Esercitazione 6: More on Greedy Algorithms and Dynamic Programming

Giacomo Paesani

April 16, 2025

Esercizio 1 (23.1-11, [1])**.** Sia $G = (V, E)$ un grafo non diretto e connesso con gli archi pesati da una funzione w e T un albero di copertura di G di peso minimo. Supponiamo che il peso di un'arco uv che non appartiene a T diminuisce. Fornire un algoritmo con tempo di esecuzione $\mathcal{O}(n)$, in pseudo-codice che dato G, T e come viene modificato il peso dei un arco uv, trova un albero di copertura di peso minimo nel grafo modificato (senza calcolarlo da capo).

Soluzione 1. Sia uv l'arco di G a cui viene cambiato peso da $w(uv)$ in $w'(uv)$. Dato che uv non appartiene a T, allora esiste un cammino P in T che collega u e v. La soluzione consiste nel capire se è c'è un arco st in P tale che $w(st) > w'(uv)$: se tale arco esiste allora è possibile trovare un nuovo albero di copertura di peso minimo. Altrimenti T resta minimo.

La soluzione proposta è quella data dal Algoritmo 1. Come prima cosa si esegue una ricerca in ampiezza BFS nell'albero T a partire da uno degli estremi dell'arco uv, ad esempio u: questa chiamata (Linea 5) restituisce il vettore dei padri. Tale vettore ci permette di ricostruire il cammino P in T tra u e v. A questo punto cerchiamo tra gli archi di P, grazie al ciclo **while** in Linea 10, quello di peso massimo: gli estremi e il peso di questo arco è salvato nelle variabili s, t e m. L'algoritmo termina sostituendo in T l'arco st ad uv. □

Esercizio 2 (23.2-7,[1])**.** Sia $G = (V, E)$ un grafo non diretto e connesso con gli archi pesati da una funzione w e T un albero di copertura di G di peso minimo. Supponiamo che viene aggiunto un nuovo vertice u e gli archi a se incidenti a G. Fornire un algoritmo in pseudo-codice che dato G, T e la

Algorithm 1 Algoritmo per aggiornare un'albero di copertura di costo minimo in seguito ad una riduzione di costo di un arco.

Input: $G = (V, E)$ grafo non diretto, connesso e con archi pesati, $T \subseteq E$, $uv \in E$ e valore w'

Output: $T' \subseteq E$ albero di copertura di peso minimo

global variables
    $Parent \hookleftarrow$ array dei padri
end global variables
function AdaptMST(G,T,uv,w')
    $Parent$=BFS(T,u)
    $m = w'$
    $s = u$
    $t = v$
    $z = v$
    while $Parent[z] \neq z$ **do**
        if $w(z, Parent[z]) > m$ **then**
            $m = w(z, Parent[z])$
            $s = z$
            $t = Parent[z]$
        $z = Parent[z]$
    $T.Remove(st)$
    $T.Add(uv)$
    return T

lista di adiacenza di u nei vertici di G, trova un albero di copertura di peso minimo nel grafo $G \cup \{u\}$ (senza calcolarlo da capo).

Soluzione 2. La problematica principale di questo esercizio è capire quali archi tra gli archi incidenti a u fanno parte di un albero di copertura di peso minimo T' nel grafo $G \cup \{u\}$ e quali tra gli archi di T non fanno parte di T'. Sia E_u l'insieme degli archi incidenti a u. L'idea è che è necessario aggiungere un arco di peso minimo $e^* \in E_u$; inoltre, per ogni arco $e \in E_u \setminus \{e^*\}$, $e \in T'$ se e solo se è possibile trovare un arco $e' \in T$ tale $(T \setminus \{e'\}) \cup \{e^*, e\}$ è un albero di copertura di $G \cup \{u\}$ di peso inferiore a $T \cup \{e^*\}$.

La soluzione proposta è data nel Algoritmo 2. Iniziamo con l'analisi della più semplice funzione che fa parte dell'esercizio MinEdge: dato un vertice u di un grafo, questa funzione mi restituisce un vicino z di u tale l'arco uz ha costo minimo (Linea 24). La funzione centrale è MaxInPath: dato un arco $uv \in E_u$, questa routine ci permette di individuare un arco di T che è il miglior candidato ad essere sostituito da uv. Infatti, dopo aver eseguito una DFS dell'albero T con radice u (Linea 10, viene percorso l'unico cammino da u a v in T, a partire da v con il ciclo **While** di Linea 12, e salviamo in $(z, Parent[z])$ un arco di peso massimo di tale cammino.

Algorithm 2 Algoritmo costruire un albero di copertura di costo minimo in seguito all'aggiunta di un nuovo vertice nel grafo.

Input: $G = (V, E)$ grafo non diretto e con archi pesati con funzione w, $T \subseteq E$ e nuovo vertice u
Output: albero di copertura di peso minimo
1: **function MST-NewVertex**(G,w,T,u)
2:     z=MinEdge(G,u)
3:     $T = T \cup \{uz\}$
4:     **for** $v \in Adj[u]$ **do**
5:         st=MaxInPath(G,T,u,v)
6:         **if** $w(st) > w(uv)$ **then**
7:             $T = (T \setminus \{st\}) \cup \{uv\}$
8:     **return** T
9: **function MaxInPath**(G,T,u,v)
10:     $Parent$=DFS(G,T,u)$\hookleftarrow$ DFS che ritorna il vettore dei padri
11:     $m = -\infty$
12:     **while** $Parent[v] \neq v$ **do**
13:         **if** $w(v, Parent[v]) > m$ **then**
14:             $m = w(v, Parent[v])$
15:             $z = v$
16:         $v = Parent[v]$
17:     **return** $(z, Parent[z])$
18: **function MinEdge**(G,u)
19:     $m = +\infty$
20:     **for** $v \in Adj[u]$ **do**
21:         **if** $m > w(u, v)$ **then**
22:             $m = w(u, v)$
23:             $z = v$
24:     **return** z

Guardiamo, in fine, la funzione MST-NewVertex che costituisce la base della soluzione proposta. Prima di tutto, aggiungiamo a T l'arco di costo minimo incidente a u (Linea 3 che troviamo dopo la chiamata alla funzione

MinEdge (Linea 2). Ora, per ogni arco uv incidente ad u troviamo l'arco di costo minimo st che può essere rimpiazzato da uv tramite la chiamata a MaxInPath di Linea 5. A questo punto confrontiamo i pesi di st e uv per decidere se aggiornare l'albero T o meno.

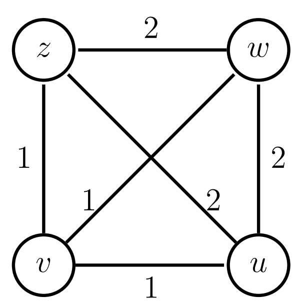
Concludiamo la spiegazione di questo esercizio con un analisi della complessità computazionale. La funzione MinEdge ha tempo di esecuzione $\mathcal{O}(|E|)$. La complessità della funzione MaxInPath è dominata dalla chiamata alla DFS che gira in tempo $\mathcal{O}(|V|)$ dato che come input ha l'albero T. In fine, la complessità della funzione MST-NewVertex è dominata dal ciclo **for** di Linea 4: per quanto detto prima questo ciclo consiste in al più $|E|$ chiamate alla funzione MaxInPath, ognuna di complessità $\mathcal{O}(n)$. Allora la complessità computazionale complessiva dell'algoritmo MST-NewVertex è di $\mathcal{O}(|V| \cdot |E|)$. □

Esercizio 3 (23.2-8, [1])**.** Sia $G = (V, E)$ un grafo non diretto, connesso e con pesi sugli archi dati da una funzione w. Consideriamo la seguente strategia ricorsiva per calcolare un albero di copertura di peso minimo di G. Partizioniamo l'insieme V in due sottoinsiemi V_1 e V_2 tali che $|V_1|$ e $|V_2|$ differiscono di al più uno. Sia E_1 l'insieme di archi che sono incidenti solo a vertici di V_1 e E_2 l'insieme di archi che sono incidenti solo a vertici di V_2. Ricorsivamente troviamo un albero di copertura di peso minimo per i due sottografi $G_1 = (V_1, E_1)$ e $G_2 = (V_2, E_2)$. In fine selezionare l'arco di peso minimo che attraversa il taglio (V_1, V_2) e aggiungerlo ai due alberi di copertura di peso minimo per i sottografi G_1 e G_2.

Dimostrare che la strategia proposta permette di implementare un algoritmo che trova correttamente una soluzione o esibire un contro-esempio che mostra come la strategia non sempre produce soluzioni ottime.

Soluzione 3. E' abbastanza immediato trovare un contro-esempio per cui la strategia proposta produce una soluzione non ottima. La figura qui sotto rappresenta un contro-esempio. Consideriamo una qualsiasi partizione, ad esempio data da $V_1 = \{u, z\}$ e $V_2 = \{v, w\}$ e troviamo che gli alberi di copertura di peso minimo per G_1 e G_2 sono dati da $T_1 = \{uz\}$ e $T_2 = \{vw\}$. Ora aggiungendo un arco leggero per il taglio (V_1, V_2), cioè un arco di costo minimo che ha un estremo in V_1 e l'altro in V_2, come ad esempio l'arco (u, v) otteniamo un albero di copertura $T = \{uv, uz, vw\}$ di costo 4. Notiamo che l'albero di copertura di costo minimo per G è dato da $\{uv, vz, vw\}$ di costo 3.

Il problema di questa strategia è la forte dipendenza della soluzione ottenuta dalla partizione scelta dell'insieme V. E' possibile avere una garanzia di ottenere una soluzione ottima al costo di un aumento di complessità dato dal uso della programmazione dinamica per poter controllare tutte le partizioni *bilanciate* di V? La risposta è nuovamente negativa e l'esempio in figura lo mostra. Più in generale non funziona la strategia del *divide-et-impera* suddividendo il problema in sottoproblemi su sottografi che formano una partizione. L'idea è che la strategia proposta tende a ignorare alcuni alberi di copertura che potrebbero essere ottimi come ad esempio quelli a *stella*, cioè quelli in cui ogni arco è incidente ad uno stesso vertice detto *centro*, come è la soluzione del esempio in figura: infatti, se due dei sottografi indotti dalla partizione connessi allora una soluzione a stella è impossibile da ottenere. Inoltre se almeno un sottografo indotto dalla partizione non è connesso, allora l'algoritmo non produce alcun albero.

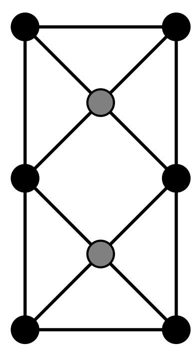


Esercizio 4 (33.4-3:4, [1])**.** Possiamo definire la distanza tra due punti in modi diversi, non solo usando quella euclidea. Sul piano, la distanza L_m tra due punti $p_1 = (x_1, y_1)$ e $p_2 = (x_2, y_2)$ è data dalla seguente espressione $d_m(p_1, p_2) = (|x_1 - x_2|^m + |y_1 - y_2|^m)^{1/m}$. Quindi, la distanza euclidea è esattamente la distanza L_2. Inoltre la distanza L_∞ è definita nella seguente maniera: $d_\infty(p_1, p_2) = \max(|x_1 - x_2|, |y_1 - y_2|)$

Modificare l'algoritmo per trovare la coppia di punti più vicini nel piano usando le distanze L_1 e L_∞.

Soluzione 4. Per qualsiasi scelta della distanza L_m, l'algoritmo rimane molto simile a quello usato per la distanza L_2. In particolare per la parte del *Divide* e *Conqueror* rimane invariata. Cambia leggermente l'implementazione della parte di *Combine*. Infatti, cambiando la distanza usata, possono variare il numero massimo degli elementi di P da analizzare per trovare una soluzione.

Sia P un insieme di punti del piano e una linea verticale $\ell : x = x^*$ che partiziona P in P_L e P_R in maniera che soddisfa la procedura *Divide* dell'algoritmo. Supponiamo che la distanza minima tra due punti ottenuta dai sotto-problemi su P_L e P_R è δ e sia $p = (x_L, y_L)$ un punto di P_L. L'idea è di controllare se esiste un punto $p' = (x_R, y_R) \in P_R$ tale che $d(p, p') = \delta' < \delta$. Allora necessariamente dobbiamo avere le seguenti condizioni: $x^* - \delta \leq x_L \leq x^* \leq x_R \leq x^* + \delta$ e $y_L - \delta \leq y_R \leq y_L + \delta$. Quanti diversi punti p' di P_R possono soddisfare contemporaneamente queste condizioni? Ricordiamo che tali punti devono avere distanza reciproca almeno δ. Per ottenere una stima (per eccesso) di questo numero dobbiamo considerare la specifica distanza utilizzata. Se la distanza adottata è la L_2 o la L_∞ allora questo numero è al più 6 come mostrato nella figura dai nodi neri. Se, invece, la distanza adottata è la L_1 allora questo numero è al più 8 come mostrato dalla figura dei nodi neri e grigi. $\square$



References

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