Esercitazione 4: Breath First Search

Giacomo Paesani

April 10, 2024

Esercizio 1 (22.2-8, [1])**.** Sia $G = (V, E)$ un grafo non diretto, allora si definisce il *diametro* di G, $diam(G) = \max_{u,v \in V} d(u, v)$, 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 1. 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 $d(u, v) = diam(G)$.

L'Algoritmo 1 è una delle possibili soluzioni: il codice è inspirato all'algoritmo ricorsivo di ricerca in profondità proposto in [1] e l'idea è la seguente. Prima svolgiamo una qualsiasi visita dell'albero (in questo caso usiamo una visita in profondità) scegliendo un qualsiasi vertice r come radice e per ogni vertice viene registrata la distanza 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 (Linee 13 e 14), viene registrata la distanza dalla radice al vertice in analisi (Linea 30): la misurare la distanza in questo caso è corretto perché G è un albero. 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 18.

Supponiamo ora che G ha esattamente una componente connessa, la prima chiamata alla funzione **DFS-Dist**(G,u) (Linea 16) serve a trovare un vertice (che in questo caso è una foglia) di G a distanza massima dalla radice u.

Algorithm 1 Algoritmo per calcolare il diametro di un albero non diretto.

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

```
global variables
    Color ↤ array di |V| elementi
    dist ↤ array di |V| elementi
    max ↤ intero che traccia la distanza massima dalla radice
    w ↤ variabile vertice di appoggio
end global variables
function DIAMETRO(G)
    for u ∈ V do
        Color[u]=BIANCO
        dist[u] = 0
    c = 0
    for u ∈ V do
        if Color[u]==BIANCO then
            w = u
            max = 0
            c = c + 1
            if c > 1 then
                return +∞
            DFS-Dist(G,u)
    for u ∈ V do
        dist[u] = 0
    max = 0
    DFS-Dist(G,w)
    return max
function DFS-Dist(G,u)
    Color[u]=GRIGIO
    for v ∈ Adj[u] do
        if Color[v]==BIANCO then
            Color[v]=GRIGIO
            dist[v] = dist[u] + 1
            if dist[v] > max then
                dist[v] = max
                w = v
            DFS-Dist(G,v)
    return
```

Allora eseguiamo una nuova ricerca in profondità di G con vertice iniziale w (Linea 21). Al termine di della chiamata **DFS-Dist**(G,w), nella variabile max viene registrato il diametro di G e riportato in Linea 24.

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

Esercizio 2 (I. Salvo)**.** Fornire un algoritmo in pseudo-codice che, dato un grafo non diretto $G = (V, E)$ e due nodi u e v, restituisce tutti i nodi che hanno la stessa distanza da u e v in tempo $\mathcal{O}(|V| + |E|)$.

Soluzione 2. L'idea della soluzione è di implementare due BFS, una con radice u e l'altra con radice v, e per ogni vertice $w \in V$ confrontare la distanza da u a w con la distanza da v a w.

L'Algoritmo 2 è una delle possibili soluzioni: il codice è inspirato all'algoritmo proposto in [1]. Prima consideriamo il caso più semplice, cioè quello in cui i due vertici in input u e v sono uguali. Allora, grazie al ciclo **for** in Linea 10, si deduce banalmente che tutti i vertici hanno la medesima distanza da u e da v. L'effetto della chiamata alla funzione BFS in Linea 16 è quello che nel vettore d_2 vengono messe le distanze dei vertici di G da u (Linea 18). In maniera simile, l'effetto della chiamata alla funzione BFS in Linea 21 è quello che nel vettore d_1 vengono messe le distanze dei vertici di G da v (Linea 32). Alla fine, è sufficiente confrontare le coordinate dei vettori d_1 e d_2 per determinare i vertici di G che sono equidistanti da u e da v. Questo completa la descrizione della funzione **BFS-equidistant**.

Algorithm 2 BFS modificata per tutti i vertici equidistanti da due vertici.

Input: grafo $G = (V, E)$ e due vertici u e v.
Output: .

1: **global variables**
2: $Parent \hookleftarrow$ array di $|V|$ elementi
3: $Color \hookleftarrow$ array di $|V|$ elementi
4: $d_1 \hookleftarrow$ array di $|V|$ elementi
5: $d_2 \hookleftarrow$ array di $|V|$ elementi
6: $S \hookleftarrow$ insieme, inizialmente vuoto
7: C coda, inizialmente vuota
8: **end global variables**
9: **function BFS-equidistant**(G, u, v)
10: **if** $u == v$ **then**
11: **return** V
12: **for** $w \in V$ **do**
13: $d_1[w] = 0$
14: $d_2[w] = 0$
15: $C.enqueue(u)$
16: BFS(G,u)
17: **for** $w \in V$ **do**
18: $d_2[w] = d_1[w]$
19: $d_1[u] = 0$
20: $C.enqueue(u)$
21: BFS(G,v)
22: **for** $w \in V$ **do**
23: **if** $d_1[w] == d_2[w]$ **then**
24: $S.add(w)$
25: **return** S
26: **function BFS**(G,z)
27: **while** $C \neq \emptyset$ **do**
28: $w = C.dequeue()$
29: **for** $t \in Adj[w]$ **do**
30: **if** $d_1[t] == 0$ and $t \neq z$ **then**
31: $Parent[t] = w$
32: $d_1[t] = d_1[w] + 1$
33: $C.enqueue(t)$

La soluzione si conclude specificando che la funzione BFS in Linea 26 è inspirata dal codice della ricerca in ampiezza. $\square$

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 a partire da 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 a partire da 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.

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.

```
global variables
    d ↤ array di |V| elementi
end global variables
function Dist-Modif(G, s, u, v, Parent)
    if Parent[v] ≠ u then
        return FALSE
    for z ∈ V do
        d[z] = −1
        CalcolaDistanza(G, z)
    for z ∈ V do
        if z ≠ u and d[z] == d[u] and (z, v) ∈ E then
            return FALSE
    return TRUE
function CalcolaDistanza(G,z)
    if d[z] == −1 then
        if Parent[z] == v then
            d[z] = 0
        else
            d[z] = 1+CalcolaDistanza(G, Parent[z])
    return
```

In particolare, andiamo a controllare 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 14-20) sfrutta l'idea della programmazione dinamica e il vettore dei padri per calcolare le distanza di s dai vertici del grafo.

Allora una volta calcolate le distanze nella chiamata alla funzione **CalcolaDistanza** in Linea 9, scorriamo la lista dei vertici di G per controllare se esiste un vertice z adiacente a v diverso da u tale che $dist(s, u) = dist(s, z)$ e $(z, v) \in E$. Se tale vertice esiste allora viene riportato correttamente il valore `FALSE` in Linea 6. 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 13.

Soffermiamoci, in fine, sul tempo di esecuzione dell'Algoritmo 3. Partendo da s e procedendo induttivamente sui suoi vertici a se adiacenti, la chiamata CALCOLADISTANZA si risolve in tempo costante $\mathcal{O}(1)$. Inoltre, facendo $|V|$ chiamate a tale funzione, si ottiene che 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 MxN, 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 $\mathcal{O}(M + N)$.

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.

Algorithm 4

Input:

Output:

```
global variables
    R ↤ array di 8 elementi, inizializzato come {2, 2, −2, −2, 1, 1, −1, −1}
    C ↤ array di 8 elementi, inizializzato come {−1, 1, 1, −1, 2, −2, 2, −2}
    dist ↤ matrice MxN
    visited ↤ matrice MxN
end global variables
function ShortestPath-Knight(M, N, x1, y1, x2, y2)
    if isValid(M,N,x1,y1)==FALSE or isValid(M,N,x2,y2)==FALSE then
        return +∞
    for i = 1, . . . , M do
        for j = i, . . . , N do
            visited[i, j] = 0
    dist[x2, y2] = 0
    visited[x1, y1] = 1
    ComputeDist(M, N, x1, y1, x2, y2)
    return dist[x1, y1]
function isValid(M,N,x,y)
    if 1 ≤ x ≤ M and 1 ≤ y ≤ N then
        return TRUE
    return FALSE
function ComputeDist(M, N, x1, y1, x2, y2)
    if x1 == x2 and y1 == y2 then
        return
    min = +∞
    T==TRUE
    for k = 1, . . . , 8 do
        z = x1 + R[k]
        w = y1 + C[k]
        T = T and (¬isValid(M,N,z,w) or visited[z, w] == 1)
    if T == TRUE then
        dist[x1, y1] = +∞
        return
    for k = 1, . . . , 8 do
        x′ = x1 + R[k]
        y′ = y1 + C[k]
        if isValid(M,N,x′,y′)==TRUE then
            if visited[x′, y′] = 0 then
                visited[x′, y′] = 1
                ComputeDist(M, N, x′, y′, x2, y2)
            if dist[x′, y′] < min then
                min = dist[x′, y′]
    dist[x1, y1] = min + 1
    return
```

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 solo per il valore di (x_1, y_1) (Linea 13), rappresenta per ogni entrata la distanza del vertice in oggetto dalla posizione di partenza (x_1, y_1). La matrice *visited* (inizializzata nelle Linee 10 e 14) rappresenta per ogni vertice se quel vertice è stato già visitata a partire dalla posizione di partenza (x_1, y_1). La funzione **isValid**, data in Linea 17 con input (M, N, x, y) ha il semplice compito di controllare se il vertice (x, y) è all'interno della scacchiera MxN: ogni chiamata a tale funzione ha come obbiettivo se la posizione proposta è un vertice del grafo.

Ora analizziamo la funzione principale **ShortestPath-Knight**. Come prima cosa, vengono controllate che sia la posizione iniziale che quella finale proposte sono valide, in caso contrario l'algoritmo ritornerà il valore $+\infty$ (Linea 9) esprimendo il fatto che non esiste cammino tra questi due vertici (visto che almeno uno dei due vertici non esiste). A questo punto viene chiamata la funzione **ComputeDist** che ha il compito di calcolare la distanza minima tra i due vertici in input, e infine riporta il valore desiderato (Linea 16).

La funzione **ComputeDist** è la funzione principale del complesso che forma la soluzione. L'idea è quella di usare un approccio di programmazione dinamica per calcolare la distanza dal corrente vertice (x_1, y_1) al vertice terminale (x_2, y_2). Il ciclo **for** in Linea 33 analizza tutti i vicini del vertice corrente (Linee 34 e 35), analizza le posizioni valide (Linea 39) e tra queste ne seleziona una che realizza la distanza minima. Le chiamate ricorsive della funzione **ComputeDist** si incapsulano finchè non si giunge ad una chiamata che è stata già risolta e a quel punto si procede a ritroso finchè possibile.

Che succede se non esiste nessuna sequenza di mosse per muovere il cavallo dalla posizione iniziale (x_1, y_1) a quella finale (x_2, y_2)? Cioè come valutiamo rapidamente se (x_1, y_1) e (x_2, y_2) sono in diverse componenti connesse del grafo della scacchiera? Se tale condizione è vera, allora ad un certo momento nell'esecuzione dell'algoritmo esiste un vertice tale che questo vertice e tutti i suoi vertici adiacenti sono stati già visitati e le corrispondenti chiamate alla funzione **ComputeDist** sono tutte contemporaneamente aperte. Questa condizione è rappresentata dalla variabile booleana T. Se T risulta vera, allora possiamo affermare che non ci sono cammini tra (x_1, y_1) e (x_2, y_2), assegnare $+\infty$ al valore di $dist[x_1, y_1]$.

References

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