Esercitazione 4: Breath First Search

Giacomo Paesani

April 2, 2025

Esercizio 1 (I. Salvo)**.** Sia $G = (V, E)$ un grafo diretto, un vertice $u \in V$ è detto *principale* se per ogni vertice $v \in V$ esiste un cammino diretto da u a v. Fornire un algoritmo in pseudo-codice che, dato un grafo diretto G, determina tutti i vertici principali di G. E' possibile che tale algoritmo abbia complessità $\mathcal{O}(|V| + |E|)$?

Soluzione 1. Per risolvere questo algoritmo, dobbiamo prima ricordare cosa è una componente fortemente connessa: sia $G = (V, E)$ un grafo diretto, allora una componente fortemente connessa di G è un sottoinsieme di vertici $U \subseteq V$ massimale tale che per ogni due vertici $u, v \in U$ allora esiste sempre un cammino da u a v e un cammino da v ad u.

Supponiamo inizialmente che G sia aciclico e che quindi ammette ordinamenti topologici per i suoi vertici. Sia $W \subseteq V$ l'insieme di tutti i vertici di G con grado entrante pari a 0: questi sono tutti i possibili candidati ad essere vertici principali. Ricordiamo che se G è aciclico, allora $|W| \geq 1$. Se l'insieme W contiene almeno due vertici w_1 e w_2 allora G non ammette vertici principali: infatti non esiste alcun cammino da w_1 a w_2 perché w_2 non ha alcun arco entrante e non esiste alcun cammino da w_2 a w_1 perché w_1 non ha alcun arco entrante. Finalmente consideriamo il caso $W = \{w\}$ e allora possiamo concludere che w è l'unico vertice principale di G.

Consideriamo ora il caso più generale, cioè quello in cui G contiene cicli. Facciamo ora la seguente affermazione: siano v_1 e v_2 due vertici appartenenti alla stessa componente fortemente connessa di G, allora v_1 è un vertice principale di G se e solo se v_2 lo è. Infatti se da v_1 esiste cammino diretto per ogni altro vertice di G, allora tale cammino esiste anche da v_2 (basta usare il cammino da v_2 a v_1 come prefisso) e viceversa.

Grazie alla precedente affermazione possiamo adottare l'idea implementata nel Algoritmo 1: prima calcoliamo il grafo G^{SCC} delle componenti forte-

Algorithm 1 Algoritmo per calcolare tutti i vertici principali di un grafo diretto.

Input: grafo diretto $G = (V, E)$.
Output: insieme che contiene tutti i vertici principali di G.

```
1: function PRINCIPAL(G)
2:     G^SCC =SCC(G)
3:     InDeg ↤ vettore dei gradi entranti di G^SCC, inizializzato a 0
4:     A ↤ insieme, inizialmente vuoto
5:     a = 0
6:     for u ∈ V(G^SCC) do
7:         for (v, u) ∈ E(G^SCC) do
8:             InDeg[v] = InDeg[v] + 1
9:     for u ∈ V(G^SCC) do
10:        if InDeg[u] == 0 and a! = 0 then
11:            return ∅
12:        if InDeg[u] == 0 and a == 0 then
13:            a = 1
14:            p = u
15:    for w ∈ p do
16:        A.add(w)
17:    return A
```

mente connesse di G con la chiamata alla funzione SSC(G) (Linea 2), esso è per sua natura aciclico, e su G^{SCC} possiamo adottare la strategia descritta per i grafi aciclici. Il ciclo **for** in Linea 6 serve a calcolare il grado entrante di ogni vertice di G^{SCC} e salvarlo nel vettore $InDeg$. La variabile a controlla il numero di vertici di G^{SCC} con grado entrante nullo, se ce n'è più di uno e quindi a viene aggiornata più di una volta nel corso dell'algoritmo allora non vi sono vertici principali in G^{SCC} e quindi neanche in G (Linea 11). Supponiamo invece che a viene aggiornata una volta sola e quindi esiste un solo vertice u di G^{SCC} con grado entrante nullo. Allora u viene salvato nella variabile p (Linea 14) che rappresenta la componente fortemente connessa principale di G e quindi in A vengono aggiunti tutti i vertici di G in p (Linea 16), che quindi sono tutti e soli i vertici principali di G. □

Esercizio 2 (22.2-8, [1])**.** Sia $G = (V, E)$ un grafo non diretto, allora si definisce il *diametro* di G, $diam(G) = \max_{u,v \in V} dist(u, v)$, il massimo della distanza tra due qualsiasi vertici di G. Fornire un algoritmo in pseudo-codice che restituisca il diametro di un grafo G, nel caso in cui G sia un albero. E' possibile ottenere una soluzione con tempo di esecuzione $\mathcal{O}(|V|)$?

Soluzione 2. Per dare una spiegazione convincente della soluzione proposta abbiamo bisogno della seguente affermazione: sia $G = (V, E)$ un albero e u un vertice di G. Se u è un vertice a distanza massima in una visita in profondità radicata in un nodo arbitrario r, allora esiste un altro vertice $v \in V$ tale che $dist(u, v) = diam(G)$.

L'Algoritmo 2 è una delle possibile soluzioni: il codice è inspirato all'algoritmo ricorsivo di ricerca in profondità proposto in [1] e l'idea è la seguente. Prima svolgiamo una visita in profondità dell'albero scegliendo un qualsiasi vertice r come radice: otteniamo il vettore $Dist$ che registrata la distanza di ogni vertice di G da r. In seguito facciamo una seconda visita dell'albero radicata in uno dei vertici a distanza massima da r. Dopo questa ulteriore visita otteniamo il valore corretto del diametro dell'albero in input.

Durante la visita in profondità dell'albero G, nel vettore $Dist$, inizializzato a 0 (Linea 3), viene registrata la distanza dalla radice al vertice in analisi. Il contatore c indica il numero di componente connesse di G: se ce n'è più di una allora il diametro di G è $+\infty$ e viene correttamente riportato in Linea 12.

Supponiamo ora che G ha esattamente una componente connessa, la chiamata DFS-Dist(G,r) (Linea 13) serve a trovare un vertice u (che in questo caso è una foglia) di G a distanza massima m dalla radice r. Allora es-

Algorithm 2 Algoritmo per calcolare il diametro di un albero non diretto.

Input: grafo non diretto $G = (V, E)$.
Output: diametro di G.

```
1: global variables
2:     Visited ↤ vettore di |V| elementi, inizializzato a 0
3:     Dist ↤ vettore di |V| elementi, inizializzato a 0
4:     Q ↤ coda vuota
5: end global variables
6: function DIAMETER(G)
7:     c = 0
8:     for z ∈ V do
9:         if Visited[z]==0 then
10:            c = c + 1
11:            if c > 1 then
12:                return +∞
13:            (u, d)=DFS-DIST(G,z)
14:    for z ∈ V do
15:        Visited[z] = 0
16:        Dist[z] = 0
17:    (u', d')=DFS-DIST(G,u)
18:    return d'
19: function DFS-DIST(G,u)
20:    z = u
21:    max = 0
22:    Q.enqueue(u)
23:    while Q ≠ ∅ do
24:        v ↤ Q.dequeue()
25:        for w ∈ N(v) do
26:            if Visited[w] == 0 then
27:                Visited[w] = 1
28:                Dist[w] = Dist[v] + 1
29:                if Dist[w] > max then
30:                    max = Dist[w]
31:                    z = w
32:                Q.enqueue(w)
33:    return (z, max)
```

eguiamo una nuova ricerca in profondità di G con radice u (Linea 17). Al termine di della chiamata DFS-DIST(G,u), nella variabile m' viene registrato il diametro di G e riportato in Linea 18.

Concludiamo la soluzione a questo esercizio analizzando la complessità della soluzione proposta nel Algoritmo 2. La complessità è dominata dalle due ricerche in profondità consecutive e quindi $\mathcal{O}(|V| + |E|)$. Ricordando che se l'input è un albero, si ha che $|E| = |V| - 1$ e quindi la complessità de Algoritmo 2 è $\mathcal{O}(|V|)$. □

Esercizio 3 (I. Salvo)**.** Rimuovere un arco da un grafo (diretto o non diretto) modifica le distanze tra alcune coppie di vertici e ne lascia invariate alcune. Fornire un algortimo in pseudo-codice che dato un grafo $G = (V, E)$ diretto, un vertice $s \in V$, un arco $(u, v) \in E$ e un vettore dei padri $Parent$, relativo ad una BFS effettuata su G con radice s determina se la rimozione di (u, v) da G modifica le distanze da s. E' possibile ottenere una soluzione con tempo di esecuzione $\mathcal{O}(|V|)$?

Soluzione 3. Facciamo prima delle considerazioni generali. Se l'arco (u, v) in input non è un arco dell'albero di visita della BFS con radice s allora la rimozione di tale arco non modifica le distanze. Consideriamo ora il caso in cui (u, v) è un arco dell'albero. Allora per risolvere l'esercizio è necessario capire se esiste un cammino in $G-(u, v)$ da s ad v della medesima lunghezza di quello da s a v in G passando per u.

In particolare, si controlla se esiste un vertice z diverso da u tale che $Dist(s, u) = Dist(s, z)$ e $(z, v) \in E$: se esiste tale vertice allora esiste un cammino ulteriore rispetto a quello nell'albero di visita che preserva le distanze, altrimenti le distanze vengono alterate. Una delle possibili soluzioni è presentata nel Algoritmo 3. La funzione CALCOLADISTANZA (Linee 13-19) sfrutta l'idea della programmazione dinamica e il vettore dei padri per calcolare le distanze di s dai vertici del grafo.

Se (u, v) non è un arco dell'albero di visita, allora in Linea 6 faccio ritornare correttamente `FALSE`. Supponiamo che (u, v) è un arco dell'albero di visita; allora con la funzione CALCOLADISTANZA di Linea 8 permette il calcolo delle distanze da s. Si scorre la lista dei vertici di G per controllare se esiste un vertice z adiacente a v diverso da u tale che $Dist(s, u) = Dist(s, z)$. Se tale vertice esiste allora viene riportato correttamente il valore `FALSE` in Linea 11. Altrimenti si deduce che ogni cammino di lunghezza minima in G da s a v usa l'arco (u, v) e allora l'algoritmo riporta `TRUE` in Linea 12.

Algorithm 3 Algoritmo che valuta se cambiano le distanze tra vertici a partire da un vertice rimuovendo un arco.

Input: grafo $G = (V, E)$, tre vertici s, u e v e un vettore $Parent$.
Output: `TRUE` se almeno una distanza da s a un vertice cambia e `FALSE` altrimenti.

```
1: global variables
2:     Dist ↤ vettore di |V| elementi , inizializzato a −1
3: end global variables
4: function DIST-MODIF(G, s, u, v, Parent)
5:     if Parent[v] ≠ u then
6:         return FALSE
7:     for z ∈ V do
8:         CALCOLADISTANZA(G, z)
9:     for z ∈ V do
10:        if z ≠ u and Dist[z] == Dist[u] and (z, v) ∈ E then
11:            return FALSE
12:    return TRUE
13: function CALCOLADISTANZA(G,z)
14:    if Dist[z] == −1 then
15:        if Parent[z] == z then
16:            Dist[z] = 0
17:        else
18:            Dist[z] = 1+CALCOLADISTANZA(G, Parent[z])
19:    return
```

Soffermiamoci, in fine, sul tempo di esecuzione dell'Algoritmo 3. Partendo da s e procedendo induttivamente sui suoi vertici a se adiacenti, le chiamate a CalcolaDistanza si risolvono in tempo $\mathcal{O}(|V|)$. Inoltre, per ogni vertice $z \in V$, verificare che z soddisfa le condizioni di Linea 10 richiede tempo costante $\mathcal{O}(1)$. Quindi la complessità della soluzione è $\mathcal{O}(|V|)$. □

Esercizio 4. Nel gioco degli scacchi il *cavallo* si muove nella seguente maniera: due caselle in orizzontale e una in verticale, o viceversa. Fornire un algoritmo in pseudo-codice che, data una scacchiera con N righe e M colonne, la posizione di partenza e quella di arrivo, calcola il numero minimo di turni necessari ad un cavallo di passare da una posizione all'altra in maniera che il tempo di esecuzione sia pari a $\mathcal{O}(M \cdot N)$.

Che modifiche sono necessarie se invece del cavallo si usa uno degli altri pezzi? Oppure se sono presenti degli altri pezzi statici ma non mangiabili?

Soluzione 4. Prima di risolvere questo esercizio, è necessario trasformare la richiesta in un problema informatico e algoritmico. La scacchiera diventa grafo non diretto con $M \cdot N$ vertici, uno per ogni casella; in oltre, nel caso che il pezzo in considerazione è un cavallo, ogni vertice del grafo ha al più otto vertici adiacenti possibili perché vengono escluse tutte quelle posizioni che risultano *fuori dalla scacchiera*. Allora in seguito alla creazione di questo grafo, la richiesta dell'esercizio diventa quella di trovare il cammino minimo tra due vertici nel grafo.

La soluzione proposta è data dal Algoritmo 4. I vettori R e C sono dati in maniera che, per ogni valore $k = 1, \ldots, 8$, la coppia $(R[k], C[k])$ rappresenta una diversa possibile mossa del cavallo. La matrice $Dist$, inizializzata a 0, rappresenta alla fine dell'esecuzione dell'algoritmo la distanza di ogni casella da quella di partenza . La matrice $Visited$ rappresenta, nel corso dell'esecuzione dell'algoritmo, se una casella è stata già visitata. La funzione isValid, data in Linea 27 con input (M, N, x, y) ha il semplice compito di controllare se il vertice (x, y) è all'interno della scacchiera $M \times N$.

Ora analizziamo la funzione principale ShortestPath-Knight. Questa è una modifica del classico algoritmo di BFS. L'idea è di riconoscere quali sono gli archi di questo grafo che modella la scacchiera mentre si esegue la visita in profondità. Infatti, dato un vertice (x, y) della scacchiera, i suoi vicini sono tutti i vertici della scacchiera della forma $(x + R[k], y + C[k])$, per qualche $k \in \{1, \ldots, 8\}$, cioè quelli raggiungibili da (x, y) con una sola mossa.

Che succede se non esiste nessuna sequenza di mosse per muovere il cavallo dalla casella iniziale (a, b) ad un altra casella (x, y)? Cioè come valutiamo

Algorithm 4

Input: quattro numeri interi M, N, a e b.
Output: vettore $Dist$ delle distanze dalla casella di cordinate (a, b).

```
1: global variables
2:     R ↤ vettore di 8 elementi, inizializzato come {2, 2, −2, −2, 1, 1, −1, −1}
3:     C ↤ vettore di 8 elementi, inizializzato come {−1, 1, 1, −1, 2, −2, 2, −2}
4:     Dist ↤ matrice M × N, inizializzata a 0
5:     Visited ↤ matrice M × N, inizializzata a 0
6: end global variables
7: function ShortestPath-Knight(M, N, a, b)
8:     if ¬(1 ≤ a ≤ M and 1 ≤ b ≤ N) then
9:         for x ∈ {1, ..., M} and y ∈ {1, ..., N} do
10:            Dist[(x, y)] = +∞
11:        return Dist
12:    Visited[(a, b)] = 1
13:    Q.enqueue(a, b)
14:    while Q ≠ ∅ do
15:        (x, y) ↤ Q.dequeue()
16:        for k ∈ {1, ..., 8} do
17:            x' = x + R[k]
18:            y' = y + C[k]
19:            if isValid(M,N,x',y')==TRUE and Visited[(x', y')] == 0 then
20:                Visited[(x', y')] = 1
21:                Dist[(x', y')] = Dist[(x, y)] + 1
22:                Q.enqueue(w)
23:    for x ∈ {1, ..., M} and y ∈ {1, ..., N} do
24:        if Visited[(x, y)] == 0 then
25:            Dist[(x, y)] = +∞
26:    return Dist
27: function isValid(M,N,x,y)
28:    if 1 ≤ x ≤ M and 1 ≤ y ≤ N then
29:        return TRUE
30:    return FALSE
```

rapidamente se (a, b) e (x, y) sono in diverse componenti connesse del grafo della scacchiera? Se tale condizione è vera, allora a $Dist[x, y]$ viene assegnato il valore $+\infty$.

Notiamo che il grafo creato ha $M \cdot N$ vertici. Dato che ogni vertice ha grado massimo 8, per il lemma delle strette di mano (Handshaking's Lemma) si ha che ci sono al più $16M \cdot N$ archi. Quindi la complessità dell'algoritmo proposto è $\mathcal{O}(M \cdot N)$. □

References

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