Progetto Ricerca Operativa

Implementazione algoritmi per generare i k migliori shortest path in Python

Simone Lugaresi
0000970392

1 luglio 2023

Indice

0.1 Dijkstra

L'algoritmo di Dijkstra, tramite una lista di coppie vertice-costo, è in grado di determinare il cammino da un vertice s ad un altro vertice d di costo minore. Modificando accuratamente il codice riusciamo a determinare i primi k Shortest Path.
In *figura 1* possiamo trovare lo pseudocodice sul quale mi sono basato per crare l'algoritmo in python.

Algorithm 10 Determinazione delle K migliori soluzioni per SPP

function K-DIJKSTRA
    Input: $G(V, A)$,c, s, d, K
    Output: *K shortest paths da s a d*
    $k_i = 0 \; \forall \; i \in V$;
    Heap.Empty();
    Heap.Push($s, 0, -1, -1$);
    $i = s$;
    while $k_d \leq K$ **do**
        (i, v, h, k')=Heap.Pop();
        $k_i = k_i + 1$;
        $l(i, k_i) = v$;
        $Pred(i, k_i) = (h, k')$;
        for $j : (i, j) \in A$ **do**
            $l = l(i, k_i) + c_{ij}$;
            Heap.Push(j, l, i, k_i);
        end for
    end while
    costruisci i K shortest paths usando $Pred$;
end function

Figura 1: Pseudocodice Dijkstra

La funzione *Heap.Empty()* crea una lista vuota, la funzione *Heap.Push()* in questo caso inserisce nalla lista una quartina di valori, che inizialmente sono: il nodo sorgente, il costo, h e k' inizializzati a -1, mentre la funzione *Heap.Pop()* restituisce la quartina aggiunta. Per il funzionamento dell'algoritmo marchiamo come importati le seguenti variabili:

- *Ki* un vettore di dimensioni n(numero di nodi del grafo) inizializzato a 0 che punta alla soluzione generata per ogni vertice, viene utilizzato infatti per il ciclo più esterno per assicurarsi che il vertice destinazione sia raggiunto da almeno K soluzioni.

- *L*(elle) una matrice che corrisponde alle label di Dijkstra, di dimensioni n x K, mantiene in memoria il costo per arrivare al *Iesimo* vertice nella soluzione K.

- *Pred* è una matrice di coppie (anche questa di dimensioni n x K) che serve per ricostruire i percorsi, dove nella posizione *(i,k)* corrisponde *i1* ovvero il vertice che raggiunge *i* usando la soluzione *k1*.

0.1.1 Dettaglio

Nell'implementazione dell'algoritmo ho avuto particolari difficoltà nella corretta inizializzazione delle varibili in particolare delle dimensioni della matrice.
Riscrivendo il codice ho notato che a riga 10, sempre dello pseudocodice in *figura 1*, viene aumentato di uno il valore che indica in quante soluzioni il vertice *i* viene utilizzato; questa assegnazione viene fatta nello pseudocodice senza controlli, ma così facendo ho riscontrato quasi sempre errori di *Index Out Of Bounds*, ovvero dopo l'incremento senza controllo rischiavo di avere un valore piu grande di K e quindi nella riga successiva quando cercavo di accedere alla matrice venivano passati valori fuori dai range, il tutto causato dal fatto che prima di arrivare al vertice di destinazione avevo già visitato il vertice *i* piu volte, che è dipendete da Heap.Push(), che inserisce in lista ogni quartina *(j, l, i, ki)*, senza verificare se j è già presente.

0.2 Ricorsione Backward

L'algoritmo backward per k-best SPP permette tramite la ricorsione di determinare i k cammini minimi che vanno dal vertice *s* al vertice *d*.
Questo metodo può essere molto utile in quanto permette di non visitare tutti i nodi, migliorando di conseguenze le prestazioni, ma allo stesso tempo viene limitato dall'impossibilità di muoversi in grafi ciclici, quindi ha meno opportunità di essere utilizzato.
In *figura 2* è riportato lo pseudocodice sul quale mi sono basato per creare l'algoritmo in python.

Algorithm 15 Ricorsione Backward per k-best SPP

```
function KSPP-Back(j, K)
    if j = s then
        if k = 1 then
            return 0;
        else
            return +∞;
        end if
    end if
    if k ≤ k_j then
        return z_SPP(j, k);
    end if
    l = +∞, i' = −1;
    for (i, j) ∈ A do
        l' =SPP-kBack(i, k_ij) + c_ij;
        if l' < l then
            l = l';
            i' = i;
        end if
    end for
    Pred(j, k) = (i', k_i'j);
    k_i'j = k_i'j + 1;
    k_j = k;
    z_SPP(j, k) = l;
    return z_SPP(j, k);
end function
```

Figura 2: Pseudocodice Backward

- *k e K(input)* sono la stessa variabile che ha valore intero, indica il *k-esimo* percorso meno costoso.
- *kj* è una variabile di tipo array, al suo interno viene memorizzato il *k-esimo* miglior soluzione per l'indice j
- *A* è la lista di archi.
- *kij* matrice di interi di dimesione n x n.
- *Zspp* contiene il valore di *backup* del valore del *k-esimo* shortest path che da *source* arriva al vertice *j*.
- *Pred* è una matrice di coppie (di dimensioni n x K), che come in dijkstra serve per ricostruire i k percorsi.

Qui di seguito (*figura 3*) è riportato lo pseudocodice che serve per richiamare le prime volte l'algoritmo ricorsivo.

Algorithm 16

1: **function** SOLVE-KSPP-BACK
2:     **Input**: $G(V,A)$,$\mathbf{c}$,s,d,K
3:     **Output**: *K shortest paths da s a d*
4:     *Inizializzazione strutture dati*;
5:     **for** $k' = 1$ *to* K **do**
6:         $z_{SPP}(k')$ =SPPB-kBest(d, k');
7:     **end for**
8:     *costruisci i K shortest paths da s a d usando Pred*;
9: **end function**

Figura 3: Pseudocodice funzione richiamante Backward

N.B.

Sono presenti alcuni typhos nelle due figure:

- riga 1 *figura 2*: nome della funzione sbagliato → SPP-KBack
- riga 6 *figura 3*: nome della funzione sbagliato → SPP-KBack
- riga 6 *figura 3*: Zspp(k') → Zspp(d,k')

0.2.1 Dettaglio

Nell'implementazione dell'algoritmo ho riscontrato due punti particolarmente difficili:

- 1. La comprensione, l'inizializzazione e il corretto uso delle variabili, sopratutto dato dalla similitudine dei nomi di quest'ultime e i typhos citati in precedenza.
- 2. Nella riga 13 potrebbe sembrare che il for si ripeta per ogni arco appartenente al set di archi, ma il significato corretto è che si ripete per ogni arco in cui la variabile j è uguale alla variaible j passata in input, di conseguenza è necessario aggiungere un controllo.

0.3 Risultati computazionali

Per ricavare i risultati computazionali ho create due funzioni:

- *Generate random graph* nel quale passo in input il numero di vertici, il minimo di archi che ci devono essere per ogni nodo e la probabilità che un nodo abbia un arco con un altro in output mi restituisce un grafo casuale con questi requisiti (utilizzato per l'algoritmo di Djkstra).
- *Generate random graph acyclic* nel quale passo in input il numero di vertici, il minimo di archi che ci devono essere per ogni nodo e la probabilità che un nodo abbia un arco con un altro in output mi restituisce un grafo casuale aciclico con questi requisiti((i,j) generati solo se j è maggiore di i) (utilizzato per l'algoritmo di Backward) .
- *Generate random costs* che preso in inputo il grafo creato in precedenza, e il massimo costo che può avere un arco mi restituisce in output una lista dei costi.

Infine cambiando i valori iniziali cioè il numero di nodi(n), numero di archi minimo per nodo(a), il costo massimo per ognuno di essi(c), e (K) il numero di soluzioni da trovare, ho creato grafi diversi per ogni esecuzione e sorteggiando un vertice di partenza e uno di arrivo ho ottenuto i seguenti risultati:

0.3.1 Risultati Dijkstra

- *istanza 1* - n = 500, a = 80, c = 20, K = 50
- *istanza 2* - n = 1000, a = 100, c = 20, K = 100
- *istanza 3* - n = 500, a = 80, c = 100, K = 50

	Esecuzione 1	Esecuzione 2	Esecuzione 3	Esecuzione 4	Esecuzione 5	Media
Istanza 1	0,445229	1,835852	1,690502	1,421488	0,567620	1,192138
Istanza 2	16.117221	2.490432	1.705984	4.003449	30.726433	11.008703
Istanza 3	0.523600	3.027632	0.951037	0.058297	1.287823	1.169677

Tabella 1: Risultati con l'algoritmo di Dijkstra

0.3.2 Risultati Backwawrd

- *istanza 1* - n = 100, a = 20, c = 20, K = 20
- *istanza 2* - n = 500, a = 40, c = 20, K = 50
- *istanza 3* - n = 1000, a = 50, c = 100, K = 100

	Esecuzione 1	Esecuzione 2	Esecuzione 3	Esecuzione 4	Esecuzione 5	Media
Istanza 1	0.006980	0.028894	0.018975	0.051861	0.052890	0.031920
Istanza 2	1.222704	2.001205	2.888271	3.398471	4.333117	2.768754
Istanza 3	8.816641	8.155504	19.222883	39.050607	30.159517	21.081030

Tabella 2: Risultati con l'algoritmo ricorsivo

Per ricavare i risultati con l'algoritmo *Backward* mi sono trovato più in difficoltà perchè generando in modo casuale ogni volta il grafo, oltre che essere aciclico (ovvero senza cicli) doveva anche essere *"fattibile"* per l'algoritmo; di conseguenza i dati riportati potrebbero essere "*falsificati*" dal fatto che il grafo non aveva un tutti i k percorsi o nemmeno uno percorribili, portando ad una preventiva fine dell'esecuzione con una conseguente diminuzione del tempo di esecuzione.

0.4 Conclusione

Seppure il lavoro svolto sia funzionale per renderlo più efficente si potrebbe pensare di migliorare l'implementazione, magari utilizzando varie librerie come *numpy* per l'utilizzo di array e matrici, in sostituzione alle liste di liste di default di python che lavorano più lentamente, oppure un'eventuale parallelizzazione del codice.