16'287.89$
Come mi sono indebitato con Google...

Matteo Fasulo, Simone Flavio Paris, and Matteo Sivoccia

Contents

ABSTRACT

Il progetto nasce dall'idea di cercare di elaborare e calcolare i tempi di distribuzione e rifornimento di un vaccino in Italia a partire da una sorgente.
Per giungere al risultato finale siamo passati al teorizzare prima due modelli che ci hanno permesso di capire punti di forza e difetti da risolvere. L'ultimo modello è quello che reputiamo più sensato, ma dato l'enorme lavoro di programmazione dietro agli altri abbiamo deciso di lasciarli nel codice eseguibile per puro scopo dimostrativo.

1. INTRODUCTION

1.1 Modelli

I modelli da noi sviluppati sono così suddivisi:

1. Si assume che tutti i comuni (nodi) siano fortemente connessi tra loro con distanza (peso) pari al numero di Km che li separa considerando la formula dell'emisenoverso Haversine.[1]
   Questo modello non rendeva necessario l'uso di algoritmi per individuare i cammini minimi in quanto ogni coppia essendo linearmente collegata era già il miglior cammino.

2. Considera solo le ASL principali (circa 70 nodi) e calcola i pesi con i servizi (a pagamento) di Google, prendendo l'esatto tragitto in km e il tempo di percorrenza media. Al suo avvio il programma richiede il nodo di partenza e restituisce una mappa con:
   - nodo di partenza,
   - nodo più lontano in KM,
   - nodo più lontano in minuti,
   - il migliore nodo sorgente dal quale partire,
   - tutti gli altri nodi,
   identificati tramite colori diversi.

3. Non potendo (e volendo) più usare i sistemi di Google per il calcolo dei pesi abbiamo definito una gerarchia: Capoluoghi regione > Capoluoghi Provincia > Semplice Comune:
   in questo modo solo i nodi dei capoluoghi regionali sono fortemente connessi tra loro e a partire da ogni capoluogo regionale è possibile arrivare ai capoluoghi di provincia e successivamente ai comuni. Attraverso questa limitazione possiamo ottenere percorsi diversi e utilizzare il calcolo dei cammini minimi in base al comune di partenza scelto. Per questo modello abbiamo utilizzato poco meno di 8000 comuni e l'operazione più lunghe sono state quelle di pulizia dei dati ed elaborazione sotto forme diverse e più complete per poi arrivare alla matrice quadrata da ordinare con gli algoritmi su cui baseremo l'intera analisi che ci siamo prefissati.

2. RESULTS

2.1 Tabulare

Per la realizzazione del calcolo dei cammini minimi, è stata utilizzata la libreria *scipy* e in particolare il modulo *sparse/csgraph*. Attraverso questo modulo, abbiamo implementato i tre diversi algoritmi e computato il loro tempo di esecuzione. Riassumiamo il risultato sia in forma tabulare che grafica:

2.1.1 Risultato

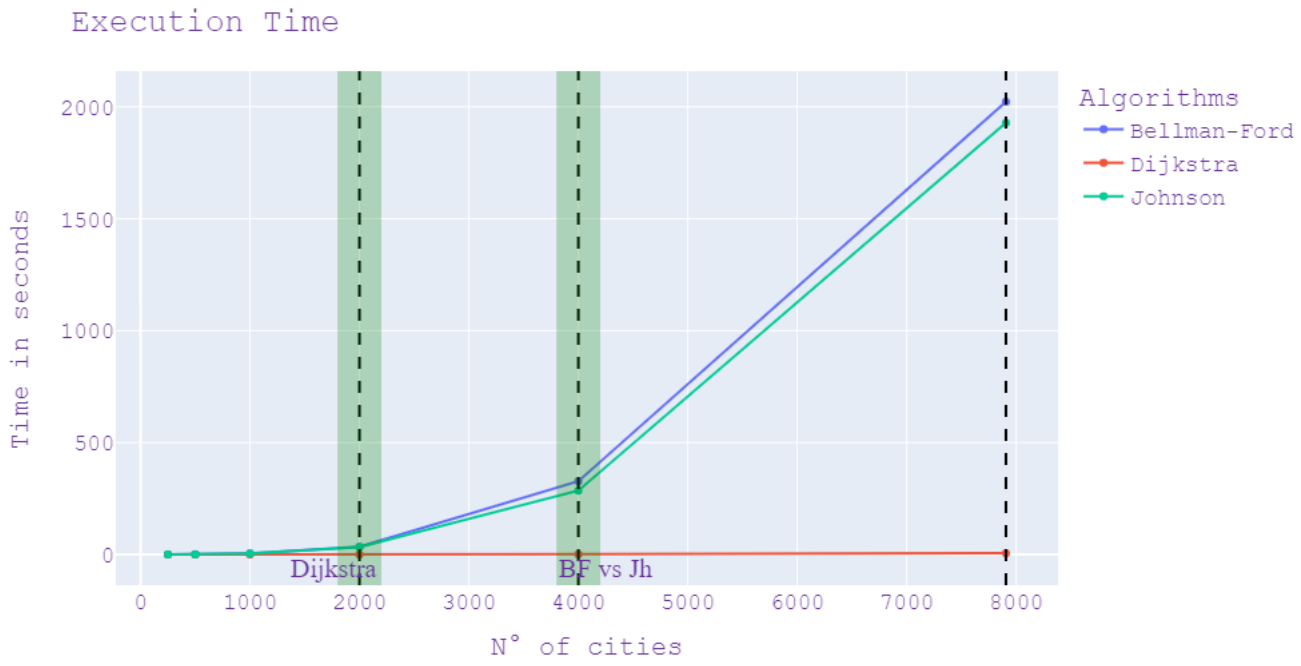
Table 1. Tempi di esecuzione (sec)

	250	500	1000	2000	4000	7906
Dijkstra (fibonacci)	0.00299	0.006	0.03798	0.19501	0.88201	5.58
Bellman-Ford	0.07099	0.49601	4.1799	34.2439	327.24	2022.419
Johnson	0.05801	0.51898	4.0939	33.099	285.228	1928.418

2.2 Grafico

2.2.1 Risultato

Il grafico complessivo dei tempi:



2.3 Commento

Abbiamo da subito notato la bontà di Dijkstra rispetto agli algoritmi di Bellman-Ford e Johnson:

1. Abbiamo osservato che l'algoritmo di **Bellman-Ford** fa tipicamente un grande numero di operazioni di rilassamento del tutto inutili, sopratutto nel nostro caso in cui il nostro grafo non ha mai valori negativi (per definizione di distanza).
   Se abbiamo **m** archi nel grafo, ogni passata richiede tempo $\mathcal{O}(m)$ e quindi il tempo totale richiesto per elaborare gli **n** vertici è $\mathcal{O}(nm)$

2. L'algoritmo di **Johnson** esegue una combinazione di Bellman-Ford e Dijkstra per trovare velocemente il cammino minimo con robustezza verso i cicli negativi. Se viene individuato un ciclo negativo, ritorna un errore e l'algoritmo non prosegue (invece che terminare l'esecuzione del programma). Per grafi con assenza di archi pesati negativi, dijkstra performerà sicuramente meglio.

3. **Dijkstra** effettua le operazioni di rilassamento in ordine topologico e per questo considera ogni arco di un grafo aciclico esattamente una volta e non **n** volte come Bellman-Ford. Il tempo di esecuzione dell'algoritmo di Dijkstra è dominato dalle operazioni sulla coda con priorità S, e quindi dipende dal tipo di struttura dati scelta per implementarla. Inoltre grazie all'implementazione mediante code con priorità (heap di Fibonacci) possiamo estrarre l'arco minimo in tempo $\mathcal{O}(\log n)$ e questo ci permette di risolvere il problema nel caso peggiore in tempo $\mathcal{O}(m + n \log n)$ piuttosto che in $\mathcal{O}(mn)$

2.3.1 Algoritmi

Table 2. Tempi di esecuzione degli algoritmi, in grassetto quelli proposti

Nome	Tempo
Dijkstra	$\mathcal{O}(mn)$
Dijkstra (d-heap, $d = 2$))	$\mathcal{O}(m \log n)$
Dijkstra (fibonacci)	$\mathcal{O}(m + n \log n)$
Bellman-Ford	$\mathcal{O}(nm)$
Johnson	$\mathcal{O}(n^2 \log n + nm)$

REFERENCES

[1] Inman, J., "Haversine formula." 1835 `https://en.wikipedia.org/wiki/Haversine_formula`.