Progetto di Laboratorio di Algoritmi 2022-2023

Andrea Spinelli, Raffaele Terracino, Marco Valenti

7 Giugno 2023

Indice

1 Traccia del problema **3**

2 Analisi del problema **4**

3 Soluzione del problema **5**
3.1 Algoritmo 5
3.2 Correttezza 5
3.3 Ottimizzazione 6
3.4 Complessità 7

4 Implementazione in C++ **7**

1 Traccia del problema

Appassionato di opere d'arte moderna, Mario ha deciso ha deciso che vorrebbe visitare la città di New York. Mario vive a Palermo e vorrebbe trovare il volo o i voli più economici che lo porterebbero a New York. Sa che un volo diretto da Palermo a New York, se esistesse, sarebbe incredibilmente costoso; quindi, è disposto a tollerare un certo numero di scali. Tuttavia, le compagnie aeree sono molto numerose e propongono voli tra diverse città, il che rende molto difficile per Mario trovare il modo più economico per arrivare a New York! Puoi scrivere un programma per aiutare Mario a pianificare il suo percorso?

Vi verrà fornito un elenco di città tra Palermo e New York, incluse entrambi. Vi verrà anche fornito un elenco di voli tra le coppie di città, e il costo associato per ogni volo, tasse incluse. Non ci sarà mai un volo da Palermo a New York: Mario ha già scartato quei voli, ritenendoli una perdita di tempo e denaro. Tieni presente, però, che potrebbero esserci più voli tra due città, in quanto Mario sta considerando voli da tutte le compagnie aeree.

Infine, vi verrà presentato un numero di query. Ogni query è un singolo numero intero che indica il numero massimo di scali che Mario è disposto a tollerare. Per ogni query, il vostro programma deve calcolare il costo totale minimo dei voli che porterebbero Mario da Palermo a New York con non più di un certo numero di scali richiesti.

Input
La prima riga dell'input contiene un singolo numero che indica il numero di scenari da elaborare. Una riga vuota precede ogni scenario.
Ogni scenario inizia con un numero $N(N \geq 2)$, il numero di città, seguito da N righe contenenti i nomi delle città. Il nome di una città è una stringa di lettere maiuscole e minuscole.
Successivamente, viene fornito un numero $M(M \geq 0)$, il numero di voli disponibili, seguito da M righe che descrivono i voli. Ogni volo è descritto dalla città di partenza, dalla città di destinazione e da un numero intero che rappresenta il costo in euro.
L'ultima riga inizia con Q $(Q \geq 1)$, il numero di query, seguito da Q numeri che indicano il numero massimo di scali.

Output
Per ogni scenario, il vostro programma dovrebbe, inserire una riga vuota tra gli scenari e, restituire il numero dello scenario, seguito dal costo totale minimo dei voli per ogni query.
Se nessun volo può soddisfare una query, scrivi "`No flights`".

Esempio di input:

```
2

Palermo
Madrid
Roma
NewYork
6
Palermo Madrid 125
Palermo Roma 300
Madrid NewYork 325
Madrid Roma 100
Palermo NewYork 875
Roma NewYork 175
3 2 1 0

3
Palermo
Montreal
NewYork
2
Palermo Montreal 300
Montreal NewYork 325
1 0
```

Esempio di output:

```
Scenario #1
Total cost of flight(s) is €400
Total cost of flight(s) is €450
Total cost of flight(s) is €875

Scenario #2
No flights
```

2 Analisi del problema

È possibile formalizzare il problema facendo uso di grafi pesati, in modo tale che ogni città in input corrisponda a un vertice, mentre un volo è rappresentato da un arco tra due città. Il costo di un arco da v a s è il prezzo del volo tra le città v ed s. Poichè non ci sarà mai un arco tra il vertice "Palermo" e il vertice "New York", il numero di scali si definisce come il numero di vertici interni tra i due. Ci si riferirà a tale grafo come "grafo dei voli".
Una prima osservazione da fare è che il grafo descritto nel problema potrebbe non essere semplice. Tuttavia lo si può rendere tale in fase di costruzione. Date le città v ed s e l'insieme di archi pesati tra i due, si può scegliere l'arco di minor costo, poichè l'unico parametro del problema relativo ai voli è proprio il loro costo.
Per cui il problema si può formalizzare come segue.

Problem 1. *Sia $G = (V, E)$ un grafo diretto semplice, con funzione di costo $w : E \to \mathbb{R} \setminus 0$. Dati $v, s \in V$ tali che $(v, s) \notin E$, e $n \in \mathbb{N}$, il problema consiste nel trovare il cammino di costo minimo da v a s con al più n vertici nel cammino, estremi esclusi.*

Il problema si può scomporre in due sottoproblemi. Infatti, trovare il cammino di costo minimo tra due vertici con al più n vertici di mezzo significa trovare i cammini di costo minimo tra i due vertici con esattamente k vertici, per $k = 1, \ldots, n$ e prendere il minimo tra essi.

3 Soluzione del problema

3.1 Algoritmo

L'approccio utilizzato per la risoluzione del problema si basa sulla programmazione dinamica. Il punto di partenza è l'algoritmo di Floyd-Warshall, che permette di calcolare i cammini di costo minimo tra tutte le coppie di vertici in tempo $O(n^3)$.

Algorithm 1 - Floyd-Warshall

Require: grafo diretto e pesato G come matrice di adiacenza
1: inizializza D tale che $D_{xy} = w(x, y)$ se $(x, y) \in E$, e $D_{xy} = +\infty$ altrimenti
2: **for each** $v \in V$ **do**
3: **for each** $(x, y) \in V \times V$ **do**
4: **if** $D_{xv} + D_{vy} < D_{xy}$ **then**
5: $D_{xy} \leftarrow D_{xv} + D_{vy}$

L'idea è di aggiungere una terza dimensione all'array D, in modo tale che D_{xye} rappresenti il cammino di costo minimo tra x e y con esattamente e archi.
Una volta calcolato tale array, per ottenere il cammino minimo tra i due vertici con al più n archi basterà fare il minimo, per $e = 1, ..., n$ di D_{xye}. L'algoritmo proposto è il seguente:

Algorithm 2 - minStopovers - STV

Require: grafo diretto e pesato G come matrice di adiacenza, $v, s \in V$, $k > 0$
1: inizializza D tale che $D_{xye} = 0$ se $e = 0$ e $x = y$,
2: $D_{xye} = w(x, y)$ se $e = 1$ e $w(x, y) \neq +\infty$, e $D_{xye} = +\infty$ altrimenti
3: **for** $e = 2, \dots k$ **do**
4: **for each** $i \in V$ **do**
5: **for each** $j \in V$ **do**
6: $D_{ije} \leftarrow +\infty$
7: **for each** $a \in V$ **do**
8: **if** $D_{ije} > D_{aje-1} + w(i, a)$ **then** ▷ Esegui solo se $a \neq i \neq j$ e $w(i, a) \neq +\infty$
9: $D_{ije} \leftarrow D_{aje-1} + w(i, a)$
10: **return** $\min_{1 \leq e \leq k} D_{vse}$

3.2 Correttezza

L'algoritmo presentato è un algoritmo di programmazione dinamica fortemente ispirato all'algoritmo di Floyd-Warshall. Come descritto prima, il problema si scompone in due fasi: trovare il cammino di costo minimo con esattamente i vertici, $1 \leq i \leq k$ e prendere il minimo tra tali valori. Si noti che il problema richiede il cammino di costo minimo con al più i vertici tra "Palermo" e "New York", ma basta osservare che tale valore è il cammino di costo mininimo con al più $i + 1$ archi. Con tale osservazione è possibile calcolare il cammino minimo tenendo in considerazione gli archi, per poi operare sugli indici.
Si supponga di aver calcolato, per ogni coppia di vertici, il cammino di costo minimo con esattamente $i - 1$ archi e che ognuno di tali costi sia memorizzato in $D_{v,s,i-1}, \forall v, s \in V$. Per calcolare il cammino di costo tra x e y con esattamente i archi si applica, fissato x, il passo di rilassamento: si controlla se il costo attuale D_{xye} sia maggiore di $D_{a,y,e-1} + w(x, a), \forall a, y \in V$. Ci si chiede pertanto: fissato un vertice x e per ogni coppia di vertici a e y, il costo del cammino da x a y composto dall'arco (x, a) e dal cammino di costo minimo dal vertice a al vertice j (contenente $i - 1$ archi) è minore del cammino da x a y di costo minimo attuale? Se sì, il costo attuale $D_{x,y,i}$ viene aggiornato. Il passo di rilassamento viene eseguito solo se l'arco non è un self-loop e se l'arco (v, a) non ha costo infinito (ovvero solo se $(v, a) \in E$).
I casi base si hanno per $e = 0$ ed $e = 1$, descritti dalle righe 1 e 2 dell'algoritmo 2.

3.3 Ottimizzazione

Prima dell'ottimizzazione è possibile fare alcune considerazioni. L'algoritmo 2 prende in input: *Matrice di adiacenza del grafo - Nodo di partenza - Nodo di arrivo - Numero massimo di archi.* Detto $n = |V|$, l'algoritmo inizia allocando una matrice tridimensionale di dimensione n^2q, dove q può anche essere $n-1$ nel caso peggiore, inizializzando i primi due casi base $e = 0, 1$ rispettivamente: D_{xy0} avrà la diagonale pari a 0 e D_{xy1} sarà uguale ai valori della matrice di adiacenza.
Dopodiché, l'algoritmo (2), comincia il calcolo dal secondo livello in poi, in cui:
Il primo **for** serve a scorrere i livelli, il secondo **for** serve a scorrere le righe, il terzo **for** serve a scorrere le colonne, il quarto **for** serve ad effettuare i calcoli secondo la tecnica di rilassamento descritta precedentemente.

Le osservazioni da effettuare sono le seguenti:
1. Il livello $e = 0$ non verrà mai utilizzato,
2. Per calcolare il livello corrente, basta utilizzare soltanto quello precedente,
3. i e j non possono essere uguali nell'**if** statement.

Le osservazioni 1 e 2 permettono di dedurre il seguente ragionamento: avendo già inizializzato il livello $e = 1$, per calcolare il percorso minimo con al più 2 archi bisognerà calcolare il livello D_{xy2} facendo uso solo del livello D_{xy1}, dopodiché per calcolare il percorso minimo con al più 3 archi bisognerà calcolare D_{xy3} facendo uso solo del livello D_{xy2} già precalcolato e così via. Questo significa che ogni livello D_{xye} è esattamente il percorso minimo con e archi; inoltre, siccome il calcolo del livello corrente richiede l'uso soltanto di quello precedente, allora lo spazio si può ridurre a due matrici di dimensione $n \times n$ invece di una matrice tridimensionale.
Pertanto è possibile usare due matrici A e B scambiandone l'uso a ogni iterazione.
In particolare si mantiene una variabile x, inizialmente pari a 1, che a ogni iterazione del ciclo **for** più esterno viene moltiplicata per -1. Si userà la matrice A per il computo se $x = 1$, altrimenti si userà B. A ogni calcolo del minimo percorso con esattamente e archi si memorizza quest'ultimo in un ulteriore array supporto, in posizione e. L'osservazione 3 invece permette di inizializzare $A_{i0} = +\infty, B_{i0} = +\infty$ per $1 \leq i \leq n$, senza intaccare il calcolo del minimo, perchè già scartate nell'**if** statement.
L'algoritmo proposto diventa quindi il seguente:

Algorithm 3 - minStopovers - STV

Require: grafo diretto e pesato G come matrice di adiacenza, $k > 0$
siano A, B due matrici $n \times n$, inizializza A tale che $A_{ij} = w(i, j)$
sia *queris* l'array dei cammini minimi, altrimenti
for $e = 2, \dots k$, $x \leftarrow -x$ **do**
    for each $i \in V$ **do**
        for each $j \in V$ **do**
            $A_{ij} \leftarrow +\infty$
            $B_{ij} \leftarrow +\infty$
            for each $a \in V$ **do**
                if $x = 1$ **then** ▷ Esegui solo se $a \neq i \neq j$ e $w_{i,a} \neq +\infty$
                    if $B_{ij} > A_{aj} + w_{i,a}$ **then**
                        $B_{ij} = A_{aj} + w_{i,a}$
                        $queries[e] = B_{0,n-1}$
                else
                    if $A_{ij} > B_{aj} + w_{i,a}$ **then**
                        $A_{ij} = B_{aj} + w_{i,a}$
                        $queries[e] = A_{0,n-1}$
return $\min_{1 \leq e \leq k} queries$

3.4 Complessità

L'algoritmo occupa $O(n^2)$ spazio e $O(n^3k)$ tempo, detta k la query. Si può eseguire l'algoritmo una sola volta per la massima query in input così da avere un accesso diretto, attraverso il vettore *queries*, allo shortest path per le altre query. Si suppone che nell'input le query siano date in ordine decrescente così, da avere accesso diretto alla query di valore massimo. Se così non fosse, basterebbe prendere il minimo della sequenza di query, problema che si risolve in tempo lineare.

4 Implementazione in C++

Il primo passo per l'implementazione è la costruzione di un parser, il cui ruolo è costruire i grafi dei voli a partire dal file di input. A ogni scenario descritto è associato un grafo dei voli e una sequenza di query. Il parser costruisce tali grafi, rappresentati come liste di adiacenza, con la proprietà che il vertice numerato 0 corrisponda a Palermo e quello numerato $n-1$ corrisponda a New York, con $n = |V|$. Il parser descritto corrisponde ai file della cartella "parse" del progetto.
Dopo tale costruzione iniziale, vi è una parte di pre-processing che sfrutta la BFS. Difatti potrebbero esserci query non valide, ossia minori della distanza minima tra Palermo e New York oppure maggiori della distanza massima tra le due città. La distanza minima, intesa come numero di città tra Palermo e New York (estremi esclusi), si calcola facilmente con la BFS. Il metodo $shortestDistanceToNy()$ di $PathsManager$ svolge tale compito.

Per la distanza massima si osserva che, se vi sono N città, non vi possono essere più di $N-2$ scali tra Palermo e New York. Le query invalide non vengono considerate nel computo, difatti per ognuna di esse si stampa che non vi sono voli. Nonostante gli algoritmi 2 e 3 richiedano che il grafo venga rappresentato come matrice di adiacenza, è conveniente mantenere anche una rappresentazione del grafo come liste di adiacenza a causa della parte di pre-processing descritta. Difatti la BFS richiederebbe tempo quadratico usando le matrici di adiacenza. La classe FlightsGraph è la rappresentazione del grafo dei come liste di adiacenza, mentre FlightsMatrix è la rappresentazione tramite matrice di adiacenza. Concluso il pre-processing, attraverso il metodo toMatrix() di FlightsGraph si passa alla seconda rappresentazione.

In seguito, tra le query valide si prende quella di valore massimo, andando poi ad eseguire l'algoritmo 3 su tale query. L'algoritmo è incapsulato nel metodo $shortestPathWithinK()$ di $PathsManager$, che restituisce un vector sp di double tale che $sp[i]$ è lo shortest path da Palermo a New York con esattamente $i-1$ vertici.
Pertanto per avere il cammino di costo totale minimo con al più $i-1$ vertici basta effettuare un operazione di minimo cumulativo sull'array sp, che costa tempo lineare. Ovvero, il costo totale minimo dei voli con al più i scali è dato dal minimo del sottoarray $sp[0...i+1]$.
Infine, per ogni query, si dà il risultato corrispondente nella forma descritta dalla traccia.
Le operazioni descritte si iterano per ogni scenario.