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 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.

1.2 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.

1.3 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.

1.4 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.

1.4.1 `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.

1.5 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

Nodi	Prim	Kruskal UF	Kruskal	MST
10	0	0	0	29316
10	0	0	0	2126
10	0	0	0	-44765
10	0	0	0	20360
20	0	0	0	-32021
20	0	0	0	18596
20	0	0	0	-42560
20	0	0	0	-37205
40	0	0	0	-122078
40	0	0	0	-37021
40	0	0	0	-79570
40	0	0	0	-79741
80	0	0	0	-139926
80	0	0	0	-211345
80	0	0	0	-110571
80	0	0	0	-233320
100	0	0	0	-141960
100	0	0	0	-271743
100	0	0	0	-288906
100	0	0	0	-232178
200	0	0	16	-510185
200	0	0	0	-515136
200	0	0	0	-444357
200	0	0	0	-393278
400	0	0	0	-1122919
400	0	0	31	-788168
400	0	0	0	-895704
400	0	0	0	-733645
800	0	16	31	-1541291
800	0	0	32	-1578294
800	16	0	31	-1675534
800	0	0	15	-1652119
1000	15	0	32	-2091110
1000	0	0	32	-1934208
1000	0	0	31	-2229428
1000	0	0	31	-2359192

Nodi	Prim	Kruskal UF	Kruskal	MST
2000	0	0	109	-4811598
2000	0	16	93	-4739387
2000	0	0	109	-4717250
2000	0	0	125	-4537267
4000	0	16	423	-8722212
4000	0	0	437	-9314968
4000	15	0	412	-9845767
4000	16	0	439	-8681447
8000	16	15	1722	-17844628
8000	16	0	1713	-18800966
8000	15	15	1731	-18741474
8000	16	0	1753	-18190442
10000	16	0	2637	-22086729
10000	31	16	2601	-22338561
10000	15	0	2563	-22581384
10000	16	0	2638	-22606313
20000	63	16	12976	-45978687
20000	65	0	13002	-45195405
20000	62	16	13009	-47854708
20000	63	0	12427	-46420311
40000	189	16	81522	-92003321
40000	203	15	81525	-94397064
40000	203	19	84400	-88783643
40000	189	15	81242	-93017025
80000	703	46	459713	-186834082
80000	694	44	549392	-185997521
80000	697	47	545050	-182065015
80000	687	31	526711	-180803872
100000	1045	125	955113	-230698391
100000	1110	63	925063	-230168572
100000	1137	59	872704	-231393935
100000	1077	47	869876	-231011693

Tabella 1: Tabella dei risultati. Per ogni algoritmo, i valori si riferiscono al tempo di esecuzione, espressi in ms, di quell'algoritmo con un grafo avente n nodi.

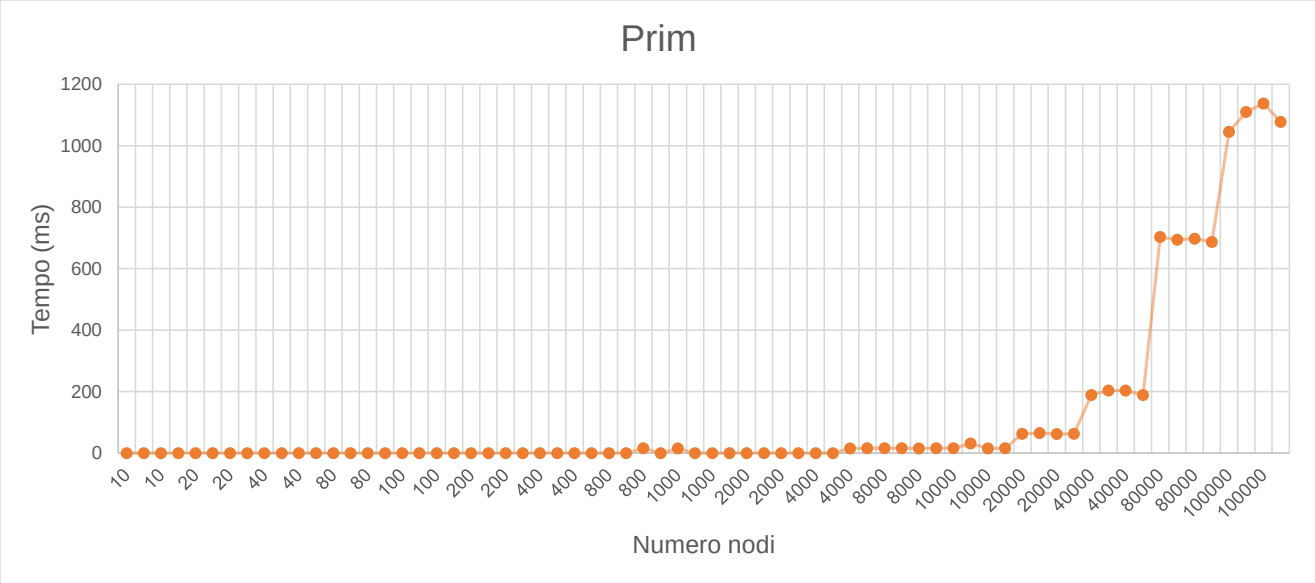

Figura 1: Grafico andamento algoritmo di Prim

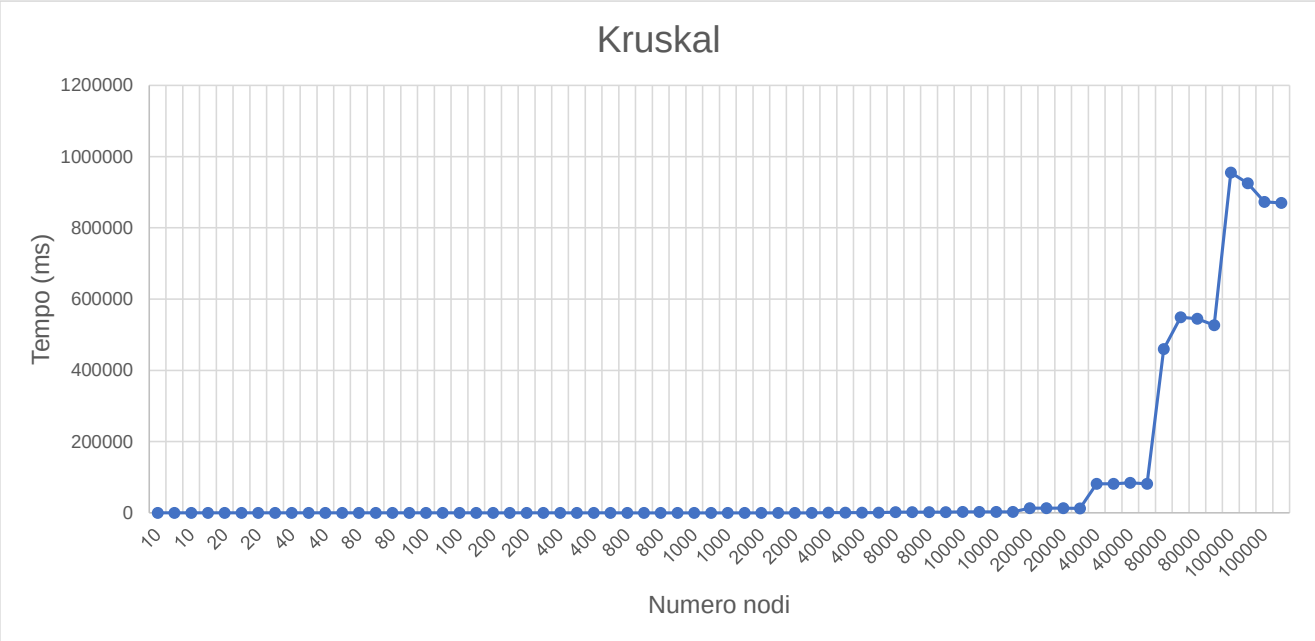

Figura 2: Grafico andamento algoritmo di Kruskal

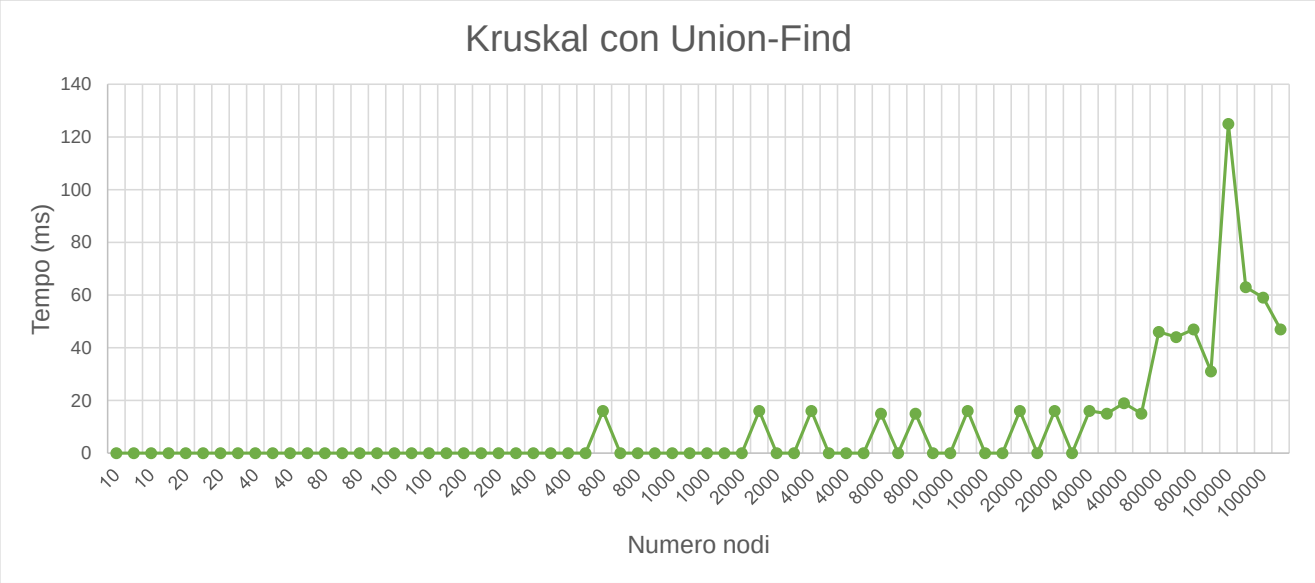

Figura 3: Grafico andamento algoritmo di Kruskal con Union-Find

3 Confronto

Sicuramente l'algoritmo con costo computazionale più basso è Kruskal con Union-Find. Confrontiamo quindi l'esecuzione di Kruskal con Union-Find con gli altri due algoritmi partendo da Prim.

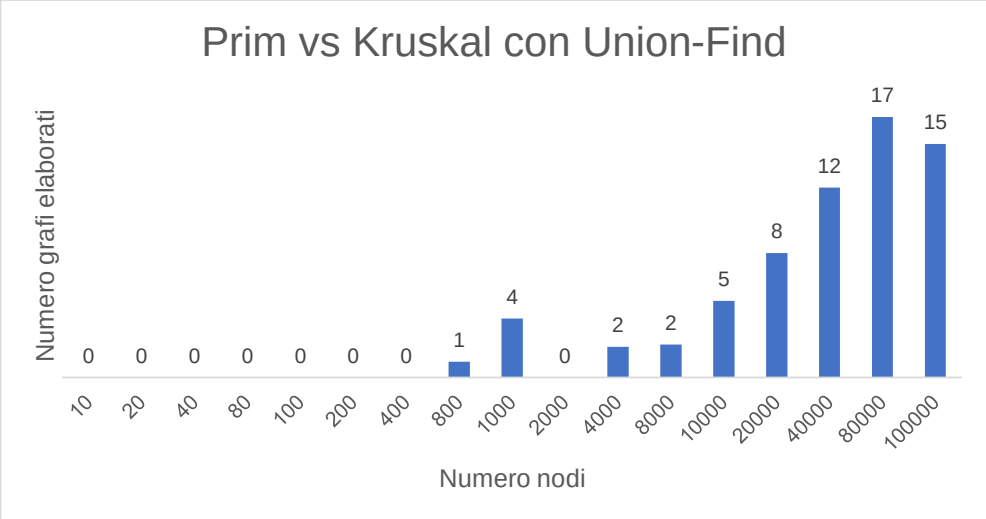

Figura 4: Numero di grafi di dimensione n che Kruskal con Union-Find è in grado di elaborare nello stesso tempo medio impiegato da Prim per elaborare **un** grafo della medesima dimensione. Come possiamo vedere Kruskal con Union-Find è molto più efficiente in grafi di grandi dimensioni. Ad esempio per un grafo di 10^5 nodi, in media, Kruskal con Union-Find è 15 volte più veloce di Prim.

Confrontiamo ora l'algoritmo di Kruskal con Union-Find con la sua versione naive:

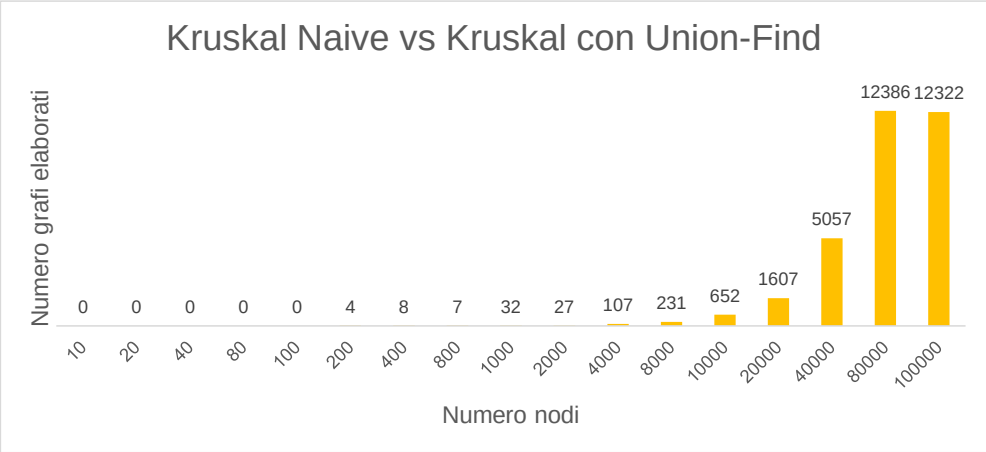

Figura 5: Numero di grafi di dimensione n che Kruskal con Union-Find è in grado di elaborare nello stesso tempo medio impiegato da Kruskal naive per elaborare **un** grafo della medesima dimensione. Come possiamo vedere Kruskal è molto più efficiente in grafi di grandi dimensioni. Ad esempio per un grafo di 10^5 nodi, in media, Kruskal con Union-Find è 12322 volte più veloce di Kruskal naive.

Da questi due grafici e nella tabella 1 possiamo affermare che Kruskal con Union-Find è l'algoritmo **più** efficiente per il calcolo di un MST, segue l'algoritmo di Prim, e infine Kruskal "Naive" che ha il costo computazionale più alto.

4 Conclusione

I risultati ottenuti rispecchiano l'andamento che ci aspettavamo conoscendo la complessità degli algoritmi, infatti Kruskal "Naive" di complessità $O(mn)$ ha ottenuto risultati peggiori di qualche ordine di grandezza rispetto agli altri due algoritmi, i quali hanno entrambi complessità $O(m \log n)$.

La differenza tra Kruskal con Union-Find e Prim è dovuta probabilmente dalla nostra implementazione degli algoritmi e delle strutture dati che abbiamo utilizzato, ad esempio esistono implementazioni migliori di Prim che utilizzano Heap di Fibonacci per

abbassare la complessità a $O(m + n \log n)$ e che avrebbero portato a tempi di esecuzione più bassi per i grafi di dimensione maggiore.

In termini di efficienza Kruskal con Union-Find ha ottenuto i tempi di esecuzione più bassi sia per grafi di piccola dimensione sia per i grafi con più di 80000 nodi, ed è quindi l'algoritmo migliore nella nostra implementazione.