11 Cammini minimi, sorgente singola

Nota. *La prima cosa da fare quando si progetta un algoritmo è capire qual è la struttura dati adatta a risolvere quel particolare problema.*

11.1 Introduzione

Definizione 11.1 (costo del cammino)**.** *Dato un cammino* $p = \langle v_1, v_2, \dots, v_k \rangle$ *con* $k > 1$*, il* costo del cammino *è dato da*

$$w(p) = \sum_{i=2}^{k} w(v_{i-1}, v_i)$$

Definizione del problema Dati in input un grafo orientato $G = (V, E)$, un nodo sorgente s ed una funzione di peso $w\colon E \rightarrow R$ (che associa ad ogni arco un numero reale che rappresenta il peso). Trovare un cammino da s ad u, per ogni nodo $u \in V$, il cui costo sia minimo, ovvero più piccolo o uguale del costo di qualunque altro cammino da s a u.

Nota. *Non ci limitiamo a trovare un solo percorso ma tutti i cammini da un nodo a tutti gli altri nodi.*

Panoramica sul problema Nella risoluzione del problema del cammino minimo fra una coppia di vertici si risolve il problema di cammmini minima da sorgente unica (si trovano tutti i cammini che partono da un nodo) e si estrae il cammino richiesto. Per quanto riguarda il *caso pessimo* non si conoscono algoritmi che abbiamo tempo di esecuzione migliore.
In alcuni casi, gli archi possono avere peso negativo. Questo influisce sul problema (se e ben definito oppure no) e sulla soluzione (in assenza di archi negativi si devono utilizzare tecniche diverse).
Nell'algoritmo di Dijkstra si suppone che tutti gli archi abbiano peso positivo, mentre nell'algoritmo di Bellman-Ford gli archi possono avere peso negativo, ma non possono esistere cicli di peso negativo.

Nota. *Possiamo ammettere pesi negativi, ma non cicli negativi.*

Se esiste un ciclo di peso negativo raggiungibile dalla sorgente, non esistono cammini finiti di peso minimo; per qualunque cammino, bastera passare per un ciclo negativo più volte per ottenere un ciclo di costo inferiore.
Ovviamente, in un cammino minimo *non è possibile sia presente un ciclo di peso positivo.* I cicli di peso nullo possono essere banalmente eliminati dal cammino minimo, in quanto inutili e ridondanti.

11.1.1 Sottostruttura ottima

Definizione 11.2 (albero dei cammini minimi)**.** *L'*albero dei cammini minimi *è un albero di copertura radicato in* s *avente un cammino da* s *a tutti i nodi raggiungibili da* s*.*

Nota. *Non bisogna far confusione con gli algeri di copertura di peso minimo.*

Soluzione ammissibile Una soluzione *ammissibile* può essere descritta da un *albero di copertura* T radicato in s e da un *vettore di distanza* d, i cui valori $d[u]$ rappresentano il costo del cammino da s a u in T.

(a) Soluzione ammissibile (albero di *peso* minimo)

(b) Soluzione ottima (albero dei *cammini* minimi)

Figura 1: Notare la differenza fra i due alberi

Nota. *In questo problema devo trovare i percorsi che minimizzano il peso fra un nodo e tutti gli altri nodi, e* ***non****, come sembra spontaneo fare, l'albero che ha complessivamente peso minimo.*

Rappresentazione dell'albero Per rappresentare l'albero, utilizziamo la rappresentazione basata su vettore dei padri, così come abbiamo fatto con le visite in ampiezza/profondità.

11.1.2 Teorema di Bellman

Teorema 1 (Teorema di Bellman)**.** *Una soluzione ammissibile T è (anche) ottima se e solo se:*

$$d[v] = d[u] + w(u,v) \qquad \textit{per ogni arco } (u,v) \in T$$
$$d[v] \leqslant d[u] + w(u,v) \qquad \textit{per ogni arco } (u,v) \in E$$

Dimostrazione per assurdo (parte 1). Sia T una soluzione ottima. Consideriamo un qualunque arco $(u,v) \in E$ e sia $\mathsf{w}(u,\ w)$ la sua lunghezza.
Ovviamente se $(u,v) \in T$, allora $d[v] = d[u] + w(u,v)$. Invece se $(u,v) \notin T$, allora poiché T è ottimo, deve risultare $d[v] \leqslant d[u] + w(u,v)$, poiché altrimenti esisterebbe nel grafo G un cammino da s a v più corto di quello in T, che è *assurdo* perché abbiamo ipotizzato che T fosse ottima. □

Dimostrazione per assurdo (parte 2). Supponiamo per assurdo che il cammino da s a u in T non sia ottimo. Allora esiste un cammino da s a u con distanza $d'[u] < d[u]$. Sia $d'[v]$ la distanza da s ad un generico nodo v che appare in tale cammino. Poichè $d'[s] = d[s] = 0$, ma $d'[u] < d[u]$, esiste un arco (h,k) per cui $d'[h] \geqslant d'[h]$ e $d'[k] < d[k]$. Per costruzione $d'_h + w(h,k) = d'_k$. Per ipotesi $d_h + w(h,k) \geq d_k$. Combinando queste due relazioni, si ottiene:

$$d'_k = d'_h + w(h,k) \geqslant d_h + w(h,k) \geqslant d_k$$

che contraddice l'ipotesi. □

11.1.3 Verso un algoritmo

Algoritmo prototipo L'algoritmo prende in input un Graph e il Node sorgente. I pesi vengono estratti dalla struttura dati Graph. Inizializziamo d con una sovrastima della distanza; con $d[s] = 0$ dico che la sorgente ha distanza da sè stessa pari a 0 (caso base) e con $d[x] = +\infty$ dico che la distanza di tutti gli altri nodi, fintanto che non è nota, è pari a $+\infty$.

Nota. *Se al termine dell'esecuzione qualche nodo mantiene una distanza infinita, esso non è raggiungibile.*

Algoritmo generico **bool**[ ] ci permette di sapere in tempo costante se un certo nodo appartiene ad una struttura dati oppure no, non è necessario quando si implementa realmente il codice.

11.2 Dijkstra

Il seguente algoritmo è stato sviluppato da Edsger W. Dijkstra nel 1956, pubblicato nel 1959. Nella versione originale veniva utilizzato per trovare la distanza minima fra due nodi sfruttando il concetto di coda con priorità. Tieni conto però che le code di priorità basate sulle PRIORITYQUEUE sono state proposte nel '64, infatti l'algoritmo che di solito viene considerato di Dijkstra in realtà è la versione modificata di Johnson.

Implementazione L'algoritmo utilizza una coda con priorità basata su vettore.

```
    int[ ], int[ ] CamminiMinimi(GRAPH G, NODE s)
(1)     PRIORITYQUEUE S ← PriorityQueue // O(n)·1
        S.inserisci(s, 0)
        finché not S.isEmpty() fai // O(n)
(2)         // O(n) vettore ordinato / O(log n) heap binario
            int u ← S.deleteMin
            b[u] ← falso
            per ciascun v ∈ G.adj(u) fai
                se d[u] + G.w(u,v) < d[v] allora
                    se not b[v] allora
(3)                     // O(1)·n vettore ordinato / O(log n)·n heap binario
                        b[v] ← vero
                    altrimenti
(4)                     // O(1)·m vettore ordinato / O(log n)·m heap binario
                    T[v] ← u
                    d[v] ← d[u] + G.w(u,v)
        ritorna (T, d)
```

Commento sulla complessità (1) Viene creato un vettore di dimensione n. Ogni elemento u-esimo rappresenta il nodo u. Le priorità (distanze) vengono inizializzate ad $+\infty$. La priorità di s è posta uguale a 0. Per un costo computazionale di $O(n)$. (2) Si ricerca il minimo all'interno del vettore, una volta trovato si "cancella" la sua priorità. Per un costo computazionale di $O(n)$. (3) Si registra la priorità nella posizione corrispondente all'indice v. Per un costo computazionale di $O(1)$. (4) Si aggiorna la priorità nella posizione corrispondente all'indice v. Per un costo computazionale di $O(1)$.

Commento Tutte le volte che estraiamo un nodo, quel nodo ha una distanza (priorità) positiva ed estraiamo nodi a distanza progressivamente crescenti. Se estraggo un nodo dalla coda tutti gli altri nodi hanno distanze più grandi. Tutte le volte che estraggo un nodo la sua distanza non può più essere modificata. Ed è questo il motivo per cui l'algoritmo di Dijkstra funziona bene solo con pesi positivi.

Nota. *L'algoritmo di Dijkstra funziona bene solo con pesi positivi.*

11.2.1 Correttezza per pesi positivi

Ogni nodo viene estratto una e una sola volta. Al momento dell'estrazione la sua distanza è minima.

Dimostrazione per induzione sul numero k di nodi estratti. Per $k = 0$ (caso base) è vero poiché $d[s] = 0$ e non ci sono lunghezze negative. Supponiamo che sia vero per i primi $k-1$ nodi (ipotesi induttiva). Quando viene estratto il k-esimo nodo u, la sua distanza $d[u]$ dipende dai $k-1$ nodi già estratti (passo induttivo). Non può quindi dipendere dai nodi ancora da estrarre, che hanno distanza $\geqslant d[u]$. Di conseguenza $d[u]$ è minimo e u non verrà più re-inserito, perché non ci sono distanze negative. □

11.3 Johnson

Analisi della complessità $\mathcal{O}(n^2 + m)$ ma siccome $m = \mathcal{O}(n^2)$, allora il costo è $\mathcal{O}(n^2)$, con l'introduzione dell'heap binario nel '64 le operazioni che prima venivano svolte con complessità $\mathcal{O}(n)$ sul vettore ordinato passano ad avere una complessità di $\mathcal{O}(\log n)$. Di conseguenza la complessità dell'algoritmo diventa $\mathcal{O}(m \log n)$.

Per *grafi densi* non conviene utilizzare uno heap binario in quanto $m = \Theta(n^2)$ e di conseguenza l'algoritmo avrebbe una complessità di $\mathcal{O}(n^2 \log n)$, mentre per *grafi sparsi* $m = \Theta(n)$ e l'algoritmo $\mathcal{O}(n \log n)$.

11.4 Fredman-Tarjan

Analisi della complessità Sfruttando un heap di fibonacci l'operazione di decrease ha costo ammortizzato costante; così facendo hanno abbassato la complessità a $\mathcal{O}(m + n \log n)$. Per *grafi sparsi* produce un miglioramento nella complessità.

11.5 Bellman-Ford-Moore

Nota. *É computazionalmente più pesante dell'algoritmo di Dijkstra ma può lavorare anche con archi di peso negativo.*

Implementazione Coda senza priorità.

```
int[], int[] CamminiMinimi(GRAPH G, NODE s)
(1)     QUEUE S ← Queue
        S.enqueue(s)  // metto in coda la sorgente
        finché not S.isEmpty fai  // O(n)
(2)         int u ← S.dequeue  // O(1·n)
            b[u] ← falso
            per ciascun v ∈ G.adj(u) fai
                se d[u] + G.w(u,v) < d[v] allora
                    se not b[v] allora
(3)                     // lo metto in cosa quando c'è un miglioramento
                        S.enqueue(v)  // O(m·n)
                        b[v] ← vero
                    T[v] ← u
                    d[v] ← d[u] + G.w(u,v)
        ritorna (T, d)
```

(1) Viene creata una coda di dimensione n. Per un costo computazionale di $O(n)$. (2) Viene estratto il prossimo elemento della coda. Per un costo computazionale di $O(1)$. (3) Si inserisce l'indice v in coda. Per un costo computazionale di $O(1)$.

L'inserimento in coda può essere fatto più di una volta durante il ciclo di esecuzione dell'algoritmo, al contrario di quel che accade nell'algoritmo di Dijkstra. Il passo (4) non è necessario in quanto non c'è una priorità da aggiornare.
Il vettore delle distanze è rappresentato dalle colonne, su fondo potete vedere lo stato della coda.

Dimostrazione di correttezza

Analisi di complessità

Cammini minimi su DAG I cammini minimi su DAG sono sempre ben definiti; anche in presenza di pesi negativi, in quanto non esistono cicli (nè tantomeno quelli negativi). É possibile rilassare gli archi *in ordine topologico*, *una volta sola*. Non essendoci cicli, non c'è modo di tornare su un nodo già visitato ed abbassare il valore della sua distanza (il suo campo d).
Si utilizza quindi l'ordine topologico.

11.5.1 Riassumendo

Tabella 1: Quale complessità preferire?

Algoritmo	**Complessità**	**Input**
Dijkstra	$\mathcal{O}(n^2)$	Pesi positivi, grafi denso
Johnson	$\mathcal{O}(m \log n)$	Pesi positivi, grafi sparso
Fredman-Tarjan	$\mathcal{O}(m + n \log n)$	Pesi positivi, grafi denso, dimensioni molto grandi
Bellman-Ford	$\mathcal{O}(m \cdot n)$ $\mathcal{O}(m + n)$	Pesi negativi DAG
BFS	$\mathcal{O}(m + n)$	Senza pesi

11.5.2 Cammini minimi, sorgente multipla

Vogliamo cercare i cammini minimi fra tutti i nodi.

Tabella 2: Quale complessità preferire?

Input	**Complessità**	**Approccio**
Pesi positivi, grafo denso	$\mathcal{O}(n \cdot n^2)$	Applicazione ripetuta (n) dell'algoritmo di Dijkstra
Pesi positivi, grafo sparso	$O(n \cdot (m \log n))$	Applicazione ripetuta dell'algoritmo di Johnson
Pesi negativi	$O(n \cdot nm)$	Applicazione ripetuta di Bellman-Ford, *sconsigliata*
Pesi negativi, grafo denso	$O(n^3)$	Algoritmo di *Floyd e Warshall*
Pesi negativi, grafo sparso	$O(nm \log n)$	Algoritmo di *Johnson per sorgente multipla*

L'algoritmo di Bellman-Ford è sconsigliato per grafi densi perché può arrivare ad avere una complessità di $\mathcal{O}(n^4)$, mentre l'algoritmo di Floyd e Warshall ha una complessità di $\mathcal{O}(n^3)$ indipendentemente dalla forma del grafo.

11.6 Floyd-Warshall

Utilizza la programmazione dinamica. Ci riesce ridefinendo la definione del costo di cammino in modo tale che possa essere calcolato in modo ricorsivo.