1.1 Definizioni

Definizione 2.1.1.1 (Algoritmo greedy)**.** Un algoritmo è detto *greedy*, se cerca una soluzione effettuando delle scelte di passo in passo, optando sempre per il passo più "appetibile" momentaneamente.

2.1.2 Trovare intervalli disgiunti

Algoritmo 2.1.2.1 Data una lista di intervalli, l'algoritmo restituisce l'insieme di restituisce il sottoinsieme di intervalli disgiunti, di cardinalità massima.
Input: I lista di intervalli di numeri reali della forma $[a, b]$, con $a, b \in \mathbb{R}$.
Output: il sottoinsieme di I di intervalli disgiunti di cardinalità massima.

```
function findIntervals(I)
    I.sort(key=λ(i) → i.right())            ▷ I ordinato sugli estremi destri
    Sol := []
    for i ∈ I do
        b_f := Sol.last().right()            ▷ estremo destro di Sol.last()
        if i.left() > b_f then
            Sol.append(i)
        end if
    end for
    return Sol
end function
```

Dimostrazione. Siano $\texttt{Sol}_0, \dots, \texttt{Sol}_n$ gli stati di $\texttt{Sol}_k$, ad ogni iterazione $k \in [1, n]$ dell'algoritmo; allora, l'algoritmo funziona correttamente se e solo se $\exists \texttt{Sol}^*$ soluzione ottimale $\mid \texttt{Sol}^* = \texttt{Sol}_n$.

Prima implicazione. $\exists \texttt{Sol}^*$ soluzione ottimale $\mid \texttt{Sol}_n \subseteq \texttt{Sol}^*$

- *caso base*
  - $k = 0 \implies \texttt{Sol}_0 := \varnothing \implies \forall \texttt{Sol}^*$ soluzione ottimale $\quad \texttt{Sol}_0 \subseteq \texttt{Sol}^*$
- *ipotesi induttiva*
  - $\exists \texttt{Sol}^*$ soluzione ottimale $\mid \texttt{Sol}_k \subseteq \texttt{Sol}^*$
- *passo induttivo*
  - è necessario dimostrare che $\exists \texttt{Sol}^*$ soluzione ottimale $\mid \texttt{Sol}_{k+1} \subseteq \texttt{Sol}^*$
  - si noti che, per via della riga 7, per ogni $k \in [1, n)$ si verifica che

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

  - allora, nel primo caso, se esiste un intervallo $[a_i, b_i] \in \texttt{Sol}_k$ tale per cui $[a_i, b_i] \cap [a_{k+1}, b_{k+1}] \neq \varnothing$, si ha che l'intervallo $[a_{k+1}, b_{k+1}]$ non può essere inserito in $\texttt{Sol}_{k+1}$, e dunque $\texttt{Sol}_{k+1} = \texttt{Sol}_k \implies \exists \texttt{Sol}^*$ soluzione ottimale $\mid \texttt{Sol}_{k+1} = \texttt{Sol}_k \subseteq \texttt{Sol}^*$ per ipotesi induttiva
  - se invece non esiste alcun intervallo $[a_i, b_i] \in \texttt{Sol}_k$, tale che $[a_i, b_i] \cap [a_{k+1}, b_{k+1}] \neq \varnothing$, allora $[a_{k+1}, b_{k+1}]$ deve essere inserito in $\texttt{Sol}_{k+1} = \texttt{Sol}_k \cup \{[a_{k+1}, b_{k+1}]\}$; si noti che, per definizione di $\texttt{Sol}_{k+1}$, si ha che

$$\forall k \in [1, n) \quad \texttt{Sol}_k \subseteq \texttt{Sol}_{k+1}$$

    allora, per dimostrare la tesi del passo induttivo è sufficiente considerare $[a_{k+1}, b_{k+1}]$; infatti $[a_{k+1}, b_{k+1}] \in \texttt{Sol}^* \implies \texttt{Sol}_{k+1} \subseteq \texttt{Sol}^*$ immediatamente
  - sia allora $[a_{k+1}, b_{k+1}] \notin \texttt{Sol}^*$; poiché $\texttt{Sol}^*$ è ottimale, tale condizione può verificarsi se e solo se

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

    dove $[a_j, b_j] \neq [a_{k+1}, b_{k+1}]$

- per assurdo sia $b_j < b_{k+1}$; si noti che $[a_j, b_j] \in \texttt{Sol}^* \implies [a_j, b_j]$ è disgiunto con ogni altro intervallo in $\texttt{Sol}^*$ per definizione; inoltre, poiché gli intervalli in I sono stati ordinati, in ordine crescente, attraverso il loro estremo destro, alla riga 2, allora $b_j < b_{k+1}$ implica che $[a_j, b_j]$ doveva essere stato esaminato prima della $(k+1)$-esima iterazione, e poiché $[a_j, b_j] \in \texttt{Sol}^*$, allora sicuramente $[a_j, b_j] \in \texttt{Sol}_k$, poiché $\texttt{Sol}_k \subseteq \texttt{Sol}^*$ per ipotesi induttiva, e dunque nessun intervallo in $\texttt{Sol}_k$ avrà intersezione con $[a_j, b_j]$
- allora $\left.\begin{array}{l} [a_j, b_j] \in \texttt{Sol}_k \subseteq \texttt{Sol}^* \\ [a_j, b_j] \cap [a_{k+1}, b_{k+1}] \neq \varnothing \end{array}\right\} \implies [a_{k+1}, b_{k+1}] \notin \texttt{Sol}_{k+1}$ ↯
- allora, segue necessariamente che $b_j > b_{k+1}$, e dunque $[a_j, b_j] \in \texttt{Sol}^* - \texttt{Sol}_k$; inoltre, poiché $[a_j, b_j] \cap [a_{k+1}, b_{k+1}] \neq \varnothing$ in ipotesi, si ha che $a_j < b_{k+1} < b_j$
- sia $[a_h, b_h] \in \texttt{Sol}^* - \texttt{Sol}_k \mid [a_h, b_h] \neq [a_j, b_j]$; poiché I è ordinato come precedentemente discusso, segue che $[a_h, b_h] \notin \texttt{Sol}_k \implies b_h > b_{k+1}$
- inoltre, per definizione $[a_h, b_h] \in \texttt{Sol}^* \iff \forall [a_i, b_i] \in \texttt{Sol}^* \quad [a_h, b_h] \cap [a_i, b_i] = \varnothing$, e poiché $[a_j, b_j] \in \texttt{Sol}^*$, in particolare si ha che $[a_h, b_h] \cap [a_j, b_j] = \varnothing$; allora, poiché $b_h > b_{k+1} \in (a_j, b_j)$, necessariamente

  $$a_j < b_{k+1} < b_j < a_h < b_h$$

  e in particolare, $b_{k+1} < a_h \implies [a_{k+1}, b_{k+1}] \cap [a_h, b_h] = \varnothing$
- questo dimostra che ogni intervallo $[a_h, b_h] \in \texttt{Sol}^* - \texttt{Sol}_k$, che non sia proprio $[a_j, b_j]$, è disgiunto con $[a_{k+1}, b_{k+1}]$; allora, per trovare un insieme $\texttt{Sol}^*$ soluzione ottimale, tale per cui $\texttt{Sol}_{k+1} \subseteq \texttt{Sol}^*$, è sufficiente considerare l'insieme

  $$(\texttt{Sol}^* - \{[a_j, b_j]\}) \cup \{[a_{k+1}, b_{k+1}]\}$$

  in quanto $\texttt{Sol}_k \subseteq \texttt{Sol}^*$ per ipotesi induttiva, e $\texttt{Sol}_{k+1} = \texttt{Sol}_k \cup \{[a_{k+1}, b_{k+1}]\}$ per osservazione precedente.

Seconda implicazione. $\exists \texttt{Sol}^*$ soluzione ottimale $\mid \texttt{Sol}^* \subseteq \texttt{Sol}_n$. Per assurdo, si assuma che $\nexists \texttt{Sol}^*$ soluzione ottimale $\mid \texttt{Sol}^* \subseteq \texttt{Sol}_n \iff \forall \texttt{Sol}^*$ soluzione ottimale $\quad \texttt{Sol}^* \not\subseteq \texttt{Sol}_n \iff$ per ogni soluzione ottimale $\texttt{Sol}^*$ si verifica una delle seguenti:

- $\texttt{Sol}^* \cap \texttt{Sol}_n = \varnothing$
- $\texttt{Sol}^* \cap \texttt{Sol}_n \neq \varnothing \land \texttt{Sol}^* \not\subseteq \texttt{Sol}_n \land \texttt{Sol}_n \not\subseteq \texttt{Sol}^*$

- $\texttt{Sol}^* \cap \texttt{Sol}_n \neq \varnothing \wedge \texttt{Sol}_n \subsetneq \texttt{Sol}^*$

  si noti che per dimostrazione precedente, deve necessariamente verificarsi il terzo caso; allora $\texttt{Sol}_n \subsetneq \texttt{Sol}^* \iff \exists [a_t, b_t] \in \texttt{Sol}^* - \texttt{Sol}_n$, dove $[a_t, b_t]$ è disgiunto con ogni altro intervallo di $\texttt{Sol}_n$, poiché $\texttt{Sol}_n \subset \texttt{Sol}^*$ per dimostrazione precedente, e $\texttt{Sol}^*$ è ottimale. Allora, poiché l'n-esima è l'ultima iterazione dell'algoritmo, si ha che $t \in [1, n]$, e dunque $[a_t, b_t] \notin \texttt{Sol}_n$ implicherebbe che l'algoritmo avrebbe sbagliato a non inserire tale intervallo in $\texttt{Sol}_n$, poiché sarebbe stato analizzato e scartato alla riga 6 $\lightning$.

$\square$

Osservazione 2.1.2.1 (Correttezza dell'algoritmo)**.** L'algoritmo inizia ordinando l'insieme di intervalli I, in ordine crescente del loro estremo destro, alla riga 2; successivamente, all'interno di `Sol` vengono inseriti tutti gli intervalli $i \in I$, tali che il loro estremo sinistro sia maggiore dell'estremo destro dell'ultimo elemento attualmente in `Sol`.

Infatti, è possibile controllare esclusivamente l'ultimo elemento in `Sol`, poiché l'ordinamento iniziale garantisce che gli unici intervalli inseriti al suo interno saranno disgiunti; inoltre, grazie al controllo della riga 6, l'estremo sinistro dell'intervallo corrente deve essere superiore dell'estremo destro di `Sol.last()`, affinchè questo possa essere inserito in `Sol`.

Infine, l'ordinamento della riga 2 garantisce di ottenere in output il sottoinsieme di intervalli disgiunti di cardinalità massima, poiché come primo intervallo in I verrà posto l'intervallo con l'estremo destro inferiore.

Osservazione 2.1.2.2 (Costo dell'algoritmo)**.** Si noti che, all'interno dell'algoritmo si assume `Sol.last()` abbia costo $O(1)$, andando ad indicizzare correttamente l'ultimo elemento della lista.

Poiché l'ordinamento di I, alla riga 2, indipendentemente dall'algoritmo di ordinamento scelto, non può avere costo inferiore a $O(n \log n)$, e il ciclo `for` della riga 4 ha costo $O(n)$, si ha che l'algoritmo ha costo $O(n \log n) + O(n) = O(n \log n)$.