Relazione di laboratorio

Corso di Algoritmi Avanzati
Laurea Magistrale in Informatica
A.A. 2019-2020

Magarotto Francesco - 1236594
Muraro Enrico - 1238899
Piva Giulio - 1242455

1 Introduzione

L'esercizio di laboratorio consiste nel implementare, valutare e confrontare tre algoritmi per il problema del Minimum Spanning Tree. Gli algoritmi sono:

- Prim implementato con Heap
- Kruskal in versione naive
- Kruskal implementato con Union-Find

Il linguaggio di programmazione scelto dal nostro gruppo è Java.

1.1 Esecuzione del programma

Gli algoritmi sono stati sviluppati come progetto Maven. All'interno della cartella é presente la versione portable di Maven, pertanto non è necessario averlo installato. É richiesto almeno il JDK 11 installato nel sistema. Per eseguire i tre algoritmi utilizzare i seguenti comandi:
Linux:

```
./mvnw install
./mvnw exec:java
```

Windows:

```
mvnw.cmd install
mvnw.cmd exec:java
```

È richiesto un tempo di esecuzione di circa 1.30h per l'esecuzione di tutti gli algoritmi su tutti i file. Ulteriori esecuzioni su hardware hanno richiesto circa 2h. L'esecuzione del main genera automaticamente dei file csv nella directory del progetto contenenti i tempi registrati.

1.2 Strutture dati utilizzate

Per rappresentare il grafo abbiamo creato la classe Graph che contiene sia una lista non ordinata di tutti i lati, che una lista di adiacenza per ogni nodo. I lati sono rappresentati dalla classe Edge che contiene tre valori interi, il nodo di partenza, il nodo di arrivo e il peso del lato.

La lista di adiacenza è implementata attraverso una HashMap dove la chiave è il nodo, e il valore è una lista di tutti i lati adiacenti al nodo stesso implementata con una LinkedList. Abbiamo scelto una HashMap al posto di un array dove ogni indice corrisponde al nodo con quel valore, per evitare di allocare un array sparso quando il valore dei nodi non rispecchia effettivamente il numero di nodi nel grafo, pur mantenendo un tempo di accesso alla lista costante.

La classe Heap è una nostra implementazione di MinHeap. L'albero binario è rappresentato da un array di interi, i valori all'interno dell'array corrispondono ai nodi presenti nel grafo. Il confronto tra i nodi per determinare il più piccolo è effettuato tramite un Comparator passato alla creazione dello Heap, questo per avere un'implementazione di Heap indipendente dal modo in cui viene utilizzato da uno specifico algoritmo.

1.3 Lettura di un grafo da file

Per caricare un grafo in memoria, abbiamo implementato una classe GraphReader, che si occupa della lettura del file tramite la libreria *nio* di Java. Dato che gli algoritmi implementati sono basati sull'assunzione che non ci siano doppi archi tra nodi, abbiamo tenuto quello di peso minore. I self-loop, invece, vengono mantenuti, anche se durante l'esecuzione degli algoritmi verranno effettivamente ignorati.

1.4 Implementazione di Prim

Prim necessita di due campi aggiuntivi "key" e "parent" per ogni nodo, key contiene l'attuale peso più piccolo per raggiungere il nodo, e parent contiene il nodo da cui si arriva. Questi campi sono stati implementati con due HashMap come per la lista di adiacenza.

L'Heap viene inizializzato con tutti i nodi del grafo, e nel costruttore viene passata un'istanza di NodeComparator che utilizza il campo key per confrontare i nodi.

Il Set Q contiene tutti i nodi che non sono ancora nel MST, ed è utile per vedere il nodo che stiamo valutando non è nel MST in tempo costante al posto di cercarlo nello Heap in tempo lineare.

Il costo totale del MST viene infine calcolato semplicemente scorrendo tutti i nodi e sommando i loro valori di key.

1.5 Implementazione di Kruskal Naive

L'implementazione di Kruskal "Naive" necessita che la lista degli archi sia ordinata in maniera crescente in base al peso, per effettuare tale operazione si è fatto uso degli stream introdotti in Java 8. Infatti, l'ordinamento dalla libreria standard è implementato attraverso l'algoritmo Timsort che ha complessità nel caso peggiore $O(n * log(n))$, mentre nel caso migliore $O(n)$ e quindi, in alcuni casi, potrebbe essere più efficiente di Merge Sort. Inoltre, viene fatto uso di una lista per contenere gli archi dell'MST e di un grafo temporaneo utilizzato per verificare che la condizione di aciclicità sia soddisfatta. Quest'ultima è il fulcro dell'algoritmo e viene realizzata implementando una ricerca BFS iterativa, infatti per aggiungere un arco all'MST bisogna verificare che l'aggiunta di questo arco non crei un ciclo. Questa verifica viene svolta attraverso BFS, infatti se esiste un cammino tra i due nodi dell'arco che sto aggiungendo questo creerebbe un ciclo.

1.5.1 `findPath`

`findPath` è il metodo che implementa iterativamente BFS e ritorna un valore booleano se e solo se esiste un cammino dal nodo di partenza a quello di destinazione. L'idea principale si basa sullo scorrere la lista di adiacenza del nodo di partenza tenendo traccia dei vertici già visitati. Se dal nodo *s* (start) arrivo ad *e* (end) allora aggiungendo l'arco (s, e) formerei un ciclo.

1.6 Implementazione di Kruskal con Union-Find

L'algoritmo funziona esattamente come la versione "Naive", ma utilizza un modo più efficiente per verificare che l'aggiunta di un arco non produca un ciclo nel MST. Questo metodo consiste nell'utilizzare una struttura dati per insiemi disgiunti e dei metodi Union e Find. Un sottoinsieme disgiunto è implementata dalla classe *subset*, che ha come campi parent e size. Ogni sottoinsieme contiene i vertici di un albero in una foresta. La funzione *find* restituisce la radice dell'albero al quale appartiene il nodo, risalendo di parent in parent. Se l'esecuzione di find su entrambi i nodi restituisce la stessa radice, significa che i due nodi appartengono allo stesso albero, e di conseguenza quell'arco non va aggiunto. Se invece non restituiscono la stessa radice, si procede ad unire i due sottoinsiemi tramite la funzione Union e ad aggiungere il lato corrente al MST. Il campo size serve per mantenere un sottoinsieme come albero bilanciato quando si fa l'unione dei due sottoinsiemi, riducendo la complessità di find da $O(n)$ a $O(log(n))$. In totale, la complessità dell'algoritmo diventa $O(mlog(n))$.

2 Risultati ottenuti

Istanza	Held-Karp			Euristica Nearest			2 Approssimazione		
	Soluzione ottima	Tempo (ms)	Errore (%)	Soluzione ottima	Tempo (ms)	Errore (%)	Soluzione ottima	Tempo (ms)	Errore (%)
ulysses22.tsp	8194,0	300,002	17%	10586	0,0	51%	8308	2	18%
gr229.tsp	182845,0	300,001	36%	162430	3	21%	179335	14	33%
kroD100.tsp	148541,0	300,001	598%	26947	0,0	27%	28599	2	34%
kroA100.tsp	172858,0	300,001	712%	27807	0,0	31%	30516	1	43%
berlin52.tsp	18866,0	300,032	150%	8980	0,0	19%	10402	1	38%
ulysses16.tsp	6859,0	15,676	0,0%	9988	1	46%	7788	0,0	14%
eil51.tsp	1059,0	300,001	149%	511	0,0	20%	605	0,0	42%
pcb442.tsp	229601,0	300,001	352%	61979	6	22%	73926	12	46%
ch150.tsp	48239,0	299,998	639%	8191	0,0	25%	9126	1	40%
dsj1000.tsp	5,52276003E8	300,004	2860%	24630960	4	32%	25526005	27	37%
d493.tsp	112899,0	300,002	223%	41665	2	19%	45623	0,01	30%
gr202.tsp	72655,0	300,001	81%	49336	0,0	23%	52615	2	31%
burma14.tsp	3323,0	1,02	0,0%	4048	0,0	22%	4003	0,0	20%