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. Si noti che, per ogni grafo G, si verifica che $E(G) \subseteq V^2$.

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:

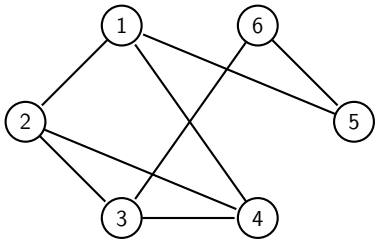


Figura 1.1: Un grafo indiretto.

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:

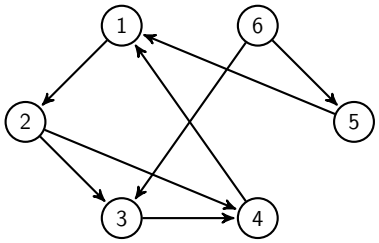


Figura 1.2: Un grafo diretto.

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 $\deg(v)$; all'interno di questi appunti, nel caso di grafi diretti, con $\deg(v)$ verrà inteso il numero di archi uscenti da v, mentre con $\deg^{in}(v)$ il numero di archi entranti in 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 $\sum_{v \in V(G)} \deg(v) = 2|E(G)|$. □

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:

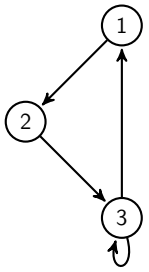


Figura 1.3: Un grafo diretto con cappio in 3.

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.

Definizione 1.1.1.9 (Multigrafo)**.** Un grafo è detto *multigrafo*, se non è un grafo 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:

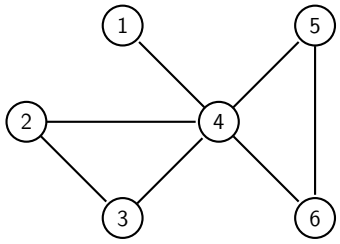


Figura 1.4: Un grafo indiretto.

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 di cardinalità minore, di vertici ed archi, della passeggiata, tale da congiungere u e v.
>
> *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:

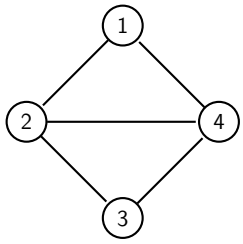


Figura 1.5: Un grafo indiretto.

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:

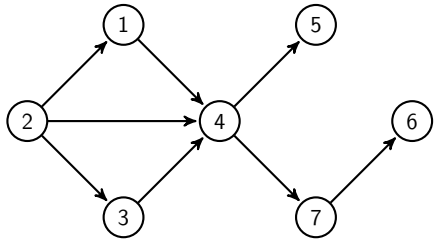


Figura 1.6: Un grafo diretto con ordinamenti topologici.

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 $v_1, \ldots, v_{k-1}$ ↯.

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:

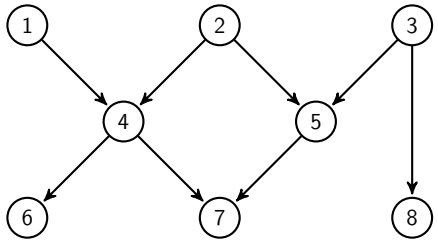


Figura 1.7: Un grafo diretto.

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

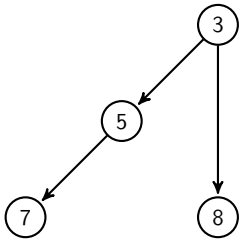


Figura 1.8: Arborescenza di 3.

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 indiretti, 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:

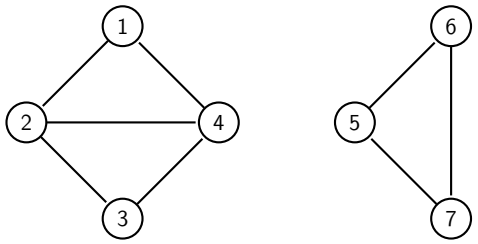


Figura 1.9: Un grafo non connesso.

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:

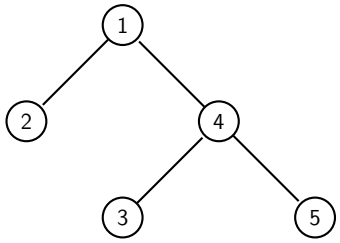


Figura 1.10: Un albero.

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:

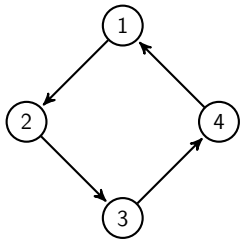


Figura 1.11: Un grafo fortemente connesso.

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:

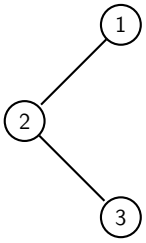


Figura 1.12: Un grafo indiretto.

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 $O(n)$ se si pone il valore `NULL` a l'intera riga e l'intera colonna del nodo da rimuovere, altrimenti è $O(n^2)$ nel caso in cui la matrice è da ricostruire.

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

1.2.2 Liste di adiacenza

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

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

Osservazione 1.2.2.1 (Spazio delle liste)**.** Dato un certo $v \in V(G)$, la lista di adiacenza corrispondente ha lunghezza $\deg(v)$; allora, il numero di elementi nelle liste di adiacenza, per il Lemma 1.1.1.1, è pari a $O\left(\sum_{v \in V(G)} \deg(v)\right) = O\left(2|E(G)|\right) = O(2m) = O(m)$. Si noti inoltre che, per un grafo con pochi archi, nonostante si abbiano le liste poco riempite, è comunque necessario salvare i puntatori a tali liste, e dunque è necessario introdurre un $O(n)$ nel costo totale dello spazio, ottenendo allora $O(n) + O(m) = O(n+m)$.

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

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

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

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

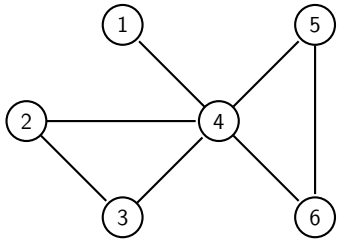


Figura 1.13: Un grafo indiretto.

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

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

1.3 Depth-first Search (DFS)

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 findCycle(G)
2:     v ∈ V(G)                                   ▷ un vertice qualsiasi di G
3:     visited := {v}                             ▷ conterrà i vertici visitati
4:     v' ∈ V(G) : v ~ v'
5:     while v' ∉ visited do          ▷ tempo costante perché visited è un set
6:         visited.add(v')
7:         v' := v'' ∈ V(G) : { v' ~ v''
                              { v'' ≠ visited[visited.length() − 2]
8:     end while
9:     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

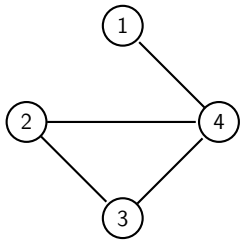


Figura 1.14: Un grafo diretto contenente un ciclo.

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 G, diretto o indiretto, e un suo vertice v, l'algoritmo restituisce tutti i vertici, raggiungibili attraverso cammini, partendo da v.
Input: G un grafo; v un vertice di G.
Output: i vertici raggiungibili da v.

```
1:  function findReachableNodes1(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 $\lightning$. □

Osservazione 1.3.2.1 (Correttezza dell'algoritmo)**.** L'algoritmo inizia definendo un array `visited`, alla riga 2, contenente inizialmente il valore sentinella 0 per ogni

vertice, il quale marcherà la presenza di nodi non ancora visitati dall'algoritmo; successivamente, viene segnato v, il vertice in input, come visitato all'interno di `visited`, alla riga 3, e viene posto all'interno dello stack `S`, alla riga 4.

Alla riga 5, viene inizializzato un ciclo `while`, all'interno del quale, fintanto che lo stack `S` non è vuoto, alla riga 6 viene definito v_{top}, pari al primo elemento dello stack, e se esiste un vertice $z \in V(G)$, non ancora visitato dall'algoritmo (`visited[`z`]` $= 0$), tale che sia adiacente a v_{top}, viene inserito all'interno dello stack, alla riga 8, e viene marcato come visitato, alla riga 9. Si noti che queste due ultime operazioni garantiscono che la visita del grafo sia in DFS, poiché v_{top} alla prossima iterazione del `while` sarà proprio il vertice z appena inserito, e dunque l'algoritmo sta procedendo più in profondità che si possa; se invece un tale vertice z non esiste, può voler dire esclusivamente che ogni singolo vertice adiacente a v_{top} corrente è già stato visitato dall'algoritmo, e dunque è possibile retrocedere di profondità nella visita del grafo, andando dunque a rimuovere dallo stack l'attuale primo vertice, alla riga 11.

Per terminare, è sufficiente dunque ritornare l'array `visited`, che conterrà 1 su tutti e soli i vertici che sono stati raggiunti dall'algoritmo, e sono dunque raggiungibili dal vertice di partenza v; infatti, gli unici vertici che non sono stati raggiunti dall'algoritmo, sono vertici che non sono raggiungibili, partendo da v, attraverso cammini. Si noti che, nel caso di un G indiretto, è sempre possibile raggiungere qualsiasi vertice, partendo da qualsiasi nodo, a meno di grafi non connessi.

Osservazione 1.3.2.2 (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 1.3.2.3 (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 1.3.2.4 (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 1.3.2.3; tale sottografo è un arborescenza di v.

Algoritmo 1.3.2.2 Seconda versione dell'algoritmo; dato un grafo G, diretto o indiretto, rappresentato attraverso liste di adiacenza, e un suo vertice v, l'algoritmo restituisce tutti i vertici, raggiungibili attraverso cammini, partendo da v.
Input: G un grafo, 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 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 $O\left(\sum_{v\in V(G)} 1+\deg(v)\right) = O\left(\sum_{v\in V(G)} 1\right) + O\left(\sum_{v\in V(G)} \deg(v)\right) = O(n)+O(m) = O(n+m)$, 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 $\text{Int}(v) := [t(v), T(v)]$.

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) $\text{Int}(u) \subseteq \text{Int}(v)$

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

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

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 *propria* non vuota tra i due intervalli, e dunque non è possibile che $\text{Int}(u) \cap \text{Int}(v) \neq \varnothing$, 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 $T(u) < T(v)$ $\lightning$

□

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

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

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* $\text{Int}(u) \subseteq \text{Int}(v)$*, oppure* $\text{Int}(v) \subseteq \text{Int}(u)$*, 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 $\text{Int}(u) \cap \text{Int}(v) = \varnothing$, 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 $u \sim v$. □

Algoritmo 1.4.1.1 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 1.4.1.3 (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 1.4.1.4 (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 $O(n+m)$.

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

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. Ad esempio, si consideri il seguente multigrafo diretto G:

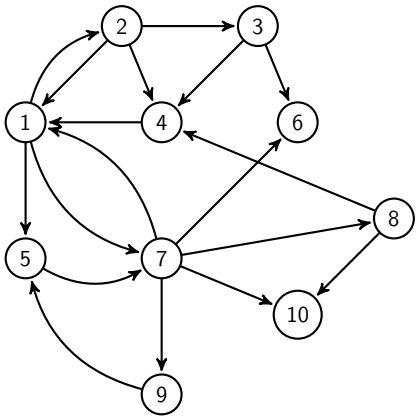


Figura 1.15: Un multigrafo diretto.

sia A_1 la seguente arborescenza di visita in DFS di G, radicata in 1:

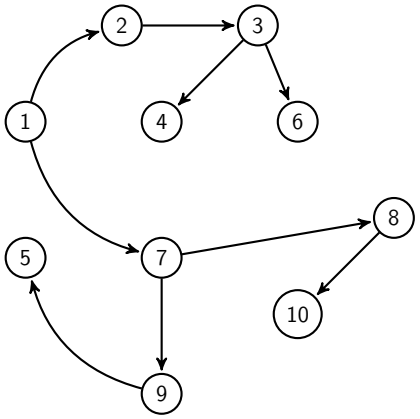


Figura 1.16: Un'arborescenza, radicata in 1, di un multigrafo diretto.

e siano inoltre i seguenti i tempi di visita e di chiusura di ogni vertice di G, relativi ad A_1

v	$t(v)$	$T(v)$
1	1	10
2	2	5
3	3	5
4	5	5
5	10	10
6	4	4
7	6	10
8	7	8
9	9	10
10	8	8

allora, è possibile classificare gli archi in $E(G) - E(A_1)$, attraverso i loro tempi di visita e di chiusura come segue

backward	$\{(2,1),(4,1),(5,7),(7,1)\}$
forward	$\{(1,5),(2,4),(7,10)\}$
cross	$\{(7,6),(8,4)\}$

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 si ha che $E(G) = E(T_{\hat{v}}) \implies G = T_{\hat{v}} \implies G$ è un albero, e di conseguenza G non ha cicli $\lightning$.
>
> *Seconda implicazione.* Sia G un grafo indiretto connesso, sia $\hat{v} \in V(G)$, sia $T_{\hat{v}}$ il suo albero, e sia $(u,v) \in E(G) - E(T_{\hat{v}})$ un backward edge. Allora, poiché $u, v \in V(T_{\hat{v}})$, è sufficiente considerare il cammino tale che $u \to v$, che esiste poiché $T_{\hat{v}}$ è un albero, e dunque $\{u \to v\} \cup (u,v)$ è un ciclo di G.

□

Algoritmo 1.4.2.1 Dato un grafo 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.

```
1:  function CATEGORIZEEDGES(G, v)
2:      visited := [0] * n
3:      visited[v] = 1
4:      Stack S := [v]
5:      c := 1
6:      t := [0] * n                                  ▷ tempi di visita
7:      T := [0] * n                                  ▷ tempi di chiusura
8:      t[v] = c
9:      parents := [0] * n
10:     parents[v] = v                                ▷ per riconoscere la radice
11:     while !S.isEmpty() do
12:         v_top := S.top()
13:         while !v_top.outgoing_adjacent().isEmpty() do
14:             z := v_top.outgoing_adjacent()[0]
15:             v_top.outgoing_adjacent().remove(0)
16:             if visited[z] == 0 then
17:                 visited[z] = 1
18:                 S.push(z)
19:                 parents[z] = v_top
20:                 c + = 1
21:                 t[z] = c
22:                 break
23:             end if
24:         end while
25:         if v_top == S.top() then
26:             S.pop()
27:             T[v_top] = c
28:         end if
29:     end while
```

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

Osservazione 1.4.2.3 (Correttezza dell'algoritmo)**.** Si noti che, dalla riga 2 alla riga 29, l'algoritmo seguente non presenta alcuna differenza dall'Algoritmo 1.3.2.2, se non per alcuni accorgimenti al fine di collezionare i tempi di visita e di chiusura dei vari vertici:

- alle righe 5, 6 e 7 viene istanziato un contatore `c`, che servirà per determinare i tempi di visita e di chiusura dei vari vertici, i quali verranno salvati all'interno dei due array `t` e `T` appena definiti;
- alla riga 8 viene aggiornato il tempo di visita di v di partenza, pari a `c`, poiché quest'ultimo è stato definito partendo da 1;
- alla riga 9 e 10 viene creato un array `parents`, che servirà per la seconda parte dell'algoritmo, ed in esso viene salvato il padre di v, pari a v stesso, poiché così facendo sarà possibile identificare la radice successivamente;
- alle righe 13, 14 e 15 è necessario controllare esclusivamente gli archi uscenti, poiché l'algoritmo deve progredire visitando i vertici adiacenti uscenti, e gli entranti sono quelli dai quali si proviene;
- alla riga 19, è necessario salvare il nodo padre di z, che sarà naturalmente v_{top};

- alle righe 20 e 21 viene incrementato il contatore dei tempi `c`, e viene posto il suo valore, appena incrementato, come tempo di visita di z;
- alla riga 27 viene aggiornato il tempo di chiusura di v_{top}, che sarà pari al valore di `c` corrente, poiché il contatore deve aumentare esclusivamente quando si incontrano vertici non ancora visitati.

Alle righe 30, 31 e 32 vengono definite 3 liste, che conterranno tutti gli archi del grafo, non facenti parte della visita di G in DFS appena realizzata dall'algoritmo, categorizzati opportunamente grazie ai cicli `for` delle righe 33 e 34. In particolare, per ogni vertice $x \in V(G)$ del grafo, vengono esaminati tutti i vertici u, che siano *adiacenti entranti* ad x, controllando di fatto l'arco (u, x):

- se il nodo padre di x è proprio u (riga 35), allora necessariamente l'arco è stato visitato dall'algoritmo nella prima fase, poiché si trova all'interno di `parents` stesso, e dunque fa parte della visita del grafo, ed è possibile ignorarlo;
- se il tempo di chiusura di uno dei due, tra x e u, è minore del tempo di visita dell'altro (riga 37), allora uno dei due è stato rimosso dallo stack, prima che l'altro potesse essere visitato, e l'unico caso in cui tale condizione può verificarsi, è se l'arco (u, x) è un cross edge;
- se il tempo di chiusura di u, il quale è entrante in x, è inferiore (o uguale) al tempo di chiusura di quest'ultimo (riga 39), allora x è stato rimosso dallo stack dopo u, nonostante u fosse adiacente entrante ad x, e dunque necessariamente l'arco (u, x) è un backward edge;
- infine, l'ultimo caso possibile restante (riga 41) è un forward edge.

L'algoritmo conclude restituendo le liste contenenti i vari archi del grafo, non facenti parte dell'arborescenza di visita di G, radicata in v, categorizzati sfruttando i loro intervalli.

Osservazione 1.4.2.4 (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 $O(n + m)$.

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

Algoritmo 1.4.2.2 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).

```
1:  function CATEGORIZEEDGE(parents, (x, y))
2:      if parents[y] == x then
3:          return NodeType::Arborescence
4:      end if
5:      z := y
6:      while parents[z] ≠ z do
7:          z = parents[z]
8:          if z == x then
9:              return NodeType::Forward
10:         end if
11:     end while
12:     z := x
13:     while parents[z] ≠ z do
14:         z = parents[z]
15:         if z == y then
16:             return NodeType::Backward
17:         end if
18:     end while
19:     return NodeType::Cross
20: end function
```

Osservazione 1.4.2.5 (Correttezza dell'algoritmo)**.** Sia `NodeType` un `enum`, che può assumere le 4 varianti `Arborescence`, `Forward`, `Backward` e `Cross`.

L'algoritmo inizia controllando, alla riga 2, se il vertice x è proprio il nodo padre di y: in tal caso, vorrebbe dire che il vertice appartiene a `parents`, e dunque appartiene all'arborescenza di visita del grafo; allora, viene restituita la variante `NodeType::Arborescence`, alla riga 3.

Successivamente, alla riga 5 viene definito z, posto inizialmente ad y, ed all'interno del ciclo `while` della riga seguente, fintanto che non viene trovata la radice (condizione di uscita, alla riga 6), viene risalita la catena di nodi padri, e se durante il percorso viene trovato proprio x, allora vuol dire che x si trova prima di y all'interno della visita in DFS del grafo, e poiché l'arco (x, y) è diretto incidente su x, allora questo deve essere un forward edge, ed è dunque sufficiente restituire `NodeType::Forward`, alla riga 9.

Simmetricamente, viene eseguito lo stesso ciclo `while`, ma partendo da x, per cercare y, e se quest'ultimo viene trovato, allora vuol dire che y viene prima di x nell'albero di visita, ma poiché l'arco (x, y) è diretto incidente su y, l'unica possibilità è che l'arco sia un backward edge, e dunque l'algoritmo restituisce `NodeType::Backward` in tal caso, alla riga 16.

Infine, se l'arco (x, y) in input non rispetta nessuna delle condizioni precedenti, segue che questo è un cross edge, e viene dunque restituito `NodeType::Cross` alla riga 18.

Osservazione 1.4.2.6 (Costo dell'algoritmo)**.** I cicli `while` delle righe 6 e 12, poiché identici, hanno lo stesso costo, ovvero $O(n)$, in quanto semplicemente risalgono ogni nodo della catena di padri in `parents`, il quale ha esattamente n nodi, e dunque nel caso peggiore verranno visitati tutti i nodi fino alla radice.

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

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.* Omessa.
>
> *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.

□

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:

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

□

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. □

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

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

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

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

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

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

1.4.4 Trovare un pozzo universale

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:

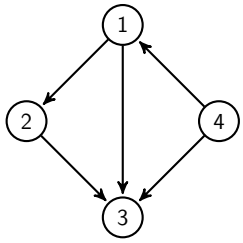


Figura 1.17: Un grafo con pozzo universale in 3.

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:

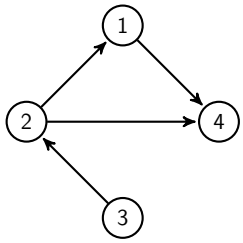


Figura 1.18: Un grafo che contiene un possibile pozzo universale.

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:

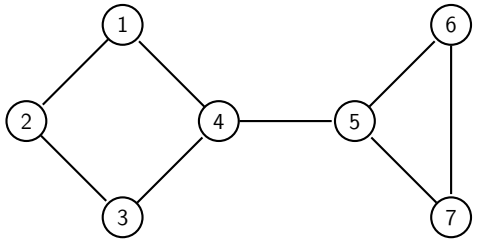


Figura 1.19: Un grafo con un ponte.

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 appartiene 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.

```
1:  function DFS(G, y, c, back, t, parents)
2:      c.increment()
3:      t[y] = c
4:      back[y] = t[y]
5:      for z ∈ V(G) : z ∼ y do
6:          if t[z] == 0 then                ▷ z non deve essere già stato visitato
7:              parents[z] = y
8:              DFS(G, z, c, back, t, parents)
9:              if back[z] < back[y] then
10:                 back[y] = back[z]
11:             end if
12:         else if { z ≠ parents[y]
                    { t[z] < back[y]   then
13:             back[y] = back[z]
14:         end if
15:     end for
16: end function
17:
18: function findBridges(G)
19:     v ∈ V(G)                             ▷ un vertice qualsiasi di G
20:     t := [0] * n
21:     parents := [0] * n
22:     parents[v] = v
23:     back := [0] * n
24:     Counter c := 0                       ▷ è un oggetto
25:     DFS(G, v, c, back, t, parents)
26:     bridges := {}
27:     for u ∈ V(G) do
28:         if { back[u] = t[u]
               { u ≠ parents[u]   then
29:             bridges.add((parents[u], u))
30:         end if
31:     end for
32:     return bridges
33: 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 `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 (`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 `back[`y`]` $=$ min(`back[`y`]`$,$ `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 `back[`y`]` con min(`t[`z`]`$,$ `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 $u \neq$ `parents[`u`]` (ovvero, u non è la radice), e `back[`u`]` $=$ `t[`u`]`, dunque il valore di `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:

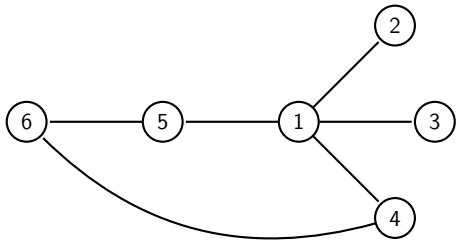


Figura 1.20: Un grafo indiretto.

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

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

ovvero

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

poiché $\texttt{back}[2] = \texttt{t}[2] = 4$ e $\texttt{back}[3] = \texttt{t}[3] = 5$; infine, si noti che $\texttt{back}[6] = \texttt{back}[6] = 1$, ma $6 \notin \texttt{bridges}$ poiché $\texttt{parents}[6] = 6$, ed 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:

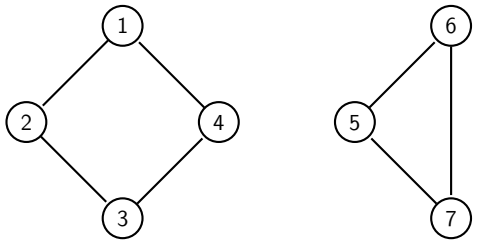


Figura 1.21: Un grafo indiretto.

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:

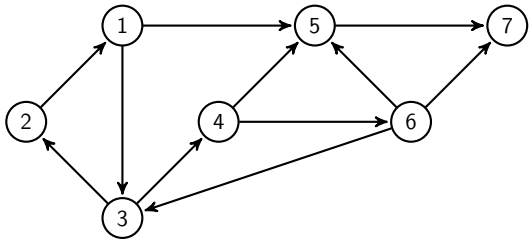


Figura 1.22: Un grafo diretto.

esso, presenta 3 componenti:

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

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

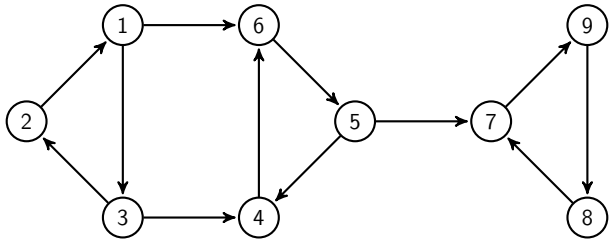
$$H_3 := (\{7\}, \varnothing)$$

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

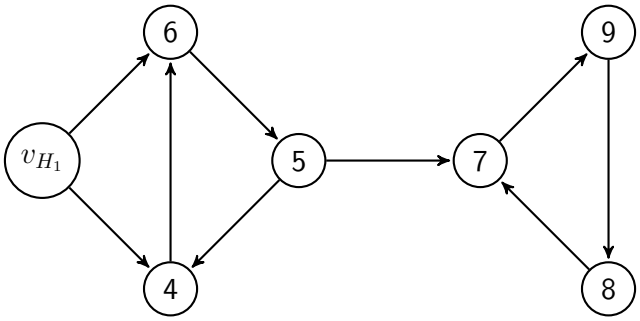
Dimostrazione. Per assurdo, sia G un grafo, diretto o indiretto, e $H_1, H_2 \subseteq G$ due sue componenti tali che $H_1 \cap H_2 \neq \varnothing$; se tali componenti esistessero, allora esisterebbe almeno un vertice nella loro intersezione, raggiungibile da entrambe le componenti (nel caso di G diretto, 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 $\lightning$. □

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 $\text{contr}(G, H)$.

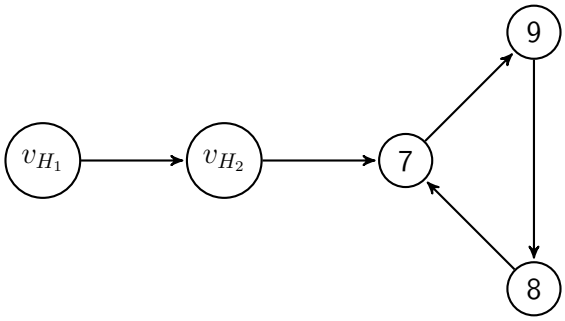
Esempio 1.4.6.3 (Contrazione di un grafo diretto)**.** Si consideri il seguente grafo diretto:



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:

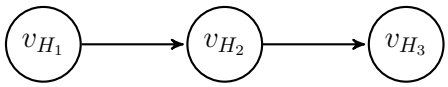


Figura 1.23: Il grafo iniziale massimalmente contratto.

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.

```
1:  function FINDCOMPONENTS_1(G)
2:      C := findCycle(G)
3:      if C == None then
4:          return {{v} : v ∈ V(G)}          ▷ le componenti sono i singoli vertici
5:      else
6:          v_C, G = contr(G, C)             ▷ v_C è il vertice della contrazione
7:          {H_1, ..., H_k} = findComponents(G)
8:          new_components := {}             ▷ conterrà le nuove componenti
9:          for i ∈ [1, k] do
10:             if v_C ∉ H_i then
11:                 new_components.add(H_i)
12:             else
13:                 new_components.add((H_i − {v_C}) ∪ V(C))
14:             end if
15:         end for
16:     end if
17:     return new_components
18: 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 $\text{contr}(H, C)$, 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.

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)**.** La funzione `findCycle` della riga 2 ha costo $O(n+m)$, ed utilizza la funzione dell'Algoritmo 1.4.8.2. Allora, poiché la riga 4 ha costo pari a $O(n)$, dovendo scorrere tutti i vertici dell'attuale G (che nel caso peggiore, contiene ogni vertice del grafo di partenza), il caso base ha costo $O(n+m)$.

L'operazione di contrazione, alla riga 6, ha costo $O(n+m)$, in quanto contrarre un grafo, nel caso peggiore, implica contrarne ogni vertice, e dunque modificarne ogni arco e nodo presente; inoltre, il ciclo `for` della riga 9 esegue un loop sulle componenti trovate alla riga 7 dalla chiamata ricorsiva, ed il suo costo dipende dunque dal numero di componenti che nel caso peggiore possono essere trovate, ovvero $n-1$, dunque $O(n)$, poiché un ciclo è costituito da minimo 2 nodi.

Allora, il costo dell'algoritmo è $O(n \cdot (n+m))$, poiché nel caso peggiore si hanno $n-1$ contrazioni (chiamate ricorsive), ovvvero $O(n)$, ed in ogni chiamata ricorsiva si effettuano operazioni in $O(n+m)$.

Si noti che questa stima è particolarmente approssimativa, e sarebbe necessario risolvere l'equazione di ricorrenza associata all'algoritmo per giustificarne completamente il costo dell'algoritmo, ma è omesso tale calcolo per semplicità.

1.4.7 Algoritmo di Tarjan

Definizione 1.4.7.1 (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 1.4.7.1 (C-radici)**.** Si consideri il seguente G grafo diretto:

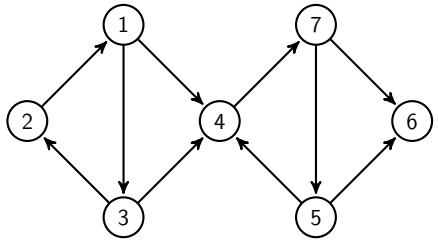


Figura 1.24: Un grafo diretto.

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:

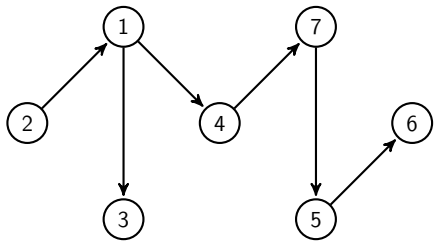


Figura 1.25: Arborescenza di 2.

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 1.4.7.1 (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 $u_1, \dots, u_k \in V(A_u)$, 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

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

e anche che

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

Allora, necessariamente

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

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.

□

Algoritmo 1.4.7.1 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.

```
1:  function DFS(G, u, components, S, c, cc)
2:      c.increment()
3:      components[u] = −c          ▷ i valori negativi indicano i tempi di visita
4:      S.push(u)
5:      b := c
6:      for v ∈ V(G) : (u, v) ∈ E(G) do
7:          if components[v] == 0 then
8:              b' := DFS(G, v, components, S, c, cc)
9:              b = min(b, b')
10:         else if components[v] < 0 then
11:             b = min(b, −components[v])
12:         end if
13:     end for
14:     if b == −components[u] then                          ▷ u è c-radice
15:         cc.increment()
16:         do
17:             w := S.pop()
18:             components[w] = cc
19:         while u ≠ w
20:     end if
21:     return b
22: end function
23:
24: function FINDCOMPONENTS2(G)
25:     components := [0] * n
26:     Counter cc := 0
27:     Counter c := 0
28:     Stack S := []
29:     for u ∈ V(G) do
30:         if components[u] == 0 then
31:             DFS(G, u, components, S, c, cc)
32:         end if
33:     end for
34:     return components
35: end function
```

Osservazione 1.4.7.1 (Correttezza dell'algoritmo)**.** L'algoritmo inizia alla riga 25, definendo un array `components`, attraverso il quale verranno trovate le varie componenti del grafo G in input; in particolare, alla riga 26 viene definito un contatore `cc`, affinché l'array `components` contenga, per ogni vertice $v \in V(G)$, un valore `components[`v`]` pari a `cc`, se v fa parte della `cc`-esima componente trovata dall'algoritmo; infine, alla riga 27 viene definito il contatore `c`, utilizzato per i tempi di visita dei vertici, ed alla riga 28 si ha uno stack `S`, che verrà utilizzato per trovare i vertici facenti parte della `cc`-esima componente.

Alla riga 29, l'algoritmo inizializza un ciclo `for`, che per ognuno dei vertici $u \in V(G)$, effettuerà una chiamata ricorsiva alla funzione `DFS` (riga 31), solo se u non era ancora stato visitato (riga 30).

Alla riga 2, l'algoritmo incrementa il contatore dei tempi di visita, per poterne utilizzare il valore correttamente (si noti che alla riga 27 viene inizializzato a 0, e poiché canonicamente il primo vertice ha tempo di visita pari ad 1, il contatore viene incrementato prima di essere utilizzato). Alla riga 3, l'opposto del valore di tale contatore viene inserito all'interno di `compontents[`u`]`: questa tecnica permette di ridurre spazio, utilizzando solamente un array per salvare sia i tempi di visita dell'algoritmo, sia le componenti da restituire in output; infatti per ogni $u \in V(G)$, se `components[`u`]` è negativo, allora si sta salvando $-t(u)$, se il valore è nullo, allora u non è ancora stato visitato, e se è positivo, allora u appartiene alla `components[`u`]`-esima componente; infine, u viene inserito all'interno dello stack, nella riga 4, e viene definito b, alla riga 5, che rappresenta il nodo più all'indietro che è possibile raggiungere da u stesso, e da uno qualsiasi dei suoi discendenti (cfr. il ciclo `for` seguente).

Alla riga 6, la funzione ricorsiva inizializza un ciclo `for`, che per ogni vertice v, adiacente uscente da u, effettua una chiamata ricorsiva, radicata su di lui, se quest'ultimo non era ancora stato visitato dall'algoritmo (riga 7), salvando il valore trovato dalla ricorsione in b', ed aggiornando opportunamente il valore di b alla riga 9; infatti, poichè b deve rappresentare il nodo più all'indietro che è possibile raggiungere a partire da u, e da un suo qualsiasi discendente, allora b deve necessariamente essere il minimo tra i b' di tutte le chiamate ricorsive effettuate sui figli (incluso il valore iniziale); alla riga 10, se invece il valore salvato all'interno dell'array `components`, per v, era negativo, allora ne era noto il tempo di vista ma non la componente a cui faceva parte, e dunque viene solamente aggiornato b, utilizzando proprio il tempo $t(v)$, pari a $-$`components[`v`]`.

Nella riga 14, è presente una condizione che viene valutata vera, esclusivamente se u è una c-radice della componente corrente: infatti, se b è pari a $-$`components[`u`]` $= t(u)$, allora il nodo più all'indietro che è possibile raggiungere attraverso un discendente qualsiasi di u, è u stesso, e dunque per il Teorema 1.4.7.1 e per definizione di c-radice stessa, u deve necessariamente essere una c-radice.

Allora, se u è c-radice, è sufficiente incrementare il contatore delle componenti, alla riga 15, ed aggiornare il valore di `components` di tutti i vertici accumulati all'interno dello stack durante le chiamate ricorsive effettuate più in profondità della chiamata corrente, con il valore di `cc`: infatti, lo stack viene utilizzato dall'algoritmo per salvare i vertici nello stesso ordine in cui sono visitati dalle chiamate ricorsive, e facendo `S.pop()` (riga 17), si è certi di preservare l'ordine di attraversamento del grafo G; si noti che tale operazione termina non appena viene eseguita la rimozione di u stessa dallo stack, poiché sarà esso l'ultimo vertice della `cc`-esima componente.

La funzione ricorsiva termina alla riga 21, restituendo al chiamante il valore b della chiamata ricorsiva corrente. Infine, l'algoritmo termina alla riga 34, restituendo tutte le componenti di G.

Osservazione 1.4.7.2 (Costo dell'algoritmo)**.** Poiché l'algoritmo effettua esclusivamente una visita in DFS del grafo G in input, e lo stack `S` conterrà n nodi nel caso peggiore (all'interno della chiamata ricorsiva più profonda della visita in DFS), l'algoritmo ha costo $O(n + m)$.

1.4.8 Trovare un ciclo

Algoritmo 1.4.8.1 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.

```
1:  function DFS(G, u, parents, visited)
2:      visited[u] = 1
3:      for v ∈ V(G) : v ∼ u do
4:          if visited[v] == 0 then
5:              visited[v] = 1
6:              parents[v] = u
7:              output = DFS(G, v, parents, visited)
8:              if output ≠ None then
9:                  return output
10:             end if
11:         else if { parents[u] ≠ v
                    { parents[v] ≠ u   then        ▷ l'array parents è diretto
12:             output := {}
13:             while u ≠ v do
14:                 output.add(u)
15:                 u = parents[u]
16:             end while
17:             return output
18:         end if
19:     end for
20:     return None
21: end function
```

```
22: function findCycle(G)
23:     visited := [0] * n
24:     parents := [0] * n
25:     for v ∈ V(G) do
26:         if visited[v] == 0 then
27:             parents[v] = v                          ▷ la radice corrente
28:             output = DFS(G, v, parents, visited)
29:             if output ≠ None then
30:                 return output
31:             end if
32:         end if
33:     end for
34:     return None
35: end function
```

Osservazione 1.4.8.1 (Correttezza dell'algoritmo)**.** 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. Si noti che il ciclo `for` permette di gestire il caso in cui G non sia connesso; inoltre, alla riga 27 viene posto `parents[v]` $= v$, di fatto definendo più alberi di visita sullo stesso grafo, poiché se G non fosse connesso, si avrebbero necessariamente molteplici visite in DFS, e di conseguenza molteplici alberi di visita.

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

Al termine della chiamata ricorsiva della riga 7, viene esaminato il valore restituito dalla funzione `DFS`, che può essere un ciclo di G, oppure `None`; se ci si trova nel primo caso, e dunque si è trovato un output richiesto dall'algoritmo, sarà sufficiente restituire il risultato di ricorsione in ricorsione a ritroso, ritornando `output` stesso, alla riga 9.

Se invece il vertice v era già stato visitato, è necessario eseguire un controllo, poiché ci si potrebbe trovare all'interno di un ciclo; si noti che il ciclo `for` della riga 3, trova tutti i vertici adiacenti ad u, compreso il nodo padre di u stesso, ma naturalmente non deve essere di interesse per la ricerca di un ciclo in G. Allora, se u non è il nodo padre di v, né v è il nodo padre di u (si noti che l'array `parents` è diretto, anche se G non lo è, dunque non è sufficiente controllare solo uno dei due casi), l'unico caso in cui può capitare di rivisitare un nodo è se ci si trova

all'interno di un ciclo; di conseguenza, in tale circostanza, l'algoritmo istanzia un insieme `output`, alla riga 12, all'interno del quale vengono inseriti tutti i vertici che vengono trovati, risalendo la catena di padri, costruitasi in `parents`, partendo da u stesso, fino a ritornare proprio su v; infine, viene restituito tale insieme, alla riga 17, che verrà propagato fino alla riga 28.

Nel controllo della riga 29, ci si accerta che non sia ancora stato trovato alcun ciclo, e non appena ne viene trovato uno, viene restituito dall'algoritmo, troncando il loop `for`.

Osservazione 1.4.8.2 (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 $O(n)$.

Algoritmo 1.4.8.2 Dato un grafo diretto G, rappresentato attraverso liste di adiacenza, l'algoritmo restituisce un suo ciclo, se presente.
Input: G grafo diretto, rappresentato attraverso liste di adiacenza.
Output: un ciclo di G, se presente.

```
1: function DFS(G, x, t, T, S, c)
2:     for y ∈ V(G) : (x, y) ∈ E(G) do
3:         if t[y] == 0 then
4:             t[y] = c
5:             c.increment()
6:             S.push(y)
7:             output = DFS(G, y, t, T, S, c)
8:             if output ≠ None then
9:                 return output
10:            end if
11:        else if T[y] == 0 then
12:            output := {y}                    ▷ altrimenti y non verrebbe preso
13:            while x ≠ y do
14:                output.add(x)
15:                x = S.pop()
16:            end while
17:            return output
18:        end if
19:    end for
20:    T[x] = c
21:    S.pop()
22:    return None
23: end function
```

```
24: function findCycle(G)
25:     t := [0] * n
26:     T := [0] * n
27:     Counter c := 1
28:     Stack s := []
29:     for x ∈ V(G) do
30:         if t[x] == 0 then
31:             output = DFS(G, v, t, T, S, c)
32:             if output ≠ None then
33:                 return output
34:             end if
35:         end if
36:     end for
37:     return None
38: end function
```

Osservazione 1.4.8.3 (Correttezza dell'algoritmo)**.** Omessa.

Osservazione 1.4.8.4 (Costo dell'algoritmo)**.** Poiché l'algoritmo effettua esclusivamente una visita in DFS del grafo G in input, il suo costo è pari a $O(n+m)$; si noti che, nel caso peggiore, l'intero grafo descrive un ciclo, e dunque il ciclo `while` della riga 13 ha costo $O(n)$.

1.5 Breadth-first Search (BFS)

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 array di padri di un grafo (nel caso questo fosse diretto, l'array rappresenterebbe una visita in DFS di tale grafo), e due suoi vertici x ed y, l'algoritmo restituisce la loro distanza.
Input: `parents` un array di padri di un grafo; x, y due vertici del grafo.
Output: $\text{dist}(x, y)$.

```
1:  function findCommonAncestor(parents, x, y)
2:      ancestorsX := [0] * n
3:      v := x
4:      while parents[v] ≠ v do
5:          ancestorsX[v] = 1
6:          v = parents[v]
7:      end while
8:      ancestorsX[v] = 1
9:      v := y
10:     while ancestorsX[v] == 0 do
11:         v = parents[v]
12:     end while
13:     return v
14: end function
15:
16: function distanceRoot(parents, v)
17:     if parents[v] == v then
18:         return 0
19:     else
20:         return distanceRoot(parents, parents[v]) + 1
21:     end if
22: end function
23:
24: function findDistance(parents, x, y)
25:     a := findCommonAncestor(parents, x, y)
26:     d_x := distanceRoot(parents, x)
27:     d_y := distanceRoot(parents, y)
28:     d_a := distanceRoot(parents, a)
29:     return d_x + d_y − 2 · d_a
30: end function
```

Osservazione 1.5.1.1 (Correttezza dell'algoritmo)**.** L'algoritmo inizia, alla riga 25, chiamando la funzione `findCommonAncestor`, che restituisce l'antenato, comune ad x ed y in input, a loro più vicino.

In particolare, viene inizialmente istanziato un array `ancestorsX` a 0, alla riga 2, e successivamente viene inizializzato un ciclo `while`, il quale è utilizzato per marcare gli antenati di x, risalendo fino alla radice (condizione di uscita del ciclo, alla riga 4); si noti che la riga 8 è necessaria, in quanto v, in quel momento, sarà proprio la radice, la quale non potrebbe essere marcata all'interno del ciclo appena terminato, poiché l'incontro della radice è proprio la condizione di uscita di quest'ultimo; inoltre, è importante notare che l'antenato comune più vicino ad x e y potrebbe proprio essere la radice stessa come caso limite, ed è dunque fondamentale segnare anche quest'ultima come antenato di x.

Nelle righe 9, 10 e 11 viene poi eseguito un ulteriore ciclo `while`, grazie al quale viene effettivamente trovato l'antenato comune ad x ed y, risalendo la catena di padri di y (riga 9), fintanto che gli attuali vertici v, padri di y, non siano ancora essi stessi antenati di x (condizione di uscita del ciclo, alla riga 10); infatti, si noti che non appena viene trovato un v tale che `ancestorsX[`v`]` sia pari ad 1, allora tale v è proprio il primo antenato in comune tra x ed y cercato, che va dunque ritornato, alla riga 13.

Dopo aver salvato l'antenato, comune ad x ed y, a loro più vicino, all'interno di a, alla riga 25, l'algoritmo procede definendo 3 variabili, d_x, d_y e d_a, rispettivamente le distanze, dalla radice di `parents`, da x, y e a; tali distanzi vengono determinate utilizzando la funzione `distanceRoot`, definita alla riga 16, che ricorsivamente risale i nodi padri di v in input, fino a trovare la radice (caso base della ricorsione, riga 17), restituendo 0 in tal caso, ed accumulando +1 (riga 20) ogni volta che si decrementa la profondità delle chiamate ricorsive.

Allora, per restituire la distanza tra x ed y, sarà sufficiente calcolare

$$d_x + d_y - 2 \cdot d_a$$

come descritto nella riga 29, poiché $d_x + d_y$ è una somma di distanze che conterrà 2 volte la distanza tra la radice, e l'antenato, comune ad x ed y, a loro più vicino (di fatto, è l'intersezione dei cammini tra radice ed x, e radice ed y). Si noti che tale formula risulta essere sempre corretta, poiché `parents` è un array di padri, il quale rappresenta sempre un albero/arborescenza del grafo di partenza, e dunque per definizione la struttura che si sta utilizzando è aciclica, garantendo che l'unico modo per trovare la distanza tra due vertici è proprio attraverso la formula utilizzata dall'algoritmo.

Osservazione 1.5.1.2 (Costo dell'algoritmo)**.** La funzione `findCommonAncestor` esegue 2 cicli `while`, i quali entrambi visitano ogni nodo di `parents`, avente esattamente n nodi, una ed una sola volta; allora, il costo della funzione risulta essere $O(n) + O(n) = O(n)$.

La funzione `distanceRoot` risale ricorsivamente l'albero `parents`, e nel caso peggiore dovranno essere ripercorsi tutti i nodi padri del grafo; allora, il suo costo è $O(n)$.

Infine, `findDistance` effettua una chiamata a `findCommonAncestor`, e successivamente 3 chiamate a `distanceRoot`, e dunque il costo totale dell'algoritmo risulta essere $O(n) + 3 \cdot O(n) = 4 \cdot O(n) = O(n)$.

Algoritmo 1.5.1.2 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 1.5.1.3 (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 `distances[`v`]` $= -1$), 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 (`parents[`v`]`$, v$) stesso.

Osservazione 1.5.1.4 (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.dequeue()
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
```

Dimostrazione. Per dimostrare la correttezza dell'algoritmo, è necessario dimostrare che l'array `distances` in output sia corretto, e dunque che

$$\forall v \in V(G) \quad \texttt{distances[}v\texttt{]} = \text{dist}(u, v)$$

dove u è la radice in input. La dimostrazione procede dunque per induzione sull'array `distances`, come segue:

- *caso base*
  - `distances[`u`]` $= 0$ e questo è garantito dalla riga 4, all'interno della quale vengono inizializzate tutte le distanze dei vertici del grafo, dalla radice u, a 0; si noti che l'algoritmo si assicura di non modificare la radice ulteriormente, grazie al controllo della riga 9, che procede solo se il vertice correntemente in esame non è la radice
- *ipotesi induttiva forte*
  - $\forall v \in V(G) \mid \text{dist}(u, v) =: k \quad$ `distances[`v`]` $= k$
- *passo induttivo*
  - è necessario dimostrare che

$$\forall v \in V(G) \mid \text{dist}(u, v) =: k+1 \quad \texttt{distances[}v\texttt{]} = k+1$$

  - per la Definizione 1.5.1.1, $\forall x, y \in V(G) \quad \text{dist}(x, y)$ è il numero minimo di archi che costituiscono un cammino $x \to y$; allora, affinché v sia a distanza $k+1$ dalla radice u, deve necessariamente esistere un cammino $u \to v$ contenente esattamente $k+1$ archi, per definizione
  - sia $\hat{v}$ il penultimo vertice del cammino $u \to v$ (l'ultimo è proprio v): esso avrà esattamente $(k+1)-1 = k$ vertici, e poiché il cammino preso in considerazione aveva un numero di archi minimo, rimuovendo l'arco $(\hat{v}, v)$ da quest'ultimo, si otterrà un cammino ancora minimo, che definisce di conseguenza la distanza $\text{dist}(u, \hat{v}) = k$
  - allora, su $\hat{v}$ è possibile applicare l'ipotesi induttiva, per la quale si ha che

$$\texttt{distances[}\hat{v}\texttt{]} = k$$

    e poiché $\hat{v} \sim v$, allora l'arco $(\hat{v}, v)$ verrà percorso all'interno della visita in BFS (grazie alla riga 8), e dunque alla riga 11 si otterrà che

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

    per ipotesi induttiva.

□

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 0, ma 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:

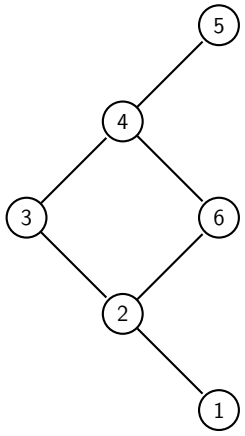


Figura 1.26: Un grafo indiretto.

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 $O(n + m)$.

Osservazione 1.5.2.3 (Archi non della visita)**.** Sia G un grafo, e sia H un albero/arborescenza di visita in BFS di G, radicato in un certo $r \in V(G)$; allora

$$\forall (x, y) \in E(G - H) \quad \operatorname{dist}(r, x) = \begin{cases} \operatorname{dist}(r, y) \pm 1 \\ \operatorname{dist}(r, y) \end{cases}$$

Esempio 1.5.2.1 (Archi non della visita)**.** Ad esempio, si consideri il seguente grafo indiretto G:

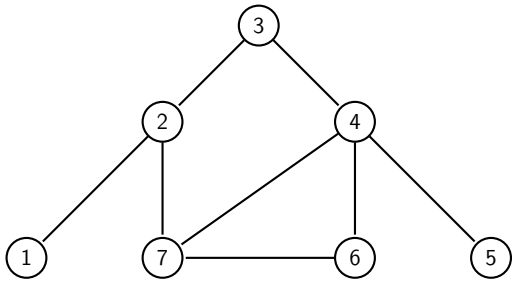


Figura 1.27: Un grafo indiretto.

eseguendo una visita in BFS su di esso, radicata in 3, si potrebbe ottenere il seguente albero T:

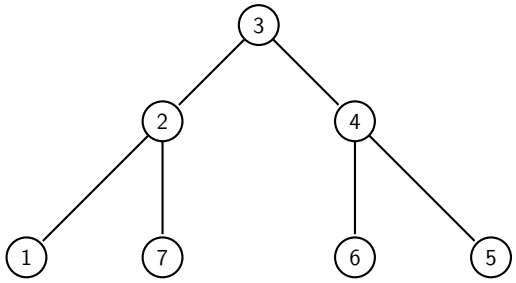


Figura 1.28: Un grafo indiretto.

allora, si noti che

$$E(G - T) = \{(4, 7), (6, 7)\}$$

ed infatti si ha che

$$\text{dist}(3, 7) = 2 = 1 + 1 = \text{dist}(3, 4) + 1$$

$$\text{dist}(3, 6) = 2 = \text{dist}(3, 7)$$

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.dequeue()
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 `distances[`y`]` sia proprio pari a `distances[`v`]` $+ 1$, poiché in tal caso il cammino corrente è un cammino minimo verso y, e dunque 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

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

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 $\text{dist}(X, Y)$.
Input: G grafo, rappresentato attraverso liste di adiacenza; $X, Y \subseteq V(G)$ sottoinsiemi di vertici di G.
Output: $\text{dist}(X, Y)$.

```
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.dequeue()
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 d
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 $Y \subseteq V(G) \implies |Y| \leq |V(G)| =: n \implies O(n+m) + O(|Y|) = O(n+m)$.

Capitolo 2

Algoritmi greedy

2.1 Algoritmi greedy

2.1.1 Definizioni

Definizione 2.1.1.1 (Algoritmi 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.

2.2 Distanza pesata

2.2.1 Archi pesati

Definizione 2.2.1.1 (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 2.2.1.1 (Grafo indiretto pesato)**.** Ad esempio, si consideri il seguente grafo indiretto:

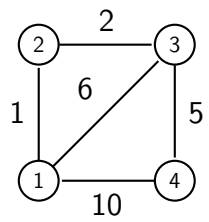


Figura 2.1: Un grafo indiretto pesato.

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 2.2.1.2 (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 2.2.1.3 (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 2.2.1.2 (Grafo indiretto pesato)**.** Ad esempio, si consideri il seguente grafo indiretto pesato:

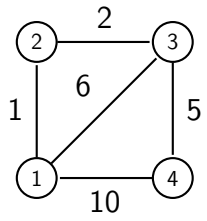


Figura 2.2: Un grafo indiretto pesato.

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

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

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

iii) $\forall x, y, z \in V(G) \quad \text{dist}_w(x, y) \leq \text{dist}_w(x, z) + \text{dist}_w(z, y)$*, 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 2.2.1.1 (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 2.2.1.2 (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*

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

allora necessariamente

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

Dimostrazione. Per ipotesi, u è un vertice adiacente a v, attraverso un arco $(v, u) \in E(G)$, avente peso minimo tra gli adiacenti a v; 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 2.2.1.1 (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*

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

allora necessariamente

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

Dimostrazione. Per ipotesi, (x, u) è un arco tale da minimizzare $\mathrm{dist}_w(v, x) + w(x, u)$; 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 2.2.1.2;
- 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).

□

2.2.2 Algoritmo di Dijkstra

Algoritmo 2.2.2.1 Dato un grafo pesato G attraverso w, e $u \in V(G)$ un suo vertice, l'algoritmo restituisce le distanze pesate dei vertici di G da u; l'algoritmo assume che ogni vertice di G sia raggiungibile da u, e che ogni peso definito da w sia positivo.
Input: G grafo, e u un suo vertice tale che per ogni $x \in V(G)$ esiste un cammino della forma $u \to x$; w una funzione che associa pesi agli archi in $E(G)$, tale che ogni peso sia positivo.
Output: le distanze pesate dei vertici di G da u.

1: **function** DIJKSTRA$_1$(G, w, v)
2: `distancesW` $:=$ `[`$+\infty$`]` $* n$ ▷ valore sufficientemente grande
3: `distancesW[`v`]` $= 0$
4: $R := \{v\}$
5: **while** $\exists(\alpha, \beta) \in E(G) \mid \begin{cases} \alpha \in R \\ \beta \in V(G) - R \end{cases}$ **do**
6: $(x, u) \in \underset{\substack{(a,b)\in E(G) \\ a\in R \\ b\in V(G)-R}}{\arg\min}$ (`distancesW[`a`]` $+ w(a, b)$)
7: `distancesW[`u`]` $=$ `distancesW[`x`]` $+ w(x, u)$
8: $R = R \cup \{u\}$
9: **end while**
10: **return** `distancesW`
11: **end function**

Osservazione 2.2.2.1 (Correttezza dell'algoritmo)**.** L'algoritmo inizia definendo, alla riga 2, l'array `distancesW`, interamente inizializzato a $+\infty$, che dovrà essere riempito dall'algoritmo e restituito al termine dell'esecuzione; inoltre, alla riga 3 viene definito un insieme di vertici $R \subseteq V(G)$.

Alla riga 5 viene istanziato un ciclo `while`, il quale procede fintanto che è presente un arco uscente da R, dunque composto da un vertice in R ed uno in $V(G) - R$; al suo interno, alla riga 6, viene preso un arco (x, u), uscente da R, tale da minimizzare la somma

$$\texttt{distancesW[}x\texttt{]} + w(x, u) = \text{dist}_w(v, x) + w(x, u)$$

e si noti che, per il Teorema 2.2.1.1, si ha che

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

allora, viene aggiornato il valore di `distancesW[`u`]` coerentemente, alla riga 7; infine, alla riga 8, viene aggiunto u ad R. Infine, si noti inoltre che il `while`

della riga 5 prosegue, fintanto che esistono ancora archi uscenti da R, e dunque l'algoritmo termina quando R contiene ogni vertice del grafo.

Si noti che l'algoritmo, ad ogni iterazione, sceglie dunque un arco (x, u) tale da minimizzare il peso del cammino $\{v \to x\} \cup (x, u)$, e al termine di ogni iterazione del ciclo `while` viene inserito u all'interno di R, estendendo l'insieme di vertici visitati; di conseguenza, il comportamento dell'algoritmo è greedy. Si noti che, per trovare un tale arco è necessario controllare ogni (x, u) uscente da R.

Osservazione 2.2.2.2 (Costo dell'algoritmo)**.** Per trovare l'arco cercato, alla riga 7, è necessario controllare ogni arco uscente da R, e trovare quello tale da minimizzare la somma richiesta; si noti che, per il Lemma 1.1.1.1, si ha che

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

e dunque la ricerca da effettuare per l'i-esima iterazione ha costo $O(m)$; si noti inoltre che il ciclo `while` prosegue fino all'esaurimento degli archi uscenti da R, e dunque fino al termine della visita completa del grafo.

Allora il costo del ciclo, e dunque dell'algoritmo, è pari a $O(n \cdot m)$.

Algoritmo 2.2.2.2 Dato un grafo pesato G attraverso w, e $u \in V(G)$ un suo vertice, l'algoritmo restituisce le distanze pesate dei vertici di G da u; l'algoritmo assume che ogni vertice di G sia raggiungibile da u, e che ogni peso definito da w sia positivo.
Input: G grafo, e u un suo vertice tale che per ogni $x \in V(G)$ esiste un cammino della forma $u \to x$; w una funzione che associa pesi agli archi in $E(G)$, tale che ogni peso sia positivo.
Output: le distanze pesate dei vertici di G da u.

```
1:  function DIJKSTRA2(G, w, u)
2:      distancesW := [+∞] * n                    ▷ valore sufficientemente grande
3:      distancesW[u] = 0
4:      MinHeap H := []
5:      for v ∈ V(G) do
6:          H.insert(v, distancesW[v])  ▷ le chiavi sono le distanze pesate da u
7:      end for
8:      while !H.isEmpty() do
9:          v := H.extract_min()                  ▷ il minore viene rimosso da H
10:         distancesW[v] = H.get_key(v)
11:         for x ∈ V(G) : { (v,x) ∈ E(G)
                           { H.contains(x)   do
12:             d := distancesW[v] + w(v, x)
13:             if H.get_key(x) > d then
14:                 H.set_key(x, d)
15:             end if
16:         end for
17:     end while
18:     return distancesW
19: end function
```

Osservazione 2.2.2.3 (Correttezza dell'algoritmo)**.** L'algoritmo inizia alla riga 2, ponendo a $+\infty$ l'array che, al termine dell'algoritmo, conterrà le distanze, pesate su w, tra u ed ogni altro vertice di G; inoltre, alla riga 3 viene posta la distanza pesata $\text{dist}_w(u, u) = 0$, ed alla riga 4 viene definito `H`, un min-heap. Successivamente, alla riga 5, viene istanziato un ciclo `for`, utilizzato per inizializzare il min-heap, ponendo in esso ogni vertice $v \in V(G)$, associando loro i valori

$$\texttt{distancesW[}v\texttt{]} = \text{dist}_w(u, v)$$

Alla riga 8, l'algoritmo inizia un ciclo `while`, che terminerà solamente quando il min-heap diventerà vuoto, ovvero quando ogni elemento al suo interno sarà

stato analizzato; infatti, alla riga 9, con l'operazione `H.extract_min()`, il vertice v che attualmente minimizza la distanza pesata da u, viene *rimosso* dal min-heap. Dopo averlo estratto, alla riga 10 ne viene aggiornato il valore all'interno di `distancesW[`v`]` (si noti che il valore *correntemente* noto della distanza pesata tra u e v è contenuto all'interno della chiave del min-heap, mentre l'array conterrà il valore finale). Alla riga 11, viene istanziato un ciclo `for`, il quale per ogni vertice x uscente da v, ancora contenuto nel min-heap (e dunque ancora non esaminato), ne aggiorna la chiave all'interno di `H` (riga 14), se quest'ultima è maggiore di `distancesW[`v`]`$+w(v,x)$, ovvero $\text{dist}_w(u,v)$ sommata al peso dell'arco (v,x) stesso (righe 12 e 13).

L'algoritmo termina restituendo `distancesW` alla riga 18.

Osservazione 2.2.2.4 (Costo dell'algoritmo)**.** Il primo ciclo `for`, della riga 5, ha costo $O(n \log n)$, poiché contiene la riga 6, all'interno della quale viene utilizzata la funzione `H.insert()`, che ha costo $O(\log n)$.

Il ciclo `for` della riga 11, contiene le righe 11, 13, 14, all'interno delle quale vengono utilizzate rispettivamente le funzioni `H.contains()`, `H.get_key()` e `H.set_key()`, che hanno tutte costo $O(\log n)$; inoltre, tale ciclo viene effettuato per ognuno dei vertici adiacenti uscenti da x, e dunque il suo costo è pari a $O\left(\sum_{x \in N(v)} 3 \log n\right) = \deg(v) \cdot O(\log n)$, dove con $N(v)$ si sta indicando l'insieme

$$N(v) := \{x \in V(G) : (v,x) \in E(G)\}$$

degli archi adiacenti uscenti da v, e dunque si ha che $|N(v)| = \deg(v)$.

Infine, il ciclo `while` della riga 8, poiché contiene le rige 9 e 10, che utilizzano rispettivamente le funzioni `H.extract_min()` ed `H.get_key()`, le quali hanno entrambe costo $O(\log n)$, ha costo pari a $O\left(\sum_{v \in V(G)} 2 \log n + \deg(v) \cdot \log n\right) =$ $O\left(\sum_{v \in V(G)} \log n\right) + O\left(\sum_{v \in V(G)} \deg(v) \cdot \log n\right) = O(n \log n) + \log n \cdot O(m)$.

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

2.3 Intervalli

2.3.1 Trovare intervalli disgiunti

Algoritmo 2.3.1.1 Data una lista di intervalli, l'algoritmo 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.

```
1:  function findIntervals(I)
2:      I.sort(key=λ(i) : i → i.right())        ▷ I ordinato sugli estremi destri
3:      Sol := []
4:      for i ∈ I do
5:          b_f := Sol.last().right()           ▷ estremo destro di Sol.last()
6:          if i.left() > b_f then              ▷ equivalente a Sol ∩ i = ∅
7:              Sol.append(i)
8:          end if
9:      end for
10:     return Sol
11: end function
```

Dimostrazione. Siano $\texttt{Sol}_0, \ldots, \texttt{Sol}_n$ 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 $\left.\begin{array}{l} [a_j, b_j] \in \texttt{Sol}_k \subseteq \texttt{Sol}^* \\ [a_j, b_j] \cap [a_{k+1}, b_{k+1}] \neq \varnothing \end{array}\right\} \implies [a_{k+1}, b_{k+1}] \notin \texttt{Sol}_{k+1} \lightning$

– 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 $\texttt{Sol}^* \not\subseteq \texttt{Sol}_n \iff$ per ogni soluzione ottimale $\texttt{Sol}^*$ si verifica una delle seguenti:

– $\texttt{Sol}^* \cap \texttt{Sol}_n = \varnothing$

– $\texttt{Sol}^* \cap \texttt{Sol}_n \neq \varnothing \wedge \texttt{Sol}^* \not\subseteq \texttt{Sol}_n \wedge \texttt{Sol}_n \not\subseteq \texttt{Sol}^*$

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

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 $\lightning$.

□

Osservazione 2.3.1.1 (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 2.3.1.2 (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 (per confronti) 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 $O(n \log n) + O(n) = O(n \log n)$.

2.3.2 Trovare insieme non disgiunto

Algoritmo 2.3.2.1 Data una lista di intervalli, l'algoritmo restituisce l'insieme, di cardinalità minima, di interi $x_1, \ldots, x_k$, tali da intersecarsi con ogni intervallo.
Input: I lista di intervalli di numeri reali della forma $[a, b]$, con $a, b \in \mathbb{R}$.
Output: l'insieme minimo di $x_1, \ldots, x_n$ tali che $\forall i \in I \quad i \cap \{x_1, \ldots, x_n\} \neq \varnothing$.

```
1:  function findIntersectionPoints(I)
2:      I.sort(key=λ(i) : i → i.right())        ▷ I ordinato sugli estremi destri
3:      Sol := ∅
4:      for i ∈ I do
5:          b_f := Sol.last().right()           ▷ estremo destro di Sol.last()
6:          if i.left() > b_f then              ▷ equivalente a Sol ∩ i = ∅
7:              Sol = Sol ∪ {i.right()}
8:          end if
9:      end for
10:     return Sol
11: end function
```

Dimostrazione. La struttura della dimostrazione di questo algoritmo è analoga a quella dell'Algoritmo 2.3.1.1; è dunque omessa la premessa iniziale, e la dimostrazione procede assumendo le stesse condizioni di partenza per l'induzione.

> *Prima implicazione.* È riportata di seguito la dimostrazione a partire dal passo induttivo.

- *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 6, si ha che

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

– si noti che, nel primo caso, si ha che $\texttt{Sol}_{k+1} = \texttt{Sol}_k$, e dunque $\exists \texttt{Sol}^*$ soluzione ottimale $\mid \texttt{Sol}_{k+1} = \texttt{Sol}_k \subseteq \texttt{Sol}^*$ per ipotesi induttiva

– allora sia $[a_{k+1}, b_{k+1}]$ tale che $\texttt{Sol}_k \cap [a_{k+1}, b_{k+1}] = \varnothing \implies \texttt{Sol}_{k+1} = \texttt{Sol}_k \cup \{b_{k+1}\}$

– sia $\texttt{Sol}^*$ una soluzione ottimale tale da soddisfare l'ipotesi induttiva; si noti che $b_{k+1} \in \texttt{Sol}^* \implies \exists \texttt{Sol}^*$ soluzione ottimale $\mid \texttt{Sol}_{k+1} = \texttt{Sol}_k \cup \{b_{k+1}\} \subseteq \texttt{Sol}^*$ per ipotesi induttiva

– sia allora $b_{k+1} \notin \texttt{Sol}^*$; si noti che, poiché $\texttt{Sol}^*$ è una soluzione ottimale, necessariamente

$$b_{k+1} \notin \texttt{Sol}^* \iff \exists x \in \texttt{Sol}^* \mid x \in [a_{k+1}, b_{k+1}]$$

inoltre, si noti che $x \notin \texttt{Sol}_k$, poiché

$$\left.\begin{aligned} b_{k+1} &\in \texttt{Sol}_{k+1} \\ x &\in \texttt{Sol}_k \\ x &\in [a_{k+1}, b_{k+1}] \end{aligned}\right\} \implies \texttt{Sol}_k \cap [a_{k+1}, b_{k+1}] \neq \varnothing \lightning$$

allora, poiché per ipotesi induttiva si ha che $\texttt{Sol}_k \subseteq \texttt{Sol}^*$, necessariamente $x \in \texttt{Sol}^* - \texttt{Sol}_k$

– sia $j \in [1, n) \mid x \in [a_j, b_j]$; allora, si verifica uno dei seguenti casi:

  * $j \leq k \implies \exists \hat{x} \in \texttt{Sol}_k \mid \hat{x} \in [a_j, b_j]$ poiché doveva essere già stato analizzato dall'algoritmo

  * $j > k + 1 \implies b_j > b_{k+1}$; si noti inoltre che l'intervallo $[a_j, b_j]$ è stato scelto tale che $x \in [a_j, b_j]$, e poiché $x \in [a_{k+1}, b_{k+1}]$ per sua definizione, si verifica necessariamente che

$$\left.\begin{aligned} x \in [a_j, b_j] \cap [a_{k+1}, b_{k+1}] &\neq \varnothing \\ b_j &> b_{k+1} \end{aligned}\right\} \implies a_j \leq x \leq b_{k+1} \leq b_j$$

  in particolare, si noti che $b_{k+1} \in [a_j, b_j]$, e poiché la dimostrazione non dipende dalla scelta di $x \in \texttt{Sol}^* - \texttt{Sol}_k$, né da $j \in [1, n)$, si ha che b_{k+1} copre ogni intervallo coperto da x; allora, per far si che esista soluzione ottimale $\texttt{Sol}^*$ tale da contenere $\texttt{Sol}_{k+1} = \texttt{Sol}_k \cup \{b_{k+1}\}$, è sufficiente considerare

$$(\texttt{Sol}^* - \{x\}) \cup \{b_{k+1}\}$$

Seconda implicazione. $\exists \texttt{Sol}^*$ soluzione ottimale $\mid \texttt{Sol}^* \subseteq \texttt{Sol}_n$. La dimostrazione è analoga a quella dell'Algoritmo 2.3.1.1, ed è dunque omessa.

□

Osservazione 2.3.2.1 (Correttezza dell'algoritmo)**.** L'algoritmo inizia ordinando gli intervalli in I, attraverso il loro estremo destro, e definendo successivamente un insieme `Sol` che conterrà la soluzione cercata.

Alla riga 4, viene istanziato un ciclo `for`, che per ogni intervallo $i \in I$, ne inserisce l'estremo destro all'interno di `Sol`, se e solo tale i non si interca con l'ultimo intervallo contenuto attualmente in `Sol` stesso.

Si noti che la scelta dell'estremo destro garantisce di trovare una soluzione ottimale, poiché gli intervalli sono ordinati attraverso il loro estremo destro, ed è dunque necessario utilizzare gli estremi destri degli intervalli anche per trovare i valori da inserire in `Sol`, altrimenti non sarebbe garantito che quest'ultimo abbia cardinalità minima.

Osservazione 2.3.2.2 (Costo dell'algoritmo)**.** L'ordinamento della riga 2 non può essere eseguito con costo minore di $O(n \log n)$; inoltre, il ciclo `for` della riga 4 esegue linearmente una sola iterazione, per ogni intervallo in I, e dunque il suo costo è pari a $O(n)$.

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

2.4 Minimum Spanning Tree (MST)

2.4.1 Definizioni

Definizione 2.4.1.1 (MST)**.** Sia G un grafo indiretto pesato; allora, si definisce *MST (Minimum Spanning Tree)* un sottografo di G, connesso, di peso minimo, contenente ogni vertice di G.

Esempio 2.4.1.1 (MST)**.** Ad esempio, si consideri il seguente grafo indiretto pesato:

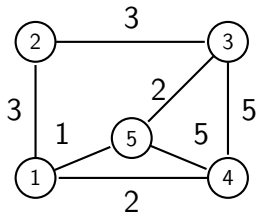


Figura 2.3: Un grafo indiretto pesato.

di esso, un possibile MST è il seguente:

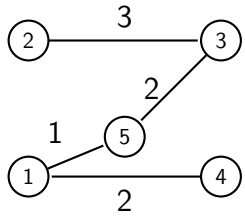


Figura 2.4: Un MST del grafo pesato.

Lemma 2.4.1.1 (MST aciclici)**.** *Sia G un grafo indiretto pesato, con pesi strettamente positivi, e sia H un suo MST; allora H è aciclico.*

Dimostrazione. Per assurdo, sia H ciclico; allora, H contiene almeno un ciclo, e per definizione ogni vertice di tale ciclo avrà grado almeno pari a 2; allora, rimuovendo un arco (u, v) dal ciclo di H, si ottiene un grafo H' con le seguenti caratteristiche:

- $\deg(u)$ e $\deg(v)$ saranno decrementati di 1, e poiché $\deg(u), \deg(v) \geq 2$, allora in H' si ha $\deg(u), \deg(v) \geq 1$; allora, segue che H' è ancora connesso, e che $V(H') = V(H) = V(G)$;
- sia w_H il peso di H, e $w_{H'}$ il peso di H'; allora si ha che

  $$w_{H'} = w_H - w(u, v) \implies w_{H'} < w_H$$

  poiché $w(u, v) > 0$ in ipotesi;

allora, segue che H' è un MST, ma poiché $w_H \neq w_{H'}$, allora H scelto in ipotesi non aveva peso minimo $\lightning$. □

Lemma 2.4.1.2 (Unicità dell'MST)**.** *Sia G un grafo indiretto connesso, pesato attraverso una funzione w che associa un peso ad ogni arco, tale che w sia iniettiva; allora, esiste un unico MST di G.*

Dimostrazione. Omessa. □

2.4.2 Algoritmo di Kruskal

Algoritmo 2.4.2.1 Dato un grafo indiretto connesso G, pesato attraverso w con pesi strettamente positivi, l'algoritmo ne restituisce un MST.
Input: G grafo indiretto connesso; w una funzione che associa pesi, strettamente positivi, agli archi in $E(G)$.
Output: un MST di G.

```
1:  function KRUSKAL(G, w)
2:      E(G).sort(key=λ(e) : e → w(e))          ▷ E(G) ordinato sui pesi di w
3:      Sol := ∅
4:      for e ∈ E(G) do
5:          if findCycle(Sol ∪ {e}) == None then        ▷ se non contiene cicli
6:              Sol = Sol ∪ {e}
7:          end if
8:      end for
9:      return Sol
10: end function
```

Dimostrazione. Per dimostrare la correttezza dell'algoritmo, è necessario dimostrare che l'output sia un MST; allora, $\texttt{Sol}_m$ (si noti che l'algoritmo ha m iterazioni) deve essere un albero, deve essere di copertura, e deve essere minimale rispetto al peso degli archi dato da w; si ha dunque che:

- $\texttt{Sol}_m$ è un albero
  - $\texttt{Sol}_m$ è aciclico, in quanto alla riga 5 vengono aggiunti esclusivamente archi che non formano un ciclo con l'insieme `Sol` di archi già aggiunti
  - per assurdo, sia $\texttt{Sol}_m$ un grafo non connesso, e dunque in esso esiste una componente C, non contenente tutti i vertici in $V(G)$; si noti che, poiché G in input è connesso, allora necessariamente deve esistere un arco
    $$(x, y) \in E(G) \mid \begin{cases} x \in V(C) \\ y \in V(G - C) \end{cases}$$
    si noti inoltre che tale arco non può formare un ciclo in C, poiché $\texttt{Sol}_m$ è aciclico per il punto precedente; allora, per via della riga 5, l'algoritmo avrebbe sbagliato a non selezionare l'arco (x, y)
  - allora, per definizione, `Sol` è un albero
- $\texttt{Sol}_m$ è un albero di copertura

– poiché $\texttt{Sol}_m$ è connesso, e l'algoritmo visita ogni arco di G, anch'esso connesso, non può esistere un vertice non contenuto in $\texttt{Sol}_m$, e dunque l'output è necessariamente un albero di copertura

La struttura della dimostrazione per garantire che $\texttt{Sol}_m$ sia un albero di copertura minimale, è analoga a quella dell'Algoritmo 2.3.1.1, con la sola differenza che le iterazioni del ciclo sono m e non n (si noti il ciclo `for` della riga 4); è dunque omessa la premessa iniziale, e la dimostrazione procede assumendo le stesse condizioni di partenza per l'induzione.

Prima implicazione. È riportata di seguito la dimostrazione a partire dal passo induttivo.

- *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 5, si ha che per ogni $k \in [1, m)$ si ha che

$$\texttt{Sol}_{k+1} = \begin{cases} \texttt{Sol}_k & \texttt{findCycle(Sol} \cup \{e_{k+1}\}\texttt{)} \neq \texttt{None} \\ \texttt{Sol}_k \cup \{e_{k+1}\} & \texttt{findCycle(Sol} \cup \{e_{k+1}\}\texttt{)} = \texttt{None} \end{cases}$$

  - si noti che, nel primo caso, si ha che $\texttt{Sol}_{k+1} = \texttt{Sol}_k$, e dunque $\exists \texttt{Sol}^*$ soluzione ottimale $\mid \texttt{Sol}_{k+1} = \texttt{Sol}_k \subseteq \texttt{Sol}^*$ per ipotesi induttiva
  - allora sia e_{k+1} tale che $\texttt{findCycle(Sol}\cup\{e_{k+1}\}\texttt{)} = \texttt{None} \implies \texttt{Sol}_{k+1} = \texttt{Sol}_k \cup \{e_{k+1}\}$
  - sia $\texttt{Sol}^*$ una soluzione ottimale tale da soddisfare l'ipotesi induttiva; si noti che $e_{k+1} \in \texttt{Sol}^* \implies \exists \texttt{Sol}^*$ soluzione ottimale $\mid \texttt{Sol}_k \cup \{e_{k+1}\} \subseteq \texttt{Sol}^*$ per ipotesi induttiva
  - sia allora $e_{k+1} \notin \texttt{Sol}^*$, e sia $e_{k+1} := (x, y) \in E(G)$ per certi $x, y \in V(G)$; si noti che, poichè $\texttt{Sol}^*$ è una soluzione ottimale, e dunque un MST, per il Lemma 2.4.1.1 è aciclica, e dunque $(x, y) \notin \texttt{Sol}^*$ se e solo se esiste un cammino della forma $x \to y$ in $\texttt{Sol}^*$; allora, necessariamente $\{x \to y\} \cup \{e_{k+1}\}$ è un ciclo in $\texttt{Sol}^* \cup \{e_{k+1}\}$
  - si noti che, essendo $\{x \to y\} \cup \{e_{k+1}\}$ un ciclo parzialmente contenuto in $\texttt{Sol}_k$, allora deve esistere un arco $e_j := (z, y) \in \{x \to y\} - \texttt{Sol}_{k+1}$ per un certo $z \in V(G)$ (di fatto, l'ultimo arco del cammino $x \to y$)
  - si noti che, considerando l'insieme di archi $(\texttt{Sol}^* \cup \{e_{k+1}\}) - \{e_j\}$, il vertice y resta coperto

– si noti che l'arco e_j non è in $\texttt{Sol}_{k+1}$, e dunque $k+1 < j$; allora, per via dell'ordinamento degli archi alla riga 2, necessariamente $w(e_{k+1}) \leq w(e_j)$; dunque, si ha che

$$\sum_{e \in ((\texttt{Sol}^* \cup \{e_{k+1}\}) - \{e_j\})} w(e) \leq \sum_{e \in \texttt{Sol}^*} w(e)$$

ma poiché $\texttt{Sol}^*$, per definizione, è soluzione ottimale, non è possibile trovare una soluzione con peso inferiore, e dunque necessariamente

$$(\texttt{Sol}^* \cup \{e_{k+1}\}) - \{e_j\}$$

ha lo stesso peso di $\texttt{Sol}^*$, che costituisce un MST di G; allora, tale insieme di archi è una soluzione ottimale, contenente $\texttt{Sol}_{k+1}$ per definizione.

Seconda implicazione. $\exists \texttt{Sol}^*$ soluzione ottimale $\mid \texttt{Sol}^* \subseteq \texttt{Sol}_m$. Si noti che, per implicazione precedente, esiste una soluzione ottimale $\texttt{Sol}^\#$ tale che $\texttt{Sol}_m \subseteq \texttt{Sol}^\#$; allora, poichè $\texttt{Sol}_m$ è stato dimostrato essere un MST, e $\texttt{Sol}^\#$ è un MST per definizione, si ha che

$$\texttt{Sol}_m \subseteq \texttt{Sol}^\# \implies \texttt{Sol}_m = \texttt{Sol}^*$$

□

Osservazione 2.4.2.1 (Correttezza dell'algoritmo)**.** L'algoritmo inizia ordinando gli archi di G, in ordine crescente, attraverso i loro pesi, calcolati con la funzione w in input, alla riga 2; questo ordinamento garantisce che l'algoritmo prenderà in considerazione esclusivamente gli archi col peso minore.

Successivamente, alla riga 3 viene istanziato l'insieme `Sol`, che conterrà gli archi che comporranno la soluzione dell'algoritmo (ovvero, un MST di G), e all'interno del ciclo `for` della riga 4, l'algoritmo esegue un controllo (riga 5) per ogni singolo arco $e \in E(G)$, inserendolo all'interno di `Sol` (riga 6), esclusivamente se $\texttt{Sol} \cup \{e\}$ non costituisce un ciclo; questo controllo infatti garantisce che non si vengano a creare cicli all'interno della soluzione, e dunque che l'output sia sicuramente un albero.

L'algoritmo termina restituendo, alla riga 9, l'insieme $\texttt{Sol} \subseteq E(G)$ degli archi che costituiscono un MST di G.

Osservazione 2.4.2.2 (Costo dell'algoritmo)**.** Alla riga 2, gli archi di G vengono ordinati attraverso i pesi forniti dalla funzione w, e dunque il costo dell'operazione è pari a $O(m \log m)$. Si noti però che, nel caso peggiore, G ha ogni vertice connesso

con ogni altro vertice, e dunque $m = n \cdot (n-1) = n^2 - n$, ed è dunque possibile riscrivere il costo dell'operazione nel caso peggiore come segue:

$$O(m \log m) = O(m \log(n^2 - n)) = O(m \log n^2) - O(m \log n) =$$

$$= O(m \log n^2) = O(m \cdot 2 \log n) = O(m \log n)$$

Alla riga 4, viene istanziato un ciclo `for`, che per ognuno degli archi, effettua operazioni in tempo $O(n)$ (si veda l'Algoritmo 1.4.8.1 per la funzione `findCycle`), e dunque ha costo $m \cdot O(n) = O(n \cdot m)$.

Allora, l'algoritmo ha costo $O(m \log n) + O(n \cdot m) = O(n \cdot m)$.

2.4.3 Algoritmo di Prim

Algoritmo 2.4.3.1 Dato un grafo indiretto connesso G, pesato attraverso w con pesi positivi, l'algoritmo ne restituisce un MST.
Input: G grafo indiretto connesso; w una funzione che associa pesi positivi agli archi in $E(G)$.
Output: un MST di G.

1: **function** PRIM$_1$(G, w)
2: $v \in V(G)$
3: `Sol` $:= \varnothing$
4: $R := \{v\}$
5: **while** $R \neq V(G)$ **do**
6: $(x, y) \in \arg\min_{a \in R, b \notin R} w(a, b)$
7: `Sol` $=$ `Sol` $\cup \{(x, y)\}$
8: $R = R \cup \{y\}$
9: **end while**
10: **return** `Sol`
11: **end function**

Dimostrazione. Per dimostrare la correttezza dell'algoritmo, è necessario dimostrare che l'output sia un MST; allora, `Sol`$_n$ deve essere un albero, deve essere di copertura, e deve essere minimale rispetto al peso degli archi dato da w; si ha dunque che:

- `Sol`$_n$ è un albero
  - `Sol`$_n$ è aciclico, in quanto alla riga 7 vengono aggiunti esclusivamente archi con un estremo in R (insieme dei vertici visitati), e l'altro non in

R (si noti la riga 6), e dunque è garantito che l'algoritmo non ripercorra gli stessi vertici più di una volta

  - inoltre, per lo stesso motivo, l'algoritmo non crea componenti non connesse all'interno di $\texttt{Sol}_n$, poiché vengono presi sempre archi che congiungono R con $V(G) - R$, e il grafo in input è connesso, e dunque necessariamente deve esserlo anche $\texttt{Sol}_n$
  - allora, per definizione, $\texttt{Sol}_n$ è un albero

- $\texttt{Sol}_n$ è un albero di copertura

  - poiché il ciclo `while` della riga 5 non termina fino a quando $R = V(G)$, l'algoritmo visita tutti i vertici del grafo, e dunque $\texttt{Sol}_n$ è necessariamente un albero di copertura

La struttura della dimostrazione per garantire che $\texttt{Sol}_n$ sia un albero di copertura minimale, è analoga a quella dell'Algoritmo 2.3.1.1; è dunque omessa la premessa iniziale, e la dimostrazione procede assumendo le stesse condizioni di partenza per l'induzione.

Prima implicazione. È riportata di seguito la dimostrazione a partire dal passo induttivo.

- *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, si ha che

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

  - sia $\texttt{Sol}^*$ una soluzione ottimale tale da soddisfare l'ipotesi induttiva; si noti che $e_{k+1} \in \texttt{Sol}^* \implies \exists \texttt{Sol}^*$ soluzione ottimale $\mid \texttt{Sol}_{k+1} = \texttt{Sol}_k \cup \{e_{k+1}\} \subseteq \texttt{Sol}^*$ per ipotesi induttiva
  - sia allora $e_{k+1} \notin \texttt{Sol}^*$; si noti che, per il Lemma 2.4.1.1, si ha che qualsiasi soluzione ottimale $\texttt{Sol}^*$, che è un MST, deve essere aciclica, e dunque necessariamente $e_{k+1} \notin \texttt{Sol}^* \implies \texttt{Sol}^* \cup \{e_{k+1}\}$ è un ciclo C di $\texttt{Sol}^*$; inoltre, poiché $\texttt{Sol}^*$ è una soluzione ottimale, ed è dunque un MST, il suo insieme di archi deve necessariamente avere il peso minimo possibile, e dunque per trovare una soluzione ottimale tale da includere $\texttt{Sol}_{k+1}$, è necessario garantire che il peso di quest'ultima sia pari a quello di $\texttt{Sol}^*$

– inoltre, necessariamente $C \not\subset R_k$, poiché l'arco e_{k+1}, per via della riga 6, deve essere stato scelto con un estremo in R_k e un estremo in $\texttt{Sol}^* - R_k$; allora, C è solo parzialmente contenuto in R_k, ma poiché C è un ciclo, affinché questo si chiuda deve necessariamente esistere un secondo arco $\hat{e} \in C$, tale da congiungere R_k e $\texttt{Sol}^* - R_k$

– si noti che deve necessariamente verificarsi che

$$w(e_{k+1}) \leq w(\hat{e}) \iff w(e_{k+1}) - w(\hat{e}) \leq 0$$

poiché se $\hat{e}$ avesse avuto peso inferiore a e_{k+1}, l'algoritmo, alla riga 6, lo avrebbe scelto al posto di e_{k+1}

– allora, dalla formula precedente segue che

$$\sum_{e \in ((\texttt{Sol}^* \cup \{e_{k+1}\}) - \{\hat{e}\})} w(e) = \sum_{e \in \texttt{Sol}^*} w(e) + w(e_{k+1}) - w(\hat{e}) \leq \sum_{e \in \texttt{Sol}^*} w(e)$$

e dunque, e possibile considerare l'insieme di archi

$$(\texttt{Sol}^* \cup \{e_{k+1}\}) - \{\hat{e}\}$$

per ottenere una soluzione ottimale tale da contenere $\texttt{Sol}_{k+1}$.

Seconda implicazione. $\exists \texttt{Sol}^*$ soluzione ottimale $\mid \texttt{Sol}^* \subseteq \texttt{Sol}_m$. Si noti che, per implicazione precedente, esiste una soluzione ottimale $\texttt{Sol}^\#$ tale che $\texttt{Sol}_m \subseteq \texttt{Sol}^\#$; allora, poichè $\texttt{Sol}_m$ è stato dimostrato essere un MST, e $\texttt{Sol}^\#$ è un MST per definizione, si ha che

$$\texttt{Sol}_m \subseteq \texttt{Sol}^\# \implies \texttt{Sol}_m = \texttt{Sol}^*$$

□

Osservazione 2.4.3.1 (Correttezza dell'algoritmo)**.** L'algoritmo inizia scegliendo un verice $v \in V(G)$ qualsiasi, e lo inserisce in R alla riga 4, dove quest'ultimo rappresenterà, nelle successive iterazioni l'insieme di vertici correntemente visitati dall'algoritmo; viene inoltre definto `Sol` alla riga 3, che sarà l'insieme di archi costituenti un MST di G.

All'interno del ciclo `while` della riga 5, fintanto che R non è pari a $V(G)$ (si noti dunque che ogni vertice del grafo viene visitato, garantendo che `Sol` copra G, poiché quest'ultimo è un grafo indiretto connesso), viene scelto un arco (x, y), avente peso minore, con un estremo in R, e l'altro in $V(G) - R$ (dunque, un arco che colleghi la regione di vertici già visitati, con quelli ancora da visitare);

ciò garantisce che il grafo in output sia connesso, e poiché viene scelto con peso minimo, è anche garantito che `Sol` sia minimale rispetto ai pesi di w; allora, l'output dell'algoritmo è proprio un MST.

L'algoritmo termina restituendo, alla riga 10, l'insieme di archi trovato.

Osservazione 2.4.3.2 (Costo dell'algoritmo)**.** L'algoritmo effettua un singolo ciclo `while`, alla riga 5, che termina nel momento in cui R contiene ogni vertice di $V(G)$, e poiché ad ogni iterazione viene inserito esattamente un vertice in R (riga 8), allora il ciclo viene ripetuto esattamente n volte.

Inoltre, alla riga 6 viene cercato l'arco, uscente da R, con peso minore, e il costo di tale operazione, nel caso peggiore, è $O(m)$, poiché implica il controllo del peso di ogni singolo arco uscente da R.

Allora, complessivamente l'algoritmo ha costo $n \cdot O(m) = O(n \cdot m)$.

Algoritmo 2.4.3.2 Dato un grafo indiretto connesso G, pesato attraverso w con pesi positivi, l'algoritmo ne restituisce un MST.
Input: G grafo indiretto connesso; w una funzione che associa pesi positivi agli archi in $E(G)$.
Output: un MST di G.

```
1:  function PRIM_2(G, w)
2:      v ∈ V(G)
3:      Sol := ∅
4:      R := {v}
5:      MinHeap H := []
6:      for u ∈ V(G) − {v} do
7:          H.insert(u, +∞)
8:      end for
9:      parents := [−1] ∗ n
10:     parents[v] = v
11:     for y ∈ V(G) : y ∼ v do
12:         parents[y] = v
13:         H.set_key(y, w(v, y))
14:     end for
15:     while R ≠ V(G) do
16:         y := H.extract_min()                ▷ l'elemento viene rimosso
17:         Sol = Sol ∪ {(parents[y], y)}
18:         R = R ∪ {y}
19:         for x ∈ V(G) − R : x ∼ y do
20:             if H.get_key(x) > w(x, y) then
21:                 parents[x] = y
22:                 H.set_key(x, w(x, y))
23:             end if
24:         end for
25:     end while
26:     return Sol
27: end function
```

Osservazione 2.4.3.3 (Correttezza dell'algoritmo)**.** L'algoritmo inizia alla riga 2, scegliendo un vertice qualunque $v \in V(G)$, dal quale partire; alla riga 3, viene definito `Sol`, che contterrà l'insieme di archi dell'MST di G, che verrà restituito alla riga 26 al termine della procedura; alla riga 4, viene inoltre definito un insieme di vertici R, che costituirà la regione di vertici correntemente coperta dall'algoritmo, coerentemente con gli archi presenti `Sol`; infine, alla riga 5, viene inizializzato un min-heap, che viene riempito con le righe 6 e 7, utilizzando tutti i vertici

in $V(G)$, eccetto v scelto in partenza; si noti che ad ogni nodo viene associato inizialmente $+\infty$, poiché per ogni $u \in V(G) - \{v\}$ il criterio di ordinamento del min-heap dovrà essere il peso minimo dell'arco tale da inserire u in R, ed è dunque necessario inizializzare le loro chiavi con un valore sufficientemente alto, tale da non essere mai scelto inizialmente dall'algoritmo.

Alla riga 9 viene definito un array di padri `parents`, il quale non costituirà propriamente un albero di padri della visita del grafo in input, ma un albero i cui padri dei nodi sono sempre i vertici tali da minimizzare l'arco tra padre e figlio; alla riga 10 viene marcata la radice dell'albero ponendo `parents[`v`]` $= v$.

Il ciclo `for` della riga 11, per ogni vertice y adiacente a v di partenza, inizializza `parents[`y`]` $= v$ alla riga 12, ed alla riga 13 viene aggiornato il valore della chiave di y all'interno del min-heap, attraverso proprio il peso dell'arco (v, y).

La procedura dell'algoritmo inizia dunque alla riga 15, definendo un ciclo `while` che termina se R contiene tutti i vertici in $V(G)$, garantendo dunque che l'output `Sol` sarà un grafo di copertura; alla riga 16 viene salvato y, definito come l'elemento minore del min-heap (*si noti che l'operazione* `H.extract_min()` *rimuove il minimo dal min-heap*); inoltre, per definizione di y, per le chiavi usate nel min-heap, e per come `parents` è stato definito, è garantito che l'arco (`parents[`y`]`$, y)$ minimizza il peso, e dunque nella riga 17 quest'ultimo arco viene inserito in `Sol`, ed y viene aggiunto ad R alla riga seguente (si noti che questo assicura che l'algoritmo scelga un insieme di archi minimale rispetto al peso di w).

Successivamente, alla riga 19 viene istanziato un ulteriore ciclo `for`, il quale per ogni vertice x non contenuto in R (dunque ancora non visitato dall'algoritmo), adiacente all'attuale y, effettua un aggiornamento simile a quello eseguito dal ciclo `for` della riga 11: infatti, la riga 20 controlla che il peso dell'arco (x, y) sia minore della chiave di x, attualmente in `H`, poiché in tal caso è necessario aggiornare quest'ultima con $w(x, y)$, dunque minimizzando sempre il peso associato ad x nel min-heap; inoltre, in tal caso va anche aggiornato il padre di x con y, salvando dunque all'interno di `parents[`x`]` il padre di x tale da minimizzare il peso dell'arco tra padre e figlio.

L'algoritmo termina alla riga 26, restituendo l'insieme di archi `Sol` ottenuto.

Osservazione 2.4.3.4 (Costo dell'algoritmo)**.** Si noti che il costo di inserimento di un nodo all'interno di un heap è pari a $O(\log n)$, e dunque il costo del ciclo `for` alla riga 6 è pari a $O(n \log n)$, poichè viene inserito ogni nodo in $V(G) - \{v\}$ all'interno di `H`, e $O(|V(G) - \{v\}|) = O(n - 1) = O(n)$. Inoltre, la creazione dell'array `parents`, alla riga 9, ha costo $O(n)$.

Il costo di aggiornamento di una chiave all'interno di un heap è pari anch'esso a $O(\log n)$, e dunque la complessità del ciclo `for` della riga 11 è pari a $\deg(v) \cdot O(\log n)$, per via della riga 13.

L'operazione di estrazione del minimo dal min-heap, alla riga 16, ha anch'essa costo pari a $O(\log n)$, esattamente come il controllo della riga 20, e l'operazione di aggiornamento della chiave di x nella riga 22; allora, il ciclo `for` della riga 19 ha costo $\deg(y) \cdot O(\log n)$, mentre il costo del ciclo `while` della riga 15 è pari a

$$O\left(\sum_{y \in V(G)-\{v\}} \log n + \deg(y) \cdot \log n\right)$$

Allora, accorpando il costo del ciclo `for` della riga 11, all'interno del costo del ciclo `while` della riga 15, è possibile riscrivere l'equazione precedente come segue:

$$O\left(\sum_{y \in V(G)} \log n\right) + O\left(\sum_{y \in V(G)} \deg(y) \cdot \log n\right) =$$

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

Dunque, il costo dell'algoritmo è pari a $O(n \log n) + O(n) + O(\log n \cdot (n+m)) = O(\log n \cdot (n + m))$.

Capitolo 3

Algoritmi Divide et Impera

3.1 Algoritmi Divide et Impera

3.1.1 Definizioni

Definizione 3.1.1.1 (Divide et Impera)**.** Con la locuzione latina *divide et impera*, ci si riferisce ad una tecnica di risoluzione dei problemi, basata sull'induzione, attraverso la quale:

- il problema di partenza viene inizialmente scomposto in sotto-problemi di dimensione inferiore, ricorsivamente (la fase del *divide*);
- successivamente, una volta raggiunto un caso base della ricorsione, per il quale è nota la soluzione, l'algoritmo procede a combinare le soluzioni dei sotto-problemi, per trovare la soluzione del problema di partenza (la fase dell'*impera*).

Teorema 3.1.1.1 (Master theorem - Teorema principale)**.** *Siano* $\alpha, \beta \geq 1$ *tali da descrivere la seguente equazione di ricorrenza:*

$$\begin{cases} T(n) = \alpha \cdot T\left(\frac{n}{\beta}\right) + f(n) \\ T(1) = \Theta(1) \end{cases}$$

allora, si ha che

- *se* $f(n) = \Theta(n^c)$*, con* $c < \log_\beta \alpha$*, allora*

$$T(n) = \Theta(n^{\log_\beta \alpha})$$

- *se* $f(n) = \Theta(n^c \log^k n)$, *con* $\begin{cases} c = \log_\beta \alpha \\ k \geq 0 \end{cases}$, *allora*

$$T(n) = \Theta(n^{\log_\beta \alpha} \log^{k+1} n)$$

- *se* $f(n) = \Theta(n^c)$, *con* $c > \log_\beta \alpha$, *allora*

$$T(n) = f(n)$$

3.1.2 Trovare la somma massima dei sotto-array

Algoritmo 3.1.2.1 Dato un array `A` di n interi, l'algoritmo restituisce la somma massima tra i suoi sotto-array contigui.
Input: `A` un array di n interi.
Output: la somma massima tra i sotto-array contigui di `A`.

```
1:  function FMSSInternals(A, a, b)
2:      if a == b then
3:          if A[a] ≥ 0 then
4:              return A[a]
5:          else
6:              return 0
7:          end if
8:      else
9:          m := ⌊(a + b)/2⌋
10:         M_s := FMSSInternals(A, a, m)
11:         M_d := FMSSInternals(A, m + 1, b)
12:         t, s := 0
13:         for i ∈ [m, a] do                    ▷ dal centro verso sinistra
14:             t += A[i]
15:             if t > s then
16:                 s = t
17:             end if
18:         end for
19:         t, d := 0
20:         for i ∈ [m + 1, b] do                ▷ dal centro verso destra
21:             t += A[i]
22:             if t > d then
23:                 d = t
24:             end if
25:         end for
26:         return max(M_s, M_d, s + d)
27:     end if
28: end function
29:
30: function findMaxSubarraySum(A)
31:     return FMSSInternals(A, 0, A.length() − 1)    ▷ −1 per indicizzazione
32: end function
```

Osservazione 3.1.2.1 (Correttezza dell'algoritmo)**.** L'algoritmo inizia alla riga 31, chiamando una funzione ricorsiva interna, ovvero `FMSSInternals`, poiché è necessario inizializzarne correttamente i parametri; infatti, ad essa vengono forniti, oltre all'array di interi `A`, anche gli estremi di quest'ultimo, in modo da effettuare la ricorsione sull'intero array.

La funzione ricorsiva inizia controllando se a e b in input sono uguali: questo infatti è il caso base della ricorsione, poiché per tale condizione l'array della chiamata ricorsiva corrente è lungo 1, e dunque ha esattamente 1 elemento; allora, in tal caso è sufficiente restituire `A[`a`]` stesso, se quest'ultimo fosse positivo, mentre se dovesse essere negativo, è necessario restituire 0. Questo, in quanto, nel caso peggiore, `A` è costituito esclusivamente da interi negativi, e dunque il sotto-array contiguo di `A` con somma maggiore risulta essere l'array nullo, che ha sempre somma 0, e dunque la riga 6 garantisce tale risultato in questo caso particolare (per la precisione, si noti che le righe 15 e 22, discusse nella sezione successiva, già garantiscono che il risultato sia 0 nel caso in cui `A` contenesse eslcusivamente interi negativi, con la sola eccezione in cui `A` è costituito da un solo intero, negativo, nel qual caso la ricosione non verrebbe effettuata, e dunque la riga 6 garantisce di restituire 0 ugualmente).

Se invece $a \neq b$, allora non si entra nel caso base, poiché gli estremi del sotto-array da controllare non coincidono; alla riga 9 viene dunque calcolato il punto medio m tra a e b, arrotondato per eccesso, per poter effettuare 2 chiamate ricorsive, nelle righe 10 e 11, grazie alle quali vengono rispettivamente restituiti M_s ed M_d, che costituiscono il primo la somma massima tra i sotto-array compresi in `A[`a`:`m`]`, ed il secondo la somma massima tra i sotto-array compresi in `A[`$m+1$`:`b`]` (considerando la notazione degli *slice* inclusiva su entrambe gli estremi).

Successivamente, dalla riga 12 alla riga 18, viene effettuato un ciclo `for`, che calcola la somma massima della prima metà del sotto-array del livello di ricorsione corrente, dunque partendo dal centro muovendosi verso sinistra; viceversa, il ciclo `for` delle righe successive (dalla 19 fino alla 25), effettua il calcolo della somma massima della seconda metà del sotto-array del livello di ricosione corrente, dunque partendo dal centro muovendosi verso destra. Questo, poiché se entrambe i cicli si muovessero con indici crescenti (dunque da sinistra verso destra), di fatto i due `for` starebbero controllando 2 sotto-array diversi nel livello di ricorsione corrente, e non lo stesso; per ovviare a questo problema, è sufficiente realizzare i cicli come appena descritto. Si noti che all'interno di questi, viene semplicemente istanziato un accumulatore t, che viene aggiornato ad ogni iterazione dei cicli (righe 14 e 21) con l'i-esimo elemento di `A`, e se t diventa maggiore del totale massimo raggiunto correntemente (in un caso rappresentato da s, nell'altro rappresentato da d), lo si aggiorna col valore attuale di t. Si noti che questo garantisce che venga propagato 0 nel caso di array totalmente negativo, poiché s e d vengono aggiornati soltanto se

t diventa maggiore del loro valore corrente, e sono inizializzati entrambi a 0 nelle righe 12 e 19. Infine, si noti che t alla riga 19 deve essere nuovamente inizializzato a 0 per funzionare correttamente.

Concludendo, il caso ricorsivo, alla riga 26, restituisce il massimo tra M_s, M_d e $s + d$, dove quest'ultima costituisce proprio la somma massima del sotto-array corrente.

Osservazione 3.1.2.2 (Costo dell'algoritmo)**.** Il caso base dell'algoritmo è costituito da un controllo, alla riga 2, con costo $\Theta(1)$, e da un secondo controllo, alla riga 3, con costo $\Theta(1)$ anch'esso; allora, il caso base ha costo $\Theta(1)$.

Nel caso ricorsivo, incontrato dall'algoritmo nella riga 8, vengono effettuate 2 chiamate ricorsive, alle righe 10 e 11, che hanno entrambe costo $T\left(\frac{n}{2}\right)$, e successivamente vengono eseguiti due cicli `for`, nelle righe 13 e 19, i quali hanno entrambi costo $\Theta\left(\frac{n}{2}\right) = \Theta(n)$; allora, il costo del caso ricorsivo è pari a $2 \cdot T\left(\frac{n}{2}\right) + \Theta(n)$.

Allora, l'equazione di ricorrenza che rappresenta il costo dell'algoritmo è la seguente

$$\begin{cases} T(n) = T\left(\frac{n}{2}\right) + \Theta(n) \\ T(1) = \Theta(1) \end{cases}$$

Si noti che questa è un'equazione che è possibile risolvere utilizzando il Teorema 3.1.1.1, in quanto

$$\left.\begin{matrix} \alpha = 2 \\ \beta = 2 \end{matrix}\right\} \implies n^{\log_2 2} = n^1 = n$$

e poiché $f(n) = \Theta(n)$, allora ci si trova nel secondo caso del Master theorem, ed infatti ponendo $k = 0$ si ha

$$f(n) = \Theta(n) = \Theta(n \log^0 n)$$

e dunque si ha che

$$T(n) = \Theta(n^{\log_2 2} \log^{0+1} n) = \Theta(n \log n)$$

Capitolo 4

Programmazione dinamica

4.1 Programmazione dinamica

4.1.1 Definizioni

Definizione 4.1.1.1 (Programmazione dinamica)**.** Con *programmazione dinamica*, si intende un processo di risoluzione dei problemi, in cui questi vengono risolti partendo da soluzioni delgli stessi, ma di dimensioni inferiori; si noti che tali sottoproblemi sono sovrapponibili, e questo rende l'approccio fondamentalmente differente rispetto al *divide et impera*.

Definizione 4.1.1.2 (Tempo polinomiale)**.** Un algoritmo è detto essere in *tempo polinomiale* se il suo tempo di esecuzione è limitato superiormente da un'espressione polinomiale della dimensione del suo input.

Definizione 4.1.1.3 (Insieme P)**.** L'*insieme P* è costituito dalla categoria di problemi per i quali esiste un algoritmo deterministico in tempo polinomiale, rispetto alla dimensione del proprio input.

Definizione 4.1.1.4 (NP-completezza)**.** Un problema è detto *NP-completo*, se non è possibile risolverlo deterministicamente in tempo polinomiale, rispetto alla dimensione del suo input.

Esempio 4.1.1.1 (Problema NP-completo)**.** Un esempio di problema attualmente NP-completo è rappresentato dal sudoku, in quanto non sono noti algoritmi di risoluzione in tempo polinomiale, rispetto alla dimensione dell'input, per risolverne uno.

4.2 Memoizzazione

4.2.1 Definizioni

Definizione 4.2.1.1 (Memoizzazione)**.** La *memoizzazione*, nota anche con il termine inglese *memoization*, è una tecnica di programmazione che consiste nel salvare in memoria i valori restituiti da una funzione, in modo da averli nuovamente a disposizione per un uso successivo, senza aver bisogno di ricalcolarli; tale tecnica viene spesso utilizzata per risolvere problemi di programmazione dinamica.

4.2.2 Trovare il massimo spazio allocabile

Algoritmo 4.2.2.1 Data una lista `S` di dimensioni di n file, e una memoria di capacità C, l'algoritmo restituisce il massimo spazio che può essere allocato dai file in `S` sulla memoria; inoltre, i file hanno tutti dimensione inferiore a C.
Input: `S` lista di dimensioni di file; C capacità della memoria, tale che $\forall s_i \in$ `S` $\quad s_i \leq C$.
Output: quantità di spazio massima che è possibile allocare.

```
1:  function maxAllocation(S, C)
2:      T := [[−1] ∗ (C + 1)] ∗ (n + 1)
3:      for k ∈ [0, n] do                          ▷ [0, n] ha n + 1 elementi
4:          T[k] [0] = 0                           ▷ prima colonna a 0
5:      end for
6:      for j ∈ [0, C] do                          ▷ [0, C] ha C + 1 elementi
7:          T[0] [j] = 0                           ▷ prima riga a 0
8:      end for
9:      for k ∈ [1, n] do
10:         for j ∈ [1, C] do
11:             T[k] [j] = T[k − 1] [j]            ▷ lo stesso della riga precedente
12:             v := T[k − 1] [j − s_k] + s_k
13:             if { j ≥ s_k
                   { v > T[k − 1] [j]   then
14:                 T[k] [j] = v
15:             end if
16:         end for
17:     end for
18:     return T[n] [C]
19: end function
```

Osservazione 4.2.2.1 (Correttezza dell'algoritmo)**.** L'algoritmo inizia definendo una matrice `T` di dimensioni $(n+1) \times (C+1)$, inizializzandone ogni cella a -1; successivamente, con il ciclo `for` della riga 3, viene posta ogni cella della prima colonna a 0, e con il ciclo `for` della riga 6 ogni cella della prima riga a 0.

Lo scopo della matrice costruita dall'algoritmo, è di rappresentare, per ogni cella `T[`k`][`j`]`, dati k file aventi dimensione $s_1, \ldots, s_k$, lo spazio massimo che è possibile allocare con un sottoinsieme di tali file, su un disco di capacità j; di conseguenza, il numero di file che è possibile allocare con una memoria di capacità nulla, è sicuramente 0 (dunque la prima colonna è sempre costituita interamente da 0), e la memoria che è possibile allocare con 0 file è sicuramente 0 (dunque la prima riga vede solamente il valore 0). In simboli, si ha che

$$\forall k \in [0, n], j \in [0, C] \quad \texttt{T}[k]\,[0] = \texttt{T}[0]\,[j] = 0$$

Procedendo per gradi, si può notare che, avendo ad esempio $k = 1$, indipendentemente dalla memoria disponibile, non sarà possibile allorare uno spazio superiore a $s_k = s_1$, e dunque segue che

$$\texttt{T}[1]\,[j] = \begin{cases} 0 & j < s_1 \\ s_1 & j \geq s_1 \end{cases}$$

ovvero, la seconda riga della tabella sarà completamente nulla, fino a quando $j = s_1$, valore dal quale la riga sarà interamente pari a s_1.

Successivamente, analizzando il caso più generale, per descrivere la funzione della matrice, è necessario osservare la legge che ne definisce una generica cella:

$$\texttt{T}[k]\,[j] = \begin{cases} \texttt{T}[k-1]\,[j] & j < s_k \\ \max\{\texttt{T}[k-1]\,[j], \texttt{T}[k-1]\,[j-s_k] + s_k\} & j \geq s_k \end{cases}$$

Partendo dal primo caso, se si verifica che $j < s_k$, e dunque la capacità del disco è inferiore della dimensione del k-esimo file, non sarà certo possibile inserire quest'ultimo all'interno della memoria, e dunque in tal caso è sufficiente riportare il valore che la tabella aveva nella cella sovrastante, ovvero `T[`$k-1$`][`j`]`, indicando che il file con dimensione s_k non è stato scelto. Viceversa, nel caso in cui $j \geq s_k$, e dunque vi è spazio all'interno della memoria per il k-esimo file, vengono prese in considerazione 2 possibili scelte, e di queste viene selezionata quella con valore maggiore: in particolare, viene comparata la quantità di memoria allocata senza l'inserimento del k-esimo file (primo argomento di max), con la memoria allocata nel caso in cui il k-esimo file è stato inserito. Quest'ultimo caso è descritto dal secondo argomento di max, controllando all'interno di `T` quanto spazio è possibile allocare con una memoria di dimensione $j - s_k$, potendo scegliere tra un sottoinsieme dei primi $(k-1)$-file; infatti, poiché la dimensione della memoria attuale

è j, ed è noto che $j \geq s_k$, allora all'interno del disco di dimensione j esisteranno certamente 2 regioni di dimensioni, rispettivamente, s_k e $j - s_k$ (nel caso peggiore $j = s_k \implies j - s_k = 0$). Allora, per sapere quanto spazio è possibile allocare nella regione di memoria avente $j - s_k$ unità di spazio, è sufficiente controllare proprio la matrice stessa, che sicuramente avrà raccolto l'informazione in qualche iterazione precedente; infine, è necessario aggiungere s_k dopo aver preso tale quantità di spazio massimo allocabile, per coprire la regione di memoria di dimensione s_k discussa precedentemente.

Tornando alla discussione dell'algoritmo, nelle righe 9 e 10 vengono istanziati due cicli `for`, con i quali verranno attraversate e riempite tutte le celle della matrice `T`; in particolare, alla riga 11 il valore della cella corrrente viene aggiornato con quello della sua sovrastante (il caso in cui $j < s_k$, di fatto scelto come default), e nel caso in cui, alla riga 13, si verificasse invece che $j \geq s_k$, e v (definito alla riga 12, pari al valore discusso precedentemente), è maggiore del valore contenuto nella cella sovrastante, allora la cella corrente viene rimpiazzata con v stesso; si noti che quest'ultimo controllo fonde la definizione per casi della cella generica, ed il max presente nella formula.

Infine, l'algoritmo termina alla riga 18, restituendo `T[`n`][`C`]`, poiché lo scopo iniziale dell'algoritmo era proprio di trovare la dimensione massima che fosse possibile allocare in una memoria di dimensione C, avendo n file aventi dimensioni $s_1, \ldots, s_n$.

Osservazione 4.2.2.2 (Costo dell'algoritmo)**.** L'algoritmo inizializza una matrice `T` di dimensioni $(n+1) \times (C+1)$ alla riga 2, e il costo della sua creazione è dunque $O((n+1) \cdot (C+1)) = O(n \cdot C)$; i cicli `for` delle righe 3 e 6 hanno rispettivamente costo $O(n)$ e $O(C)$.

Il ciclo `for` della riga 9, poiché contiene un ulteriore ciclo `for` annidato alla riga 10, ha costo $O(n \cdot C)$; inoltre all'interno di questi vengono svolte solamente operazioni elementari.

Allora il costo dell'algoritmo è pari a $2 \cdot O(n \cdot C) + O(n) + O(C) = O(n \cdot C)$.

Si noti che l'input contiene n interi $s_1, \ldots, s_n$, e il numero di bit totale per salvarli in memoria è pari a

$$\sum_{i=1}^{n} O(\log s_i) = O\left(\sum_{i=1}^{n} \log s_i\right)$$

ma poiché nel caso peggiore i file hanno tutti dimensione pari a C, allora

$$O\left(\sum_{i=1}^{n} \log s_i\right) = O(n \cdot \log C)$$

poiché C necessita di $O(\log C)$ bit per essere rappresentato; infine, contando anche la dimensione in bit di C in input, si ha che la dimensione dell'input dell'algoritmo è pari a $O((n+1) \cdot \log C) = O(n \cdot \log C)$. Allora, l'algoritmo non ha tempo polinomiale, poiché il suo costo è pari a $O(n \cdot C)$; infatti, tale problema è NP-completo.

Algoritmo 4.2.2.2 Data una lista `S` di dimensioni di n file, una memoria di capacità C, e la matrice costruita attraverso la funzione `fileAllocation` dell'Algoritmo 4.2.2.1, l'algoritmo restituisce un insieme di file che massimizza la memoria allocata; inoltre, i file hanno tutti dimensione inferiore a C.
Input: `S` lista di dimensioni di file; C capacità della memoria, tale che $\forall s_i \in$ `S` $s_i \leq C$; `T` matrice prodotta precedentemente.
Output: un insieme di file che massimizza la memoria allocata.

```
1:  function maxAllocationFiles(S, C, T)
2:      Sol := ∅
3:      j := C
4:      for k ∈ [n, 1] do                          ▷ il ciclo decrementa
5:          if T[k][j] > T[k − 1][j] then
6:              Sol = Sol ∪ {s_k}
7:              j −= s_k
8:          end if
9:      end for
10:     return Sol
11: end function
```

Osservazione 4.2.2.3 (Correttezza dell'algoritmo)**.** L'algoritmo inizia definendo un insieme vuoto `Sol` alla riga 2, che conterrà l'insieme di file al termine della procedura; inoltre, alla riga 3 viene definito j, pari alla capacità del disco C.

Il ciclo `for` della riga 4 scorre la matrice `T` in input seguendo un preciso criterio: partendo dall'ultima riga della matrice (si noti che con $k \in [n, 1]$ si sta indicando un ciclo che parte da n e termina ad 1), il ciclo controlla che la cella corrente, ovvero `T[`k`][`j`]`, sia maggiore della cella direttamente sopra ad essa, ovvero `T[`$k-1$`][`j`]`; in tal caso, l'algoritmo inserisce s_k all'interno della soluzione `Sol` alla riga 6, e alla riga 7 j viene decrementato di s_k;

L'idea alla base di questo ciclo è di partire dall'ultima cella della matrice (ultima riga, ultima colonna, ovvero `T[`n`][`C`]`), salendo le righe in ogni iterazione, spostanosi orizzontalmente ogni volta che la cella sovrastante a quella corrente aveva valore inferiore ad essa; questo, poiché per via di come è stata costruita la

matrice, necessariamente

$$\texttt{T[}k\texttt{][}j\texttt{]} \neq \texttt{T[}k-1\texttt{][}j\texttt{]} \iff \texttt{T[}k\texttt{][}j\texttt{]} > \texttt{T[}k-1\texttt{][}j\texttt{]}$$

e dunque se la cella corrente ha valore maggiore della cella ad essa sovrastante, allora il k-esimo file era stato scelto dall'algoritmo, e va dunque inserito all'interno della soluzione; si noti dunque che poiché è stato preso il k-esimo file, la dimensione della memoria all'iterazione successiva dovrà essere pari a $j - s_k$, altrimenti si lascerebbe lo spazio del k-esimo file.

L'algoritmo termina restituendo la soluzione accumulata, alla riga 10.

Osservazione 4.2.2.4 (Costo dell'algoritmo)**.** L'algoritmo effettua un solo ciclo `for` alla riga 4, che scorre la matrice in input parentdo da $k = n$ fino a $k = 1$, ma si noti che non viene controllata ogni cella di `T`, poiché viene controllata al massimo 1 cella per riga.

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

4.2.3 Knapsack problem

Algoritmo 4.2.3.1 Date due liste `V` e `P`, rispettivamente rappresentanti il valore e il peso di n oggetti, ed uno zaino di capienza C, l'algoritmo restituisce il massimo valore che può essere totalizzato dagli oggetti in input, all'interno dello zaino, senza sforare la sua capienza massima in termini di peso; inoltre, gli oggetti hanno tutti peso inferiore a C.
Input: `V` lista di valori di oggetti; `P` lista di pesi di oggetti; C capienza dello zaino, tale che $\forall p_i \in$ `P` $\quad p_i \leq C$.
Output: valore massimo che è possibile totalizzare.

```
1:  function KNAPSACK(V, P, C)
2:      T := [[−1] ∗ (C + 1)] ∗ (n + 1)
3:      for k ∈ [0, n] do                          ▷ [0, n] ha n + 1 elementi
4:          T[k][0] = 0                            ▷ prima colonna a 0
5:      end for
6:      for j ∈ [0, C] do                          ▷ [0, C] ha C + 1 elementi
7:          T[0][j] = 0                            ▷ prima riga a 0
8:      end for
9:      for k ∈ [1, n] do
10:         for j ∈ [1, C] do
11:             T[k][j] = T[k − 1][j]              ▷ lo stesso della riga precedente
12:             v := T[k − 1][j − p_k] + v_k
13:             if { j ≥ p_k
                   { v > T[k − 1][j]   then
14:                 T[k][j] = v
15:             end if
16:         end for
17:     end for
18:     return T[n][C]
19: end function
```

Osservazione 4.2.3.1 (Correttezza dell'algoritmo)**.** Si noti che l'algoritmo presenta eslcusivamente qualche variazione rispetto all'Algoritmo 4.2.2.1, poiché quello presentato è una sola generalizzazione del problema dell'allocazione dei file, precedentemente discusso; in particolare, variano le seguenti righe:

- riga 12, dove bisogna controllare la $(j - p_k)$-esima colonna della $(k-1)$-riga, ma aggiungere v_k, poiché `T` contiene i valori massimi possibili, e non i pesi (si noti che nel problema precedente pesi e valori erano coincidenti);

- riga 13, in cui varia semplicemente il controllo $j \geq p_k$, per ragione analoga.

Infine, si noti che anche in questa generalizzazione è possibile descrivere un algoritmo in grado di restituire gli oggetti che hanno permesso di raggiungere tale valore massimo, ma il codice sarebbe uguale in tutto e per tutto all'Algoritmo 4.2.2.2, dove l'input dell'algoritmo sarebbero semplicemente i pesi degli oggetti, e viene dunque omesso.

Osservazione 4.2.3.2 (Costo dell'algoritmo)**.** Poiché l'algoritmo è di fatto una sola generalizzazione dell'Algoritmo 4.2.2.1, e il codice è pressochè invariato, il costo è esattamente lo stesso, ovvero $O(n \cdot C)$, ed infatti il problema resta NP-completo.

4.2.4 Trovare il peso massimo di un cammino

Lemma 4.2.4.1 (Pesi di cammini)**.** *Sia G un grafo diretto aciclico, pesato attraverso w, siano $u, v \in V(G)$ due suoi vertici, e sia $z \in V(G) - \{u, v\}$ un suo vertice, tale che $(z, v) \in E(G)$; se esiste un cammino della forma $u \to z$, avente peso α, allora esiste un cammino della forma $u \to v$, avente peso $\alpha + w(z, v)$.*

Dimostrazione. Sia $u \to z$ un cammino da u a z, avente peso α; allora, per trovare un cammino della forma $u \to v$, avente peso $\alpha + w(z, v)$, è sufficiente considerare il cammino $\{u \to z\} \cup (z, v)$, poiché $(z, v) \in E(G)$ in ipotesi.

Si noti che questa dimostrazione non esclude la possibilità che il cammino $u \to z$ sia passante per v; ma, se così fosse, poiché in ipotesi si ha che $(z, v) \in E(G)$, allora $(\{u \to z\} - \{u \to v\}) \cup (z, v)$ è un ciclo di G, e questo non è possibile perché G è stato scelto aciclico in ipotesi $\lightning$. Inoltre, se si rimuovesse l'ipotesi per cui G deve essere aciclico, allora la tesi non sarebbe necessariamente vera, poiché nel caso appena descritto, il percorso $u \to z$ passante per v in realtà non è un cammino ma una passeggiata, e sarebbe dunque falsa l'implicazione. □

Algoritmo 4.2.4.1 Dato un grafo G diretto aciclico, pesato attraverso w con pesi sia positivi che negativi, e due suoi nodi $u, v \in V(G)$, l'algoritmo restituisce il peso massimo che un cammino della forma $u \to y$ può avere.
Input: G grafo diretto aciclico; w funzione dei pesi degli archi; $u, y \in V(G)$ due vertici di G.
Output: peso massimo di un cammino $u \to y$.

```
1:  function maxWeightPaths(G, w, u, y)
2:      T := [[−1] ∗ n] ∗ n
3:      for x ∈ V(G) do
4:          if x == u then
5:              T[0][x] = 0
6:          else
7:              T[0][x] = −∞
8:          end if
9:      end for
10:     for k ∈ [1, n − 1] do
11:         for x ∈ V(G) do
12:             T[k][x] = T[k − 1][x]          ▷ lo stesso della riga precedente
13:             for (z, x) ∈ E(G) do           ▷ z è entrante in x (G è diretto)
14:                 v := T[k − 1][z] + w(z, x)
15:                 if v > T[k][x] then
16:                     T[k][x] = v
17:                 end if
18:             end for
19:         end for
20:     end for
21:     return T[n − 1][y]
22: end function
```

Osservazione 4.2.4.1 (Correttezza dell'algoritmo)**.** La discussione della correttezza qui presentata fa affidamento a quella dell'Algoritmo 4.2.2.1, dunque alcuni dettagli verrano sottointesi poiché l'idea alla base è la medesima.

L'algoritmo inizia alla riga 2, definendo una matrice $n \times n$, inizializzata con ogni cella a -1, e successivamente, attraverso un ciclo `for` alla riga 3, vengono inizializzate tutte le celle della prima riga della matrice a $-\infty$, eccetto per $\mathtt{T}[0][u] = 0$.

In questo algoritmo, lo scopo della matrice è di rappresentare, per ogni generica cella $\mathtt{T}[k][x]$, il peso massimo di un cammino della forma $u \to x$, avente al massimo k archi; allora, naturalmente il peso di un cammino $u \to u$ è sempre 0, mentre per quanto riguarda il resto della prima riga, ad ogni sua cella viene asse-

gnato il valore $-\infty$, poiché l'algoritmo deve trovare il peso massimo di cammini, e dunque tale valore ricopre il ruolo di invariante nei controlli, oltre al fatto che non esiste altro cammino, che inizi da u, avente 0 archi, oltre a $u \to u$ già discusso; inoltre, si noti che il grafo G in input non è necessariamente fortemente connesso, e poiché è diretto, è possibile non vi siano cammini tra due dati nodi di G, dunque in tal caso l'algoritmo propagherà il valore $-\infty$, ad indicare che il valore iniziale non è stato modificato, in quanto non vi era valore con cui sostituire quello di partenza, perché non esiste collegamento tra i due vertici. In simboli

$$\texttt{T[0]}\,[x] = \begin{cases} -\infty & x \neq u \\ 0 & x = u \end{cases}$$

Proseguendo gradualmente, se il numero di archi massimo aumenta a $k = 1$, allora si ha che

$$\texttt{T[1]}\,[x] = \begin{cases} \texttt{T[0]}\,[x] & (u,x) \notin E(G) \\ w(u,x) & (u,x) \in E(G) \end{cases}$$

poiché naturlamente, avendo a disposizione al massimo 1 arco, se l'arco (u, x) stesso è presente in $E(G)$ (si noti che il grafo è diretto), allora il valore della cella sarà assegnato al peso di quest'ultimo, ovvero $w(u, x)$, altrimenti viene lasciato il peso che aveva nella cella sovrastante (si noti che non basta inserre $-\infty$, poiché andrebbe gestito comunque il caso in cui $x = u$, e dunque basta prendere il valore della cella sovrastante per evitare ulteriori controlli).

Spostando l'attenzione al caso generale, la formula che descrive una generica cella della matrice dell'algoritmo è la seguente:

$$\forall k \in [1, n-1] \quad \texttt{T[}k\texttt{]}\,[x] = \max\{\texttt{T[}k-1\texttt{]}\,[x], \{\texttt{T[}k-1\texttt{]}\,[z] + w(z,x) \mid \exists z \in V(G) : (z,x) \in E(G)\}\}$$

Partendo da k, si può notare che tale variabile è definita nell'intervallo $[1, n-1]$, esattamente come procede l'algoritmo per il ciclo `for` della riga 10; infatti il numero di righe della matrice creata è n (riga 2), e la prima è già stata riempita come discusso precedentemente, ma il motivo per cui sono sufficienti esattamente n righe, è che k rappresenta il numero massimo di archi di un cammino da u ad ognuno dei nodi della k-esima riga, e poiché sono cammini, e dunque per definizione in essi non è possibile ripercorrere vertici o archi più di una volta, allora segue che il massimo numero di archi che ognuno dei cammini può avere è $n - 1$, e dunque è sufficiente avere n righe, nell'intervallo $[0, n - 1]$.

Inoltre, la formula definisce che la cella generica, che rappresenta il peso massimo che un cammino $u \to x$ avente al massimo k archi può avere, è pari al valore massimo tra i seguenti:

- $\texttt{T[}k-1\texttt{]}\,[x]$, e se questo dovesse essere stato il valore preso dal max, allora qualsiasi altro cammino $u \to x$ presente in G, con al massimo k archi,

doveva avere necessariamente peso inferiore al peso massimo tra i cammini, correntemente trovati, aventi al massimo $k-1$ archi (si noti che questa eventualità si può verificare in quanto G contiene archi anche negativi, altrimenti se i pesi degli archi fossero stati tutti positivi, aggiungere un arco avrebbe certamente restituito un valore maggiore, o uguale, al peso del cammino);

- $\{\texttt{T}[k-1][z] + w(z,x) \mid \exists z \in V(G) : (z,x) \in E(G)\}$, che rappresenta l'insieme dei pesi dei cammini $u \to z$, aventi al massimo $k-1$ archi (dove z è un nodo adiacente entrante in x), ad ognuno dei quali è stato aggiunto il peso dell'arco (z,x); si noti che è garantito che

$$\texttt{T}[k-1][z] = \alpha \implies w_p(\{u \to x\}) = \alpha + w(z,x)$$

grazie al Lemma 4.2.4.1 (il quale è applicabile, poiché G in input è aciclico).

Allora, l'algoritmo effettua 2 cicli `for` annidati, nelle righe 10 e 11, per poter scorrere l'intera matrice, e come per l'Algoritmo 4.2.2.1, alla riga 12 viene posto come valore di default il primo argomento del max della formula discussa in precedenza; successivamente, viene effettuato un ulteriore ciclo `for`, alla riga 13, il quale per ogni arco (z,x), con z vertice adiacente entrante in x, aggiorna il valore della cella corrente con quello definito alla riga 14, se si verifica la condizione della riga 15, di fatto implementando i controlli per il secondo argomento del max della formula, assegnando a $\texttt{T}[k][x]$ sempre il valore maggiore corrente.

Infine, l'algoritmo termina alla riga 21, restituendo $\texttt{T}[n-1][y]$, poiché l'indicizzazione parte da 0 e l'ultima riga della matrice è l'$(n-1)$-esima, ritornando dunque il peso massimo di un cammino $u \to y$, avente al massimo $n-1$ archi.

Per concludere, si noti che invertendo la condizione della riga 15, si ottiene il noto algoritmo di Bellman-Ford, grazie al quale è possibile trovare il peso minimo che cammini tra due nodi di G possono avere (di fatto, la loro distanza pesata).

Osservazione 4.2.4.2 (Costo dell'algoritmo)**.** L'algoritmo costruisce una matrice $n \times n$ alla riga 2, e la sua costruzione ha dunque costo $O(n^2)$; inoltre, il ciclo `for` della riga 3 ha costo $O(n)$.

Il ciclo della riga 11 scorre ogni nodo del grafo, ma il `for` in esso annidato, della riga 13, cicla solamente sui nodi adiacenti entranti nel vertice correntemente visitato, dunque il costo di questi due `for` annidati è $O\left(\sum_{x \in V(G)} \deg^{in}(x) + 1\right) = O(n+m)$; allora, poiché questi due cicli si trovano in un ulteriore `for` (alla riga 10), che percorre ogni intero $k \in [1, n-1]$, ed ha dunque costo $O(n-1) = O(n)$, segue che il costo di questa sezione dell'algoritmo è $O(n \cdot (n+m))$

Allora, il costo dell'algoritmo è pari a $O(n^2) + O(n) + O(n \cdot (n+m)) = O(n \cdot (n+m))$.

Il grafo G in input, se rappresentato attraverso liste di adiacenza (salvando ad esempio i vertici adiacenti entranti per ogni nodo, facilitando le computazioni del ciclo `for` della riga 13), ha costo spaziale pari a $O(n+m)$, mentre il costo di w è $O(m) = O(n-1) = O(n)$, poiché è sufficiente ad esempio un array, che mappi ogni arco ad un peso. Allora, la dimensione dell'input è pari a $O(n+m)+O(n) = O(n+m)$, ma poiché il costo dell'algoritmo è $O(n \cdot (n+m))$, questo risulta dunque essere in P, poiché ha costo polinomiale rispetto alla dimensione del suo input.

Algoritmo 4.2.4.2 Dato un grafo G diretto aciclico, pesato attraverso w con pesi sia positivi che negativi, due suoi nodi $u, v \in V(G)$, e la matrice costruita attraverso la funzione `maxWeightPaths` dell'Algoritmo 4.2.4.1, l'algoritmo restituisce un cammino, della forma $u \to y$, che realizza tale peso massimo.
Input: G grafo diretto aciclico; w funzione dei pesi degli archi; $u, y \in V(G)$ due vertici di G; `T` matrice prodotta precedentemente.
Output: cammino tale da massimizzare il peso di un cammino $u \to y$.

```
1:  function maxWeightPath(G, w, u, y, T)
2:      Sol := ∅
3:      v := y
4:      k := n − 1
5:      while v ≠ u do
6:          if T[k][v] == T[k − 1][v] then
7:              k − = 1
8:          else
9:              for (z, v) ∈ E(G) do              ▷ z è entrante in v (G è diretto)
10:                 if T[k − 1][z] + w(z, v) == T[k][v] then
11:                     Sol = Sol ∪ {(z, v)}
12:                     v := z
13:                     k − = 1
14:                     break
15:                 end if
16:             end for
17:         end if
18:     end while
19:     return Sol
20: end function
```

Osservazione 4.2.4.3 (Correttezza dell'algoritmo)**.** L'algoritmo inizia definendo un insieme `Sol`, alla riga 2, che conterrà l'insieme degli archi che compongono uno

dei cammini tali da realizzare il peso massimo di un cammino della forma $u \to y$; alla riga 3, viene inoltre definito un vertice v, posto inizialmente pari ad y, per poter iterare successivamente, ed alla riga 3 si ha k, posto pari ad $n-1$, che servirà per puntare alla riga corrente della matrice `T` (partendo dunque dall'ultima riga).

Nella riga 5, viene istanziato un ciclo `while`, il quale, fintanto che v non diventa il vertice di partenza del cammino, ovvero u, visiterà la matrice `T`; si noti che la matrice viene attraversata dal basso verso l'alto, analogamente all'Algoritmo 4.2.2.2.

Se la condizione della riga 6 è verificata, allora aver aggiunto un arco ai $k-1$ archi già presenti nel cammino $u \to v$ non ha contribuito al massimo peso possibile per un tale cammino; dunque, è sufficiente aggiornare k, per andare ad esaminare la riga superiore nella matrice. Al contrario, se il controllo della riga 6 è falso, allora viene istanziato, alla riga 9, un ciclo `for`, il quale per ogni vertice z entrante in v (si noti che G è diretto), controlla se `T[`$k-1$`][`z`]` $+ w(z,v)$ è proprio pari a `T[`k`][`v`]`, ed in tal caso: l'arco (z,v) viene aggiunto a `Sol` (riga 11), v viene aggiornato con il nuovo vertice corrente, ovvero z (si sta percorrendo il cammino da y verso u), k viene decrementato di 1 (riga 13, per andare alla riga superiore di `T`), ed infine viene effettuato un `break` del ciclo `for`; di conseguenza, lo scopo del loop è quello di trovare, tra gli archi entranti in v, quello che realizzava il peso del cammino $u \to v$ avente al massimo k archi.

Al termine del ciclo `while`, l'algoritmo restituisce l'insieme di archi contenuto in `Sol`, che comporranno dunque un cammino della forma $u \to y$, avente il peso massimo cercato.

Osservazione 4.2.4.4 (Costo dell'algoritmo)**.** Partendo dal ciclo `for` della riga 9, esso controlla tutti i nodi adiacenti entranti in v, e nel caso peggiore vanno visitati tutti; inoltre, poiché al suo interno vi sono esclusivamente operazioni in tempo costante, il ciclo ha costo $O(\deg^{in}(v))$.

Infine, tenendo in considerazione la riga 6, che ha costo $O(1)$, si ha che il costo del ciclo `while` della riga 5 è pari a $O\left(\sum_{v \in V(G)} \deg^{in}(v) + 1\right) = O(n+m)$.