Cammini minimi e algoritmo di Dijkstra

Cammini minimi a singola sorgente

Sia $G = (V, E, w)$ un grafo orientato o non orientato cone pesi w reali sugli archi. il costo o lunghezza di un cammino $\pi =< v_0, v_1, v_2, \ldots, v_k >$ è:

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

Un **Cammino minimo** tra un coppia di vertici x e y è un cammino avente costo minore o uguale a quello di ogni altro cammino tra gli stessi vertici.

> **Nota**
>
> Il cammino minimo non è necesssariamente unico.

Andiamo a scrivere il costo di un cammino minimo in questo modo:

$$d_G(u, v)$$

Quindi la distanza minima d in G tra u e v.

> **Non esiste sempre un cammino minimo tra due nodi**
>
> - Se non esiste un cammino da u a v, allora $d(u, v) = +\infty$
> - Se c'è un cammino che contiene un ciclo raggiungibile il cui costo è negativo, allora $d(u, v) = -\infty$

> **Proprietà**
>
> Ogni sottocammino di un cammino minimo è un cammino minimo.
> Inoltre vale la disugualianza triangolare: $\forall u, v, x \in V, d(u, v) \leq d(u, x) + d(x, v)$

Quindi il problema del calcolo dei cammini minimi a singola sorgente lo risolviamo trovando il cammino minio da una sorgente a tutti gli altri nodi del grafo. Esistono due varianti del problema:

- Dato $G = (V, E, w), s \in V$, calcola le distanze di tutti i nodi da s, ovvero $d_G(s, v) \forall v \in V$.
- Dato $G = (V, E, w), s \in V$, calcola l'albero dei cammini minimi di G radicato in s.

Albero dei cammini minimi

T è un **SPT(Shortest Path Tree)** con sorgente s di un grafo $G = (V, E, w)$ se:

- T è un albero radicato in s
- $\forall v \in V$ vale $d_T(s, v) = d_G(s, v)$

> **Nota**
>
> Per grafi non pesati, SPT radicato in s è uguale all'albero BFS radicato in s.

Algoritmo di Dijkstra

> **Assunzione**
>
> Tutti gli archi hanno peso non negativo.

Spiegazione Algoritmo:

1. Manteniamo per ogni nodo v una stima (per eccesso), D_{sv} alla distanza $d(s, v)$. Inizialmente l'unica stima è $D_{ss} = 0$.
2. Manteniamo un insieme X di nodi per cui le stime sono esatte, e anche un albero T dei cammini minimi verso i nodi in X. Inizialmente $X = [s]$ e T non ha archi.
3. Ad ogni passo aggiungiamo a X il nodo u in $V - X$ la cui stima è minima; Aggiungiamo a T uno specifico arco entrante in u.
4. Aggiorniamo le stime guardando i nodi adiacenti a u.

I nodi da aggiungere a X e poi T sono manenuti in una coda di priorità, associati ad un unico arco che li connette a T.

La stima per un nodo $y \in V - X$ è $D_{sy} = min[D_{sx} + w(x, y) : (x, y) \in E, x \in X)]$, se y è in coda con un arco (x, y) associato, e se dopo aver aggiunto u a T troviamo un arco (u, y) tale che $D_{su} + w(u, y) < D_{sx} + w(x, y)$ allora rimpiazziamo (x, y) con (u, y) ed aggiorniamo D_{sy}

Pseudocodice:

```
algoritmo Dijkstra(grafo G, vertice s)--> albero
        for each (vertice u in G) do D_su = +inf
        T <-- albero dormato dal solo nodo s
        X <-- empty
        CodaPrioritaria S
        D_SS = 0
        S.insert(s,0)
        while(not S.isempty()) do:
                u <-- S.dekteMin()
                X <-- X U {u}
                for each (arco(u,v) in G) do:
                        if(D_SV = +inf) then
                                S.insert(v,D_SU + w(u,v))
                                D_SV <-- D_SU + w(u,v)
                                rendi u padre di v in T
                        else if(D_SU + w(u,v) < D_SV) then:
                                S.decreaseKey(v, D_SV - D_SU - w(u,v))
                                D_SV <-- D_SU + w(u,v)
                                rendi u nuovo padre di v in T
        return T
```

Correttezza

Lemma

Quando il nodo v viene estratto dalla coda con priorità vale:

- $D_{sv} = d(s,v)$ --> Stima esatta
- Il cammino da s a v nell'albero corrente ha costo $d(s,v)$ --> Camm. min in G

Dimostrazione sulle slide.

Analisi della complessità

Se andiamo ad Escludere le operazione sulla coda di priorità abbiamo tempo $O(m+n)$
Il problema della complessità nasce dal tipo di struttura che usiamo per impletare l'algoritmo.

Tempo di esecuzione: Implementazioni elementari

Ricordiamo i costi delle operazioni che ci servono:

	Insert	DelMin	DecKey
Array non ordinato	$O(1)$	$O(n)$	$O(1)$
Array ordinato	$O(n)$	$O(1)$	$O(n)$
Lista non ordinata	$O(1)$	$O(n)$	$O(1)$
Lista ordinata	$O(n)$	$O(1)$	$O(n)$

Quindi:

- $n * O(1) + n * O(n) + O(m) * O(1) = O(n^2)$ con array non ordinati;
- $n * O(n) + n * O(1) + O(m) * O(m) = O(m * n)$ con array ordinati;
- $n * O(1) + n * O(n) + O(m) * O(1) = O(n^2)$ con liste non ordinate;
- $n * O(n) + n * O(1) + O(m) * O(n) = O(m * n)$ con liste ordinate;

Tempo di esecuzione: Implementazioni efficienti

Ricordiamo i costi delle operazioni che ci servono:

	Insert	DelMin	DecKey
Heap binario	$O(log(n))$	$O(log(n))$	$O(log(n))$
Heap Binomiale	$O(log(n))$	$O(log(n))$	$O(log(n))$
Heap di Fibonacci	$O(1)$	$O(log(n))*$	$O(1)*$

Quindi:

- $n * O(log(n)) + n * O(log(n)) + O(m) * O(log(n)) = O(m * log(n))$ con heap binari o binomiali.
- $n * O(1) + n * O(log(n)) * + O(m) * O(1)* = O(m + nlog(n))$ con heap di Fibonacci.

La soluzione migliore è quella con l'Heap di Fibonacci, che non sarà mai peggiore delle altre implementazioni e a volte è anche meglio. Quindi la complessità è di:

$$O(m + nlog(n))$$