Esercitazione 1: Depth-First Search

Giacomo Paesani

March 10, 2024

Esercizio 1. Un grafo diretto $G = (V, E)$ si dice diretto aciclico (DAG) se G non contiene alcun ciclo diretto. Modificare l'algoritmo della ricerca in profondità in maniera da poter controllare se un grafo diretto è aciclico o no; è possibile fare questa modifica in modo che il controllo avvenga in $\Theta(|V| + |E|)$?

Soluzione 1. Per risolvere questo esercizio è necessaria la seguente affermazione: un grafo diretto è aciclico se e solo se in un qualsiasi albero di ricerca in profondità non c'è un arco all'indietro.

L'Algoritmo 1 è una delle possibile soluzioni: il codice è inspirato all'algoritmo ricorsivo di ricerca in profondità proposto in [1]. Notiamo come tale algoritmo è stato modificato per controllare se il grafo diretto in esame è aciclico o no. Supponiamo che durante l'algoritmo viene trovato un arco (u, v) all'indietro (linea 28) allora la variabile *test* viene posta `FALSE` e ritornata subito. Questo vuol dire che la chiamata alla funzione **DFS-Visit** con input (G, z), dove z è l'antenato dell'albero che è anche radice di u, ritorna `FALSE` in linea 16 e quindi la funzione **DFS** con input G ritorna subito `FALSE`.

Supponiamo ora che la condizione in linea 28 non viene mai soddisfatta e quindi che non vi sono archi all'indietro. Allora la variabile *test* assume sempre il valore`TRUE` e quindi anche l'output di **DFS**(G) è `TRUE`. □

Algorithm 1 DFS modificata per controllare se un grafo diretto è aciclico o no.

```
1: global variables
2:     Color ↤ array di |V| elementi
3:     Parent ↤ array dei padri
4:     t ↤ array dei tempi di inizio visita
5:     T ↤ array dei tempi di fine visita
6:     test ↤ variabile booleana, inizializzata come TRUE
7:     time intero che simulta il tempo
8: end global variables
Input: grafo diretto G = (V, E).
Output: TRUE se G è aciclico e FALSE altrimenti.
9: function DFS(G)
10:    for u ∈ V do
11:        Color[u]=BIANCO
12:    time = 0
13:    for u ∈ V do
14:        if Color[u]==BIANCO then
15:            Parent[u] = u
16:            test=DFS-Visit(G,u)
17:            if test=FALSE then
18:                return FALSE
19:    return TRUE
20: function DFS-Visit(G,u)
21:    time = time + 1
22:    t[u] = time
23:    Color[u]=GRAY
24:    for v ∈ Adj[u] do
25:        if Color[u]==BIANCO then
26:            Parent[v] = u
27:            test=DFS-Visit(G,v)
28:        if Color[v]==GRIGIO then
29:            return FALSE
30:    Color[u]=NERO
31:    time = time + 1
32:    T[u] = time
33:    return test
```

Esercizio 2. Un grafo $G = (G, E)$ non diretto dice bipartito se l'insieme dei

vertici V può essere partizionato in due insiemi disgiunti U e W tali che: (1) $U \cap W = \emptyset$, (2) $U \cup W = V$ e (3) ogni arco di G è incidente ad un vertice di U e ad vertice di W. E' noto che un grafo G è bipartito se e solo se G non ha cicli di lunghezza dispari. Modificare l'algoritmo della ricerca in profondità in maniera da poter controllare se un grafo non diretto è bipartito o no, e in caso fornire una bipartizione; è possibile fare questa modifica in modo che il controllo avvenga in $\Theta(|V| + |E|)$? Domanda bonus: nel caso in cui G non è bipartito, come deve essere ulteriormente modificato l'algoritmo per ritornare un ciclo dispari di G?

Soluzione 2. L'idea principale è di assegnare un valore, 0 o 1, ad ogni vertice nel momento che viene visitato per la prima volta in maniera che tale valore sia diverso dal valore del padre nella ricerca. Se durante lo svolgimento dell'algoritmo si crea un arco che ha i due estremi con lo stesso valore, allora questo fatto certifica l'esistenza di un ciclo di lunghezza dispari nel grafo.

L'Algoritmo 2 è una delle possibile soluzioni: il codice è inspirato all'algoritmo ricorsivo di ricerca in profondità proposto in [1]. Notiamo come tale algoritmo è stato modificato non solo per controllare se il grafo non diretto in esame è bipartito o no, ma anche per fornire una prova di questo fatto. L'array $Parity$ ha l'obbiettivo di salvare la parità o disparità della lunghezza del cammino dal vertice di partenza della ricerca ad ogni vertice nel albero di ricerca. Per il vertice di partenza della ricerca z, $Parity[z]$ è inizializzato come 0 in maniera arbitraria (linea 18). Sia ora v un vertice diverso da quello di partenza. Se v viene visitato per la prima volta a partire da un nodo u allora si pone $Parity[v] = 1 - Parity[u]$ in maniera che questi valori siano distinti (linea 30). Supponiamo in fine che v è stato già visitato in passato (quindi $Color[v]$ è GRIGIO o NERO e $Parity[v]$ è già stato determinato) e c'è un arco dal corrente vertice u a v allora l'arco (u, v) forma, insieme al cammino da u a v nell'albero di ricerca, un ciclo C. La parità di C può essere facilmente determinata paragonando $Parity[u]$ e $Parity[v]$ (linea 35): se questi due valori sono uguali allora C ha lunghezza dispari e se sono diversi C ha lunghezza pari.

Supponiamo che la condizione in linea 35 viene soddisfatta da un arco (u, v) e quindi uno di questi vertici, diciamo u viene aggiunto alla coda C. A questi punto la condizione in linea 20 viene soddisfatta è l'algoritmo correttamente ritornerà che il grafo G non è bipartito.

Algorithm 2 DFS modificata per controllare se un grafo non diretto è bipartito o no.

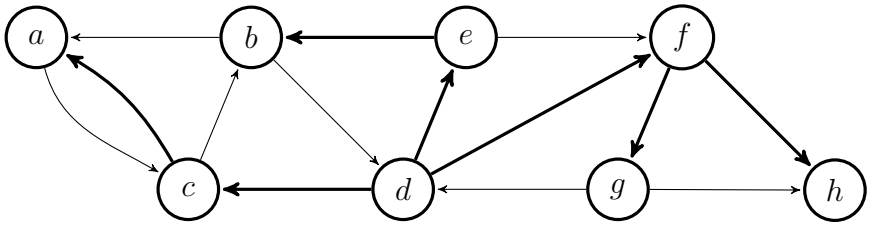
Input: grafo non diretto $G = (V, E)$.
Output: una bipartizione di G se G è bipartito e `FALSE`.

```
1: global variables
2:     Color ↤ array di |V| elementi
3:     Parent ↤ array dei padri
4:     t ↤ array dei tempi di inizio visita
5:     T ↤ array dei tempi di fine visita
6:     U ↤ queue, inizialmente vuota
7:     W ↤ queue, inizialmente vuota
8:     test ↤ variabile booleana, inizializzata come TRUE
9:     time intero che simulta il tempo
10: end global variables
11: function DFS(G)
12:     for u ∈ V do
13:         Color[u]=BIANCO
14:     time = 0
15:     for u ∈ V do
16:         if Color[u]==BIANCO then
17:             Parent[u] = u
18:             Parity[u] = 0
19:             test=DFS-Visit(G,u)
20:             if test==FALSE then
21:                 return FALSE
22:     return U and W
23: function DFS-Visit(G,u)
24:     time = time + 1
25:     t[u] = time
26:     Color[u]=GRAY
27:     for v ∈ Adj[u] do
28:         if Color[v]==BIANCO then
29:             Parent[v] = u
30:             Parity[v] = 1 − Parity[u]
31:             test=DFS-Visit(G,u)
32:             if test=FALSE then
33:                 return FALSE
34:         else
35:             if Parity[u] == Parity[v] then
36:                 return FALSE
37:     Color[u]=NERO
38:     time = time + 1
39:     T[u] = time
40:     if Parity[u] = 0 then
41:         U.enqueue(u)
42:     else
43:         W.enqueue(v)
44:     return TRUE
```

Finalmente consideriamo il caso in cui la condizione presente nella linea 35 non si verifica mai. Allora ogni vertice u sarà aggiunto ad una sola tra due liste: se $Parity[u] = 0$ allora u viene aggiunto alla lista U (linea 41) e viene aggiunto alla lista W altrimenti (linea 43). Questa bipartizione viene riportata dall'algoritmo principale (linea 22). □

Esercizio 3 (I. Salvo)**.** Si consideri il grafo diretto G illustrato nella figura qui sotto e l'albero T formato formato dagli archi evidenziati. L'albero T può essere prodotto da una ricerca in profondità?

- In caso positivo, esibire una rappresentazione di G tramite liste di adiacenza in grado di produrre T e specificare il nodo da cui parte la ricerca e il tipo degli archi ottenuto a seguito della visita.
- In caso negativo, rimpiazzare un arco di T con un altro arco in maniera da ottenere un albero T' con la proprietà che T' possa essere un albero di ricerca per una ricerca in profondità. In tal caso, esibire una rappresentazione di G tramite liste di adiacenza in grado di produrre T' e specificare il nodo da cui parte la visita e il tipo degli archi ottenuto a seguito della visita.



In fine, che succede se si considera lo stesso grafo G dove però gli archi non sono diretti?

Soluzione 3. No, T non può essere ottenuto da una ricerca in profondità, per spiegare il perché è necessario analizzare ogni singolo vertice di G come vertice di partenza.

- a non può essere il vertice di partenza perché altrimenti l'algoritmo dovrebbe visitare c ma (a, c) non fa parte di T;
- b non può essere il vertice di partenza perché altrimenti l'algoritmo dovrebbe visitare uno tra a e d ma sia (b, a) che (b, d) non fa parte di T;

- c non può essere il vertice di partenza perché dopo aver visitato a, l'algoritmo dovrebbe visitare b ma l'arco (c, b) non fa parte di T.

- e non può essere il vertice di partenza perché dopo aver visitato b, l'algoritmo dovrebbe visitare uno tra a e d ma sia (b, a) che (b, d) non fa parte di T;

- g non può essere il vertice di partenza perché altrimenti l'algoritmo dovrebbe visitare uno tra d e h ma sia (g, d) che (g, h) non fanno parte di T;

- il vertice h non avendo archi uscenti è ininfluente per la risposta all'esercizio: se h viene selezionato come vertice di partenza allora esso farà parte di un albero di cui h è l'unico vertice. Se altrimenti h non è il vertice di partenza, h è una foglia del albero di ricerca con g o f come padre. In sintesi, T può essere prodotto da una ricerca in profondità se e solo se T_h può essere prodotto da una ricerca in profondità, dove T_h è il sottografo di G_h ottenuti da T e G rimuovendo, rispettivamente, il vertice h e tutti gli archi ad esso incidenti.

- l'ultimo vertice da analizzare è d. La sequenza $d \to f \to h$ seguita da $f \to g$ è fino a questo punto una legittima sequenza di ricerca in profondità. Adesso l'algoritmo deve visita uno dei vertici adiacenti a d che non sia stato già visitato, cioè o c o e. Se l'algoritmo visita è per primo c, allora il vertice b dovrebbe essere visitato per la prima volta da c usando l'arco (c, b) che però non fa parte di T. Infine se l'algoritmo visita e per primo, allora il vertice a dovrebbe essere visitato per la prima volta da b usando l'arco (b, a) che però non fa parte di T. E' abbastanza immediato anche che scegliere sequenze iniziali diverse da $d \to f \to h$ $f \to g$ producono situazioni simili a quelle dei primi cinque vertici considerati.

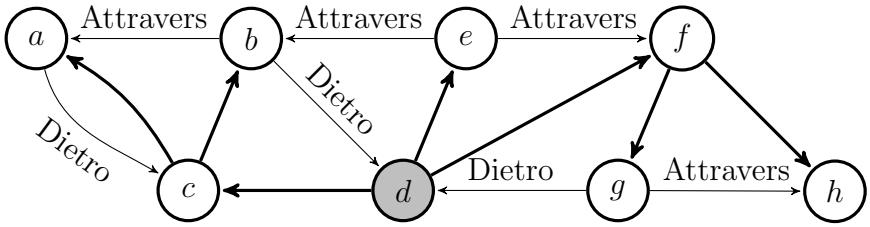
In sintesi, abbiamo mostrato come nessun vertice può essere scelto come vertice iniziale per una ricerca in profondità di G.

Consideriamo ora T', l'albero ottenuto da T rimuovendo l'arco (e, b) e includendo l'arco (c, b). Mostriamo ora che T' può essere ottenuto da una ricerca in profondità. Prima specifichiamo le liste di adiacenza:

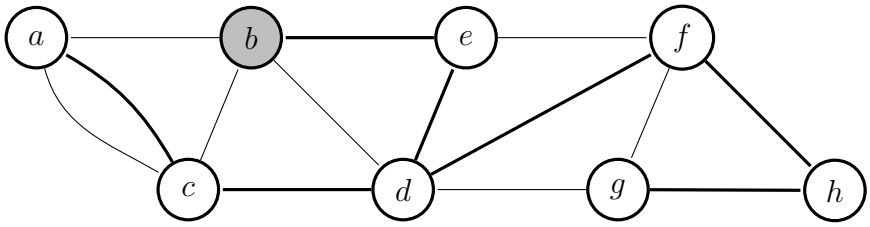
- $a : c$

- $b : a \mapsto d$
- $c : a \mapsto b$
- $d : f \mapsto c \mapsto e$
- $e : b \mapsto f$
- $f : h \mapsto g$
- $g : d \mapsto h$
- $h :$

La figura sotto rappresenta G con l'albero T' evidenziato e gli archi che non sono etichettati con il loro tipo. Ora è evidente che avendo come vertice di partenza d, l'albero T' è il prodotto della ricerca in profondità su G.



Nel caso G senza orientazione degli archi, la risposta sarebbe comunque negativa ma per motivi differenti. Ci limitiamo ad esibire un albero T' che può essere ottenuto da una ricerca in profondità dove il vertice colorato è quello da dove parte la ricerca. □



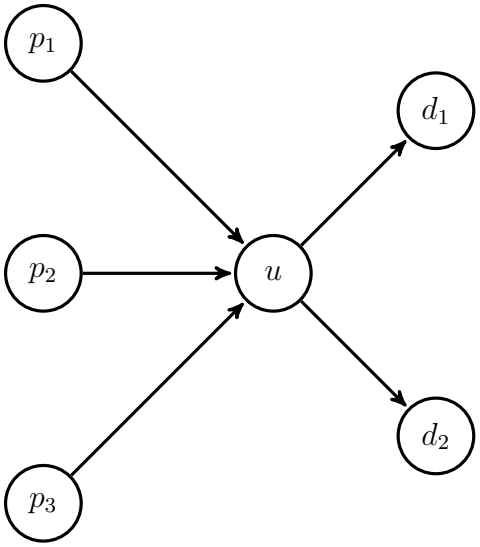
Esercizio 4 (22.3-11,[1])**.** E' possibile avere un vertice u di un grafo diretto G che finisce in un albero di ricerca in profondità che contiene solo u, anche se u ha grado entrante e grado uscente almeno uno?

Soluzione 4. Dato un grafo G, la foresta ottenuta da una visita in profondità, e quindi anche la classificazione degli archi in base alla visita, dipende fortemente da come vengono selezionati i vertici del grafo nei vari passaggi dell'algoritmo e da come sono ordinate le liste di adiacenza.

La risposta al esercizio è positiva se si ammette la presenza di cappi. Infatti ogni grafo che ammette una componente connessa contenente il solo vertice u e l'arco (u, u) verifica banalmente le richieste. Dimostriamo la seguente affermazione: sia $G = (V, E)$ un grafo diretto e $u \in V$, allora u è l'unico vertice dell'albero di visita che lo contiene se e solo se u non appartiene ad alcun ciclo di lunghezza almeno due di G.

$\Leftarrow$ Sia $u \in V$ tale che u non appartiene ad alcun ciclo di G di lunghezza almeno due. Siano $p_1, \ldots, p_k$ i vertici di G tali che (p_i, u) è un arco di G, per ogni $i = 1, \ldots, k$, e $d_1, \ldots, d_h$ i vertici di G tali che (u, d_j) è un arco di G, per ogni $j = 1, \ldots, h$. Se per qualche $i \in \{1, \ldots, k\}$, abbiamo che $t[p_i] < t[u]$ allora p_i e u appartengono allo stesso albero della visita. Se per qualche $j \in \{1, \ldots, h\}$, abbiamo che $t[u] < t[d_j]$ allora u e d_j appartengono allo stesso albero della visita.

Allora per ogni $i = 1, \ldots, k$ e per ogni $j = 1, \ldots, h$ dobbiamo avere che $t[d_j] < t[u] < t[p_i]$: questo è possibile dato che u non appartiene ad alcun ciclo di lunghezza almeno due. Infatti una volta terminate le visite a partire dai vertici di $\{d_1, \ldots, d_h\}$ i vertici $p_1, \ldots, p_k$ e u non sono ancora stati visitati. In questo caso u è l'unico vertice dell'albero di visita che lo contiene, infatti tutti gli archi della forma (p_i, u) e (u, d_j) sono di attraversamento.



$\Rightarrow$ Supponiamo, per assurdo, che u appartiene ad un ciclo $C = \{v_1, \ldots, v_k\}$ con archi (v_i, v_{i+1}) e (v_k, v_1), per $k \geq 2$, con $v_1 = u$. Sia T l'albero di visita di G che comprende un vertice v di C, allora è facile osservare che T visita tutti i vertici della componente fortemente connessa di G che contiene v, inclusi tutti i vertici di C e in particolare u. Dato che $|C| \geq 2$, abbiamo che u non è l'unico vertice visitato in T e quindi una contraddizione. $\square$

References

[1] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. Introduction to algorithms. 2022.