Esercizi Grafi

1. Dato un grafo $G = (V, E)$ orientato e memorizzato con un vettore delle adiacenze, modificare l'algoritmo DFS affinché venga restituito il tipo di ogni arco del grafo. Dare:

   (a) Descrizione dell'algoritmo a parole.

   (b) Pseudocodice della soluzione proposta.

   (c) Studio della Complessità in Tempo della soluzione proposta.

   Soluzione:

   (a) All'interno di un grafo orientato abbiamo 4 tipologie di archi:

   1 - Dell'Albero: Sono gli archi dell'albero di copertura generato dalla DFS. Si hanno quando il vertice raggiunto è bianco.

   2 - All'Indietro: Collegano un vertice ad un suo antenato, quindi si hanno quando il vertice raggiunto è grigio.

   3 - In Avanti: Collegano un vertice ad un suo discendente. Non sono archi dell'albero, infatti il vertice raggiunto è nero.

   4 - Trasversali: Sono archi che connetto vertici non imparentati direttamente; questi possono avere un antenato in comune, oppure appartenere a due alberi diversi. Si trovano quando il vertice raggiunto è nero.

   Pro: Differenziamo un arco in avanti da uno trasversale perché, sia u il vertice di partenza e v quello di arrivo, sia d il tempo in cui la DFS ha visto il vertice per la prima volta, se $u.d > v.d$ abbiamo un Arco Trasversale, se invece $u.d < v.d$ allora è un Arco in Avanti.

   Dim: Sia u un nodo dell'albero da cui la DFS sta visitando i vicini e sia v un nodo nero, quindi dove la DFS ha già terminato la visita di tutti i vicini da lui raggiungibili (il Grafo è Orientato). Ci sono allora 3 modi in cui u può relazionarsi rispetto a v:

   1) v è un discendente di u, quindi esiste un cammino di archi nell'albero da u a v e l'arco (u, v) è un arco in Avanti. Quindi v essendo un discendente di u è stato raggiunto per secondo dalla visita DFS, da cui: $v.d > u.d$

   2) u è in un ramo diverso da v, ma condividono un comune antenato; quindi sono "cugini". Qui se v è nero significa che è stato raggiunto per primo dalla DFS e quindi: $u.d > v.d$

   3) u e v sono in alberi diversi. Se v è nero significa che è stato raggiunto per primo dalla DFS e quindi: $u.d > v.d$

   Quindi: Faremo una DFS e durante la fase di visita verso i nodi adiacenti a quello in esame ed andremo a verificare il tipo di arco sfruttando le proprietà sopra elencate.

   (b) **Algoritmo per la Classificazione degli Archi del Grafo**

   Input: Un Grafo $G = (V, E)$ memorizzato con Vettori di Adiacenza

   Output: L'aggiunta del Tipo di Arco nei campi del Vettore degli Archi E

```
1 DFS(Grafo G)
2 for ogni u ∈ G.V do
3 |   u.colore = "White"
4 |   u.predecessore = Null
5 time = 0
6 for ogni u ∈ G.V do
7 |   if u.colore == "White" then
8 |   |   time = DfsVisit(G, u, time)
```

```
1  DfsVisit(Grafo G, Vertice u, int time)
2  time = time + 1
3  u.d = time
4  u.colore"Gray"
5  for ogni v ∈ {G.E[G.V[u]], ..., G.E[G.V[u + 1] − 1]} do
6      sia i l'indice il cui è v nel vettore G.E
7      if v.colore == "White" then
8          v.predecessore = u
9          G.E[i].type = "Albero" // Supponiamo in E ci sia il .type per il tipo di arco
10         time = DfsVisit(G, v, time)
11     else
12         if v.colore == "Gray" then
13             G.E[i].type = "Indietro"
14         else
15             if u.d < v.d then
16                 G.E[i].type = "Avanti"
17             else
18                 G.E[i].type = "Trasversale"
19 u.colore = "Black"
20 time = time + 1
21 u.f = time
   return: time
```

Oss: In $G.E[G.V[u+1]]$ ci sta il primo vicino del nodo u+1, infatti nel for dove scorriamo i vicini di u abbiamo scritto come ultimo indice di $G.E$ in cui guardare: $(G.V[u+1]-1)$.

(c) **Calcolo della Complessità**

L'inizializzazione del colore dei nodi e del predecessore alle righe 1-4 del codice DFS ha costo V. Per calcolare il costo del for alle righe 6-9 del codice DFS, notiamo che tale ciclo si esegue V volte, ma il suo costo dipende dal codice DfsVisit. Nella DfsVisit il costo è dato dal for al suo interno che occupa le righe 4-21. In quel for però non ci sono cicli ulteriori, od altri costi temporali maggiori di $\Theta(1)$, ma solo il richiamo della DfsVisit sui nodi "White" presi dal vettore $G.E$. Sapendo, per via della correttezza dell'algoritmo DFS, che un nodo bianco una volta colorato non può tornare bianco, il costo del for alle righe 6-9 del codice DFS è di:

$$\sum_{v \in V} \#\left(adj(v)\right) = \Theta(E)$$

Infatti per ogni nodo v bianco andiamo a percorrere tutti gli archi a lui connessi e poiché tutti i nodi all'inizio sono bianchi, finché non sono visitati per la prima volta, percorriamo tutti gli archi del grafo, facendo in totale $\Theta(E)$ chiamate ricorsive di DfsVisit.
Allora il costo dell'algoritmo è dato dal costo dell'inizializzazione dei nodi e poi dalla visita:

$$T(n) = \Theta(V) + \Theta(E) = \Theta(V + E)$$

Quindi abbiamo il costo standard della DFS.

Oss: Se fossimo stati su un grafo non orientato avremo avuto solo archi dell'Albero od all'Indietro; infatti in un grafo non orientato non ci sono nodi neri da poter raggiungere, perché se un nodo diviene nero significa che ogni sottografo a lui connesso è stato visitato dalla DFS. Essendo il grafo non orientato non ci sono archi orientati in lui entranti che ne permettano il raggiungimento da parte di altri nodi una volta divenuto nero.

2. Dato un grafo $G = (V, E)$ orientato e memorizzato con una lista delle adiacenze, modificare l'algoritmo DFS affinché dica se nel grafo vi è o no un Pozzo Universale. Dare:

   (a) Descrizione dell'algoritmo a parole.

   (b) Pseudocodice della soluzione proposta.

   (c) Studio della Complessità in Tempo della soluzione proposta.

   Soluzione:

(a) **Def:** Pozzo Universale
In un Grafo Orientato è quel nodo avente grado entrante $|V| - 1$ ed uscente 0.

Oss: Se ve ne è uno in un grafo allora è unico. Infatti se non ha archi uscenti ed archi entranti da parte di tutti gli altri nodi, non possono esservene più di uno.

Descrizione Algoritmo: Si fa la DFS e dentro la DfsVisit si tiene traccia dei vertici uscenti ed entranti del nodo; alla fine della DFS si cerca nella lista dei nodi quel nodo avente grado entrante 0 ed uscente $|V| - 1$.

(b) **Algoritmo per la Ricerca di Pozzi Universali in un Grafo Orientato**

Input: Un Grafo G memorizzato con Lista di Adiacenza

Output: Il nodo facente da Pozzo Universale, se vi è, sennò Null

```
1 DFS(Grafo G)
2 for ogni w ∈ G.V do
3 |   w.colore = "White"
4 |   w.predecessore = null
5 |   w.e = 0 // .e rappresenta il contatore dei vertici entranti
6 |   w.u = 0 // .u rappresenta il contatore dei vertici uscenti
7 for ogni w ∈ G.V do
8 |   if w.colore == "White" then
9 |   |   DfsVisit(G, w)
```

```
1 DfsVisit(Grafo G, Vertice w)  w.colore = "Gray"
2 for ogni v ∈ adj[w] do
3 |   v.e = v.e + 1 // Aggiorno il numero di vertici entranti ed uscenti
4 |   w.u = w.u + 1
5 |   if v.colore == "White" then
6 |   |   v.predecessore = w
7 |   |   DfsVisit(G, v)
8 w.colore = "Black"
```

```
1 Cerca Pozzo(Grafo G)  DFS(G) // Lancio la DFS sul grafo G
2 for ogni w ∈ G.V do
3 |   if w.u == 0  and w.e == (#G.V − 1) then
  |   |   return: w
  return: Null
```

Abbiamo tolto la variabile *time* perché non utile al fine del nostro esercizio.

Oss: A differenza che nell'esercizio precedente qui abbiamo una lista di adiacenza, quindi nella DfsVisit al posto di scrivere:" **for** ogni $v \in \{G.E[G.V[u]], \ldots, G.E[G.V[u+1]-1]\}$ **do**" Abbiamo scritto: " **for** ogni $v \in adj[w]$ **do**"
In entrambi i casi i costo dell'osservazione dei singoli elementi della lista di adiacenza o del vettore delle adiacenze è $\Theta(1)$ e noi non ci dobbiamo occupare dei dettagli implementativi della loro lettura.

(c) **Calcolo della Complessità**

Si calcola come nell'esercizio precedente. Abbiamo anche qui una DFS leggermente modificata, al termine della quale scorriamo l'array V per vedere se c'è un Pozzo Universale; il costo è quello della DFS più quello della ricerca del Pozzo.

$$T(n) = \Theta(V + E) + \Theta(V) = \Theta(V + E)$$

3. Dato un grafo G trovare il suo trasposto G^T.

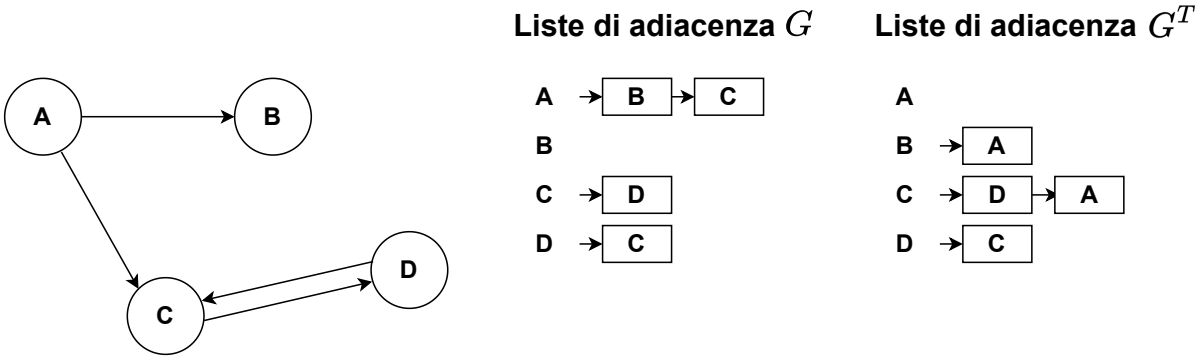
(a) Si consideri un grafo G orientato rappresentato con liste delle adiacenze. Scrivere un algoritmo che calcoli G^T.

(b) Si consideri un grafo G orientato rappresentato con matrice delle adiacenze. Scrivere un algoritmo che calcoli G^T.

(c) Discutere la loro complessità.

Soluzione:

(a) Vediamo un esempio di grafo G rappresentato con le liste delle adiacenze, e la relativa rappresentazione del grafo trasposta G^T. Di seguito un esempio.



Si può osservare che per passare da G a G^T le liste di adiacenza di G sono state "invertite". In particolare, per ogni coppia di nodi $v, u \in V$ viene invertita la relazione che li lega.

Di seguito lo pseudocodice:

```
1 Reverse_List_Adj( G ): G
2 inizializza G^T;
3 for ∀v ∈ G.V do
4 |   for ∀u ∈ G.adj[v] do
5 |   |   G^T.adj[u].append(v);
```

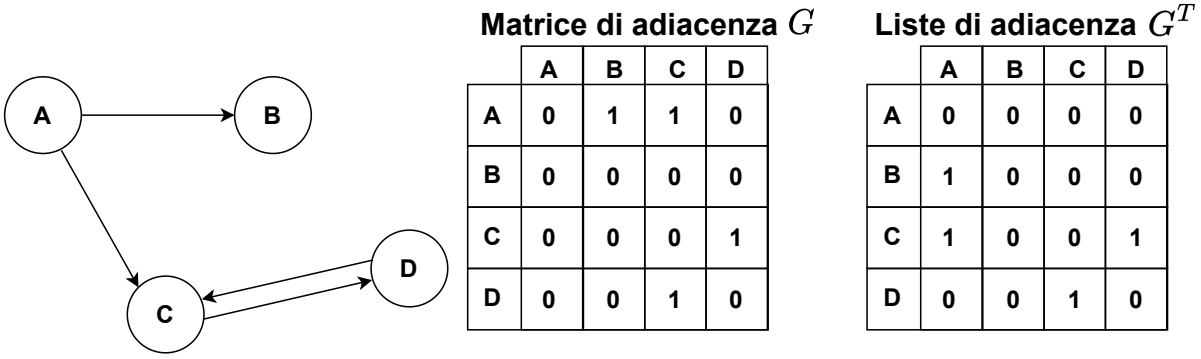
return: G^T

Alcune osservazioni sullo pseudocodice:

- per **inizializza** G^T si intende la creazione di un dizionario le cui chiavi sono i vertici della grafo G, e i cui campi sono vuoti; ovvero ogni lista delle adiacenze ha la sola testa inizializzata.
- La funzione **append()** viene utilizzata per inserire in coda l'ultimo elemento che si è letto nella rispettiva lista.

Per quanto riguarda la complessità il primo ciclo FOR itera sopra ogni vertice di $G.V$, segue che il suo costo è $\mathcal{O}(V)$. Il secondo ciclo FOR visita per ogni nodo tutti i suoi archi uscenti esattamente una volta; pertanto ci costa $\mathcal{O}(E)$. Segue che la complessità finale dell'algoritmo proposto è $\mathcal{O}(V + E)$.

(b) Vediamo come il precedente grafo G viene rappresentato con le matrici delle adiacenze, e la relativa rappresentazione del grafo trasposta G^T. Di seguito un esempio.



	A	B	C	D
A	0	1	1	0
B	0	0	0	0
C	0	0	0	1
D	0	0	1	0

	A	B	C	D
A	0	0	0	0
B	1	0	0	0
C	1	0	0	1
D	0	0	1	0

In questo caso è necessario è necessario scrivere uno pseudocodice che implementa una generica trasposizione di una matrice. Nello specifico, si devono andare a scambiare le colonne della matrice di partenza (G), con le righe della matrice di destinazione (G^T).

Di seguito lo pseudocodice:

```
1 Reverse_Matrix_Adj( G ): G
2 inizializza G^T;
3 for i = 0 to |V| − 1 do
4 |   for j = 0 to |V| − 1 do
5 |   |   G^T[j][i] = G[i][j];
  return: G^T
```

Il costo della soluzione proposta segue dall'analisi dei due FOR. Entrambi ci costano $\mathcal{O}(V)$, quindi visto che sono innestati il costo finale sarà $\mathcal{O}(V^2)$.

(c) Questo punto è stato risolto rispettivamente nei punti a), b).

4. Dato un grafo G trovare il suo **diamentro**.

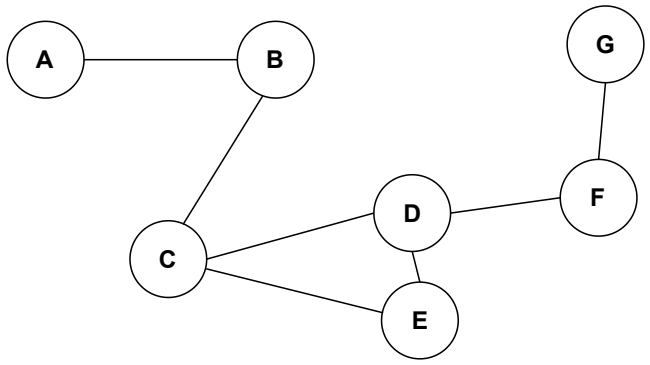
Il diametro di un grafo G è la lunghezza del *più lungo cammino minimo* in G, ovvero la più grande lunghezza in termini di archi attraversati fra tutte le coppie di vertici.

(a) Scrivere un algoritmo che calcoli il *diametro* di un generico grafo G.

(b) Discutere la complessità della soluzione proposta.

Soluzione:

(a) Con lo scopo di costruire un algoritmo che calcoli il diametro di un grafo vediamo un esempio:



Diametro: 5

$P_1 : A \to B \to C \to D \to F \to G$

Q: Perchè il diametro non ha valore 6?

$P_2 : A \to B \to C \to E \to D \to F \to G$

A: il cammino P_2 da A a G non è il cammino minimo.

Al fine di trovare un algoritmo che ritorni il diametro di un generico grafo G, bisogna tenere presente la definizione di diametro sopra riportata. Pertanto al fine di trovare il cammino minimo di lunghezza massima, è possibile sfruttare una delle proprietà della visita in ampiezza BFS.

Proprietà:

La visita in ampiezza costruisce l'albero BFS contenente i cammini minimi dalla sorgente s a tutti i nodi raggiungibili da s. Per i vertici non raggiungibili da s non esiste cammino minimo.

Pertanto per calcolare il *diametro* di un grafo G possiamo sfruttare la visita in ampiezza per calcolare tutti cammini minimi da ogni sorgente a tutti i nodi raggiungibili, e memorizzare il massimo cammino computato.

Di seguito lo pseudocodice:

```
1  get_diametro( G ): G
2  max = −∞;
3  for ∀v ∈ G.V do
4  |   QUEUE Q = QUEUE();
5  |   dist[1, ..., |V|] = −1; dist[v] = 0;
6  |   Q.ENQUEUE(v);
7  |   while Q ≠ ∅ do
8  |   |   u = Q.DEQUEUE();
9  |   |   for ∀k ∈ G.Adj[u] do
10 |   |   |   dist[k] = dist[u] + 1;
11 |   |   |   if dist[k] > max then
12 |   |   |   |   max = dist[k];
13 |   |   |   Q.ENQUEUE(k)
   return: max
```

(b) Per quanto riguarda la complessità in tempo della soluzione proposta partiamo con l'osservare che da righe $4-13$ descrivo una BFS, pertanto segue che il suo costo è $\mathcal{O}(V+E)$. Dato che il primo FOR scorre tutte i nodi in V, abbiamo una BFS ripetuta per ogni nodo del grafo G. Segue che il costo dell'algoritmo è $\mathcal{O}(V^2+VE)$. Tale analisi è stata fatta assumendo di usare liste di adiacenza.

5. Dato un grafo G verificare se 2-colorabile.

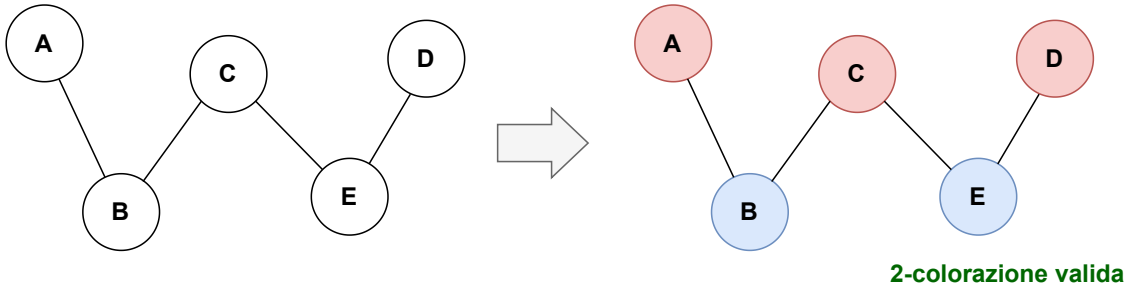
   Dato un grafo G, esso è k-colorabile se ad ogni vertice $v \in V$ è possibile assegnare un colore $C = \{1, \ldots, k\}$ in modo tale che per ogni coppia di nodi adiacenti $i, j \in V$, ovvero $(i, j) \in E$, il colore di i sia diverso dal colore di j.

   (a) Scrivere un algoritmo che dica se un grafo G dato in input è 2-colorabile.

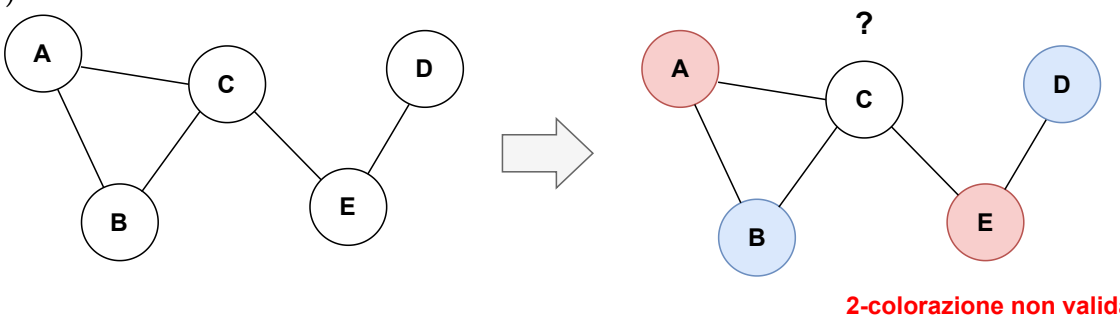
   (b) Discutere la complessità della soluzione proposta.

   Soluzione:

   a)



   b)



   (a) Si deduce dall'esempio in figura che dato il grafo G 2-colorarlo significa assegnare un colore ad ogni nodo in modo che ogni coppia di nodi adiacenti siano di colore diverso.

   Innanzitutto assumiamo che l'oggetto vertice abbia un campo 2COL il quale conterrà il colore del nodo. Il campo 2COL sarà di tipo booleano, dove TRUE significherà blu, mentre FALSE rosso, e infine NIL se il campo non è mai stato inizializzato. Pertanto il problema della 2-colorazione può essere definito formalmente come: $\forall v \in V, \forall u \in G.Adj[v], v.2\text{COL} \neq u.2\text{COL}$.

   Suggerimento: Dato che BFS ci fornisce un ordine nella visita di un generico grafo, possiamo sfruttarla come base per verificare se una 2-colorazione è ammissibile.

```
1  is_2-colorabile( G, s ): G
2  for ∀v ∈ G.V − {s} do
3      d[v] = ∞
4  d[s] = 0;
5  s.2COL = TRUE;
6  Q.ENQUEUE(s);
7  while Q ≠ ∅ do
8      v = Q.DEQUEUE();
9      for ∀u ∈ G.Adj[v] do
10         if d[u] = ∞ then
11             d[u] = d[v] + 1;
12             u.2COL = NOT(v.2COL);
13             Q.ENQUEUE(u);
14         else
15             if v.2COL == u.2COL then
                   return: "non-colorabile"
   return: "colorabile"
```

L'algoritmo proposto non fa altro che sfruttare la visita in ampiezza per colorare il grafo controllando se esiste un conflitto. In particolare, se il nodo viene scoperto per la prima volta gli viene assegnato il colore opposto del padre tramite il comando $u.2\text{COL} = \text{NOT}(v.2\text{COL})$, viceversa, se il nodo era già stato scoperto dalla visita in precedenza si procede a controllare se esiste un conflitto verificando $v.2\text{COL} == u.2\text{COL}$. Se il controllo da esito positivo è possibile concludere che il grafo G non ammette 2-colorazione, cosi come mostrato nell'esempio b). Se non si incontra alcun conflitto si conclude con "colorabile".

(b) Analogamente agli esercizi presentati precedentemente la complessità di questa funzione può essere ricondotta a quella della BFS. Pertanto assumendo di usare liste di adiacenza avremo un costo in tempo pari a $\mathcal{O}(V + E)$.