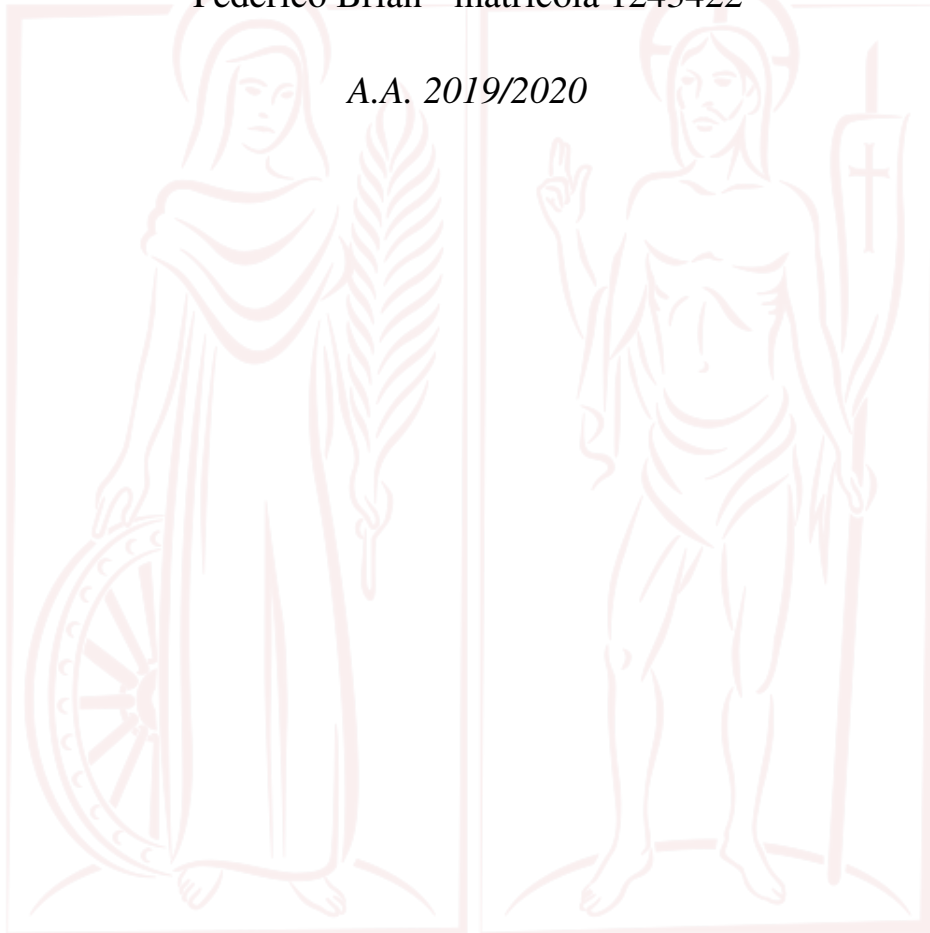


# Relazione di Algoritmi Avanzati

Nicola Carlesso - matricola 1237782

Federico Brian - matricola 1243422

*A.A. 2019/2020*



## Indice

<b>1</b>	<b>Introduzione</b>	<b>1</b>
1.1	Scelta del linguaggio di programmazione . . . . .	1
1.2	Scelte implementative . . . . .	1
<b>2</b>	<b>Algoritmi</b>	<b>2</b>
2.1	Prim . . . . .	2
2.2	Naive Kruskal . . . . .	6
2.3	Kruskal . . . . .	10
<b>3</b>	<b>Conclusioni</b>	<b>14</b>

## Elenco delle figure

1	Performance dell'algoritmo Prim. . . . .	2
2	Performance dell'algoritmo Naive Kruskal. . . . .	6
3	Performance dell'algoritmo Kruskal. . . . .	10
4	Performance dei tre algoritmi a confronto. . . . .	14

## Elenco delle tabelle

1	Risultati algoritmo di <i>Prim</i> (1 di 3) . . . . .	3
2	Risultati algoritmo di <i>Prim</i> (2 di 3) . . . . .	4
3	Risultati algoritmo di <i>Prim</i> (3 di 3) . . . . .	5
4	Risultati dell'algoritmo <i>Naive Kruskal</i> (1 di 3) . . . . .	7
5	Risultati dell'algoritmo <i>Naive Kruskal</i> (2 di 3) . . . . .	8
6	Risultati dell'algoritmo <i>Naive Kruskal</i> (3 di 3) . . . . .	9
7	Risultati algoritmo di <i>Kruskal</i> (1 di 3) . . . . .	11
8	Risultati algoritmo di <i>Kruskal</i> (2 di 3) . . . . .	12
9	Risultati algoritmo di <i>Kruskal</i> (3 di 3) . . . . .	13

# 1 Introduzione

## 1.1 Scelta del linguaggio di programmazione

Per lo svolgimento di tale assignment abbiamo scelto come linguaggio di programmazione Java 8, dato che è stato studiato da entrambi durante il percorso di laurea triennale. Inoltre, dato che Java lavora principalmente attraverso riferimenti, ha permesso lo sviluppo della struttura di grafi, vertici e lati più semplice.

## 1.2 Scelte implementative

Nell'implementazione abbiamo infatti cercato di creare meno oggetti possibile usando per lo più riferimenti. Questo ha permesso non solo un risparmio in termini di memoria, ma anche di prestazioni nel caso dell'algoritmo *NaiveKruskal*, che in una sua prima implementazione eseguiva ad iterazione del ciclo *for* una copia temporanea del grafo, più il lato preso in considerazione dall'algoritmo.

Nello specifico le varie componenti del modello presentano le seguenti caratteristiche:

- \* **Graph**: presenta una lista di nodi ed una lista di vertici ed una lista di lati. Nella costruzione del grafo non controlliamo se viene inserito un lato già presente, perciò un grafo può avere diversi pesi. Abbiamo fatto tale scelta perché la costruzione del grafo è più veloce dato che si evita un controllo di tutti i lati del grafo quando se ne aggiunge uno, riuscendo a mantenere comunque la correttezza degli algoritmi. Abbiamo implementato inoltre l'algoritmo per effettuare la DFS (Deep First Search), necessaria per gli algoritmi *Kruskal* e *Naive Kruskal*;
- \* **Node**: oltre ai campi *ID* e *Father*, sono presenti campi dati usati solo in alcuni algoritmi
  - **weight**: dato usato esclusivamente per l'algoritmo *Prim* ed indica il peso minimo del lato che collega il nodo al MST (Minimum Spanning Tree) creato fino a quel momento dall'algoritmo;
  - **visited**: dato usato solo dagli algoritmi *Kruskal* e *Naive Kruskal*;
  - **adjacentList**: dato che non contiene i nodi adiacenti al nodo selezionato, come ci si potrebbe aspettare, ma contiene la lista dei lati che hanno come estremo il nodo selezionato. Abbiamo fatto tale scelta perché in tal modo, accedendo ad un elemento di *adjacentList* abbiamo anche le informazioni dei lati, utile ad esempio per algoritmo *Prim*, e se abbiamo bisogno del nodo adiacente al nodo selezionato è possibile chiamare *Graph.opposite(edge, node)* in tempo costante.

Per utilizzare propriamente la struttura dati *Priority Queue* la classe *modo* implementa l'interfaccia *Comparator* e confronta i nodi in base al loro dato *weight*

- \* **Edge**: oltre ai riferimenti dei nodi estremi, è presente il campo *Label* per effettuare la DFS.

## 2 Algoritmi

Tale sezione descrive in breve l'implementazione e le performance degli algoritmi richiesti.

### 2.1 Prim

L'algoritmo non presenta variazioni nell'implementazione rispetto all'algoritmo mostrato a lezione, dunque possiede una complessità  $O(n \log n)$ .

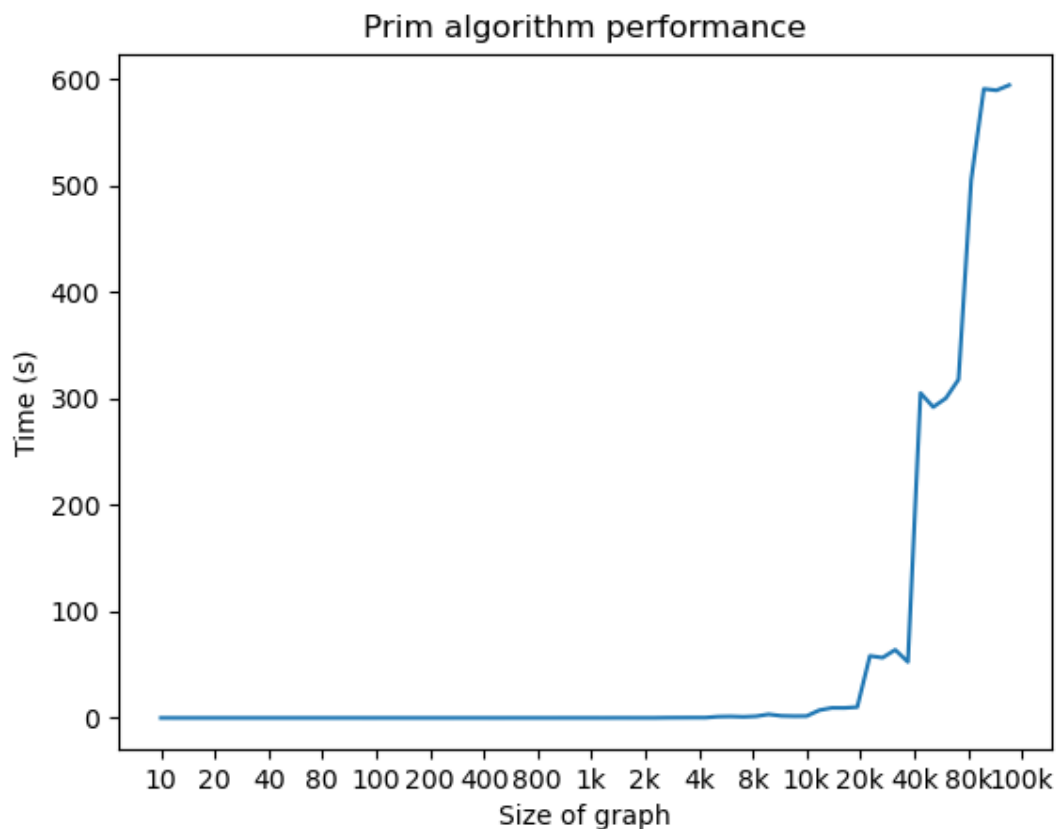


Figura 1: Performance dell'algoritmo Prim.

L'algoritmo è decisamente performante per grafi fino a 10k nodi, successivamente inizia ad essere relativamente lento per grafi da 40k nodi, impiegando 1 minuto, fino ad arrivare a grafi con 100k nodi impiegando 10 minuti.

L'algoritmo, in particolare, nel momento in cui ci si avvicina ai 200 nodi e si raddoppia la stazza del grafo, l'algoritmo richiede un tempo di risoluzione circa 6 volte superiore.

N.	Graph Size	Time (s)	MST cost
1	10	0.005691	29316
2	10	0.0002781	2126
3	10	0.0002143	-44765
4	10	0.0001572	20360
5	20	0.002932	-32021
6	20	0.0002687	18596
7	20	0.0005665	-42560
8	20	0.000397	-37205
9	40	0.0004462	-122078
10	40	0.001578	-37021
11	40	0.0012624	-79570
12	40	0.0004704	-79741
13	80	0.0022566	-139926
14	80	0.0004229	-211345
15	80	0.0004965	-110571
16	80	0.0055763	-233320
17	100	0.0003762	-141960
18	100	0.0003986	-271743
19	100	0.0053806	-288906
20	100	0.0003545	-232178
21	200	0.0008466	-510185
22	200	0.0012841	-515136
23	200	0.0007046	-444357
24	200	0.0007651	-393278
25	400	0.0081984	-1122919
26	400	0.0062876	-788168
27	400	0.0024222	-895704
28	400	0.0022616	-733645
29	800	0.0156534	-1541291
30	800	0.0170512	-1578294
31	800	0.0111666	-1675534
32	800	0.012685	-1652119

Tabella 1: Risultati algoritmo di *Prim* (1 di 3)

N.	Graph Size	Time (s)	MST cost
33	1k	0.0271609	-2091110
34	1k	0.0261404	-1934208
35	1k	0.0232982	-2229428
36	1k	0.0134004	-2359192
37	2k	0.0526213	-4811598
38	2k	0.0500484	-4739387
39	2k	0.0477902	-4717250
40	2k	0.0457845	-4537267
41	4k	0.1973502	-8722212
42	4k	0.2438554	-9314968
43	4k	0.2912372	-9845767
44	4k	0.2805648	-8681447
45	8k	1.1943197	-17844628
46	8k	1.4036635	-18800966
47	8k	1.025199	-18741474
48	8k	1.5333734	-18190442
49	10k	3.3735423	-22086729
50	10k	1.9356491	-22338561
51	10k	1.6555519	-22581384
52	10k	1.7339627	-22606313
53	20k	7.2241058	-45978687
54	20k	9.442905599	-45195405
55	20k	9.409197101	-47854708
56	20k	10.1661279	-46420311
57	40k	58.2513275	-92003321
58	40k	56.572616	-94397064
59	40k	63.9437065	-88783643
60	40k	52.5889188	-93017025
61	80k	305.1820434	-186834082
62	80k	292.0056872	-185997521
63	80k	300.2967593	-182065015
64	80k	317.9336533	-180803872

Tabella 2: Risultati algoritmo di *Prim* (2 di 3)

N.	Graph Size	Time (s)	MST cost
65	100k	505.2773482	-230698391
66	100k	590.8584786	-230168572
67	100k	589.5148767	-231393935
68	100k	594.4183702	-231011693

Tabella 3: Risultati algoritmo di *Prim* (3 di 3)

## 2.2 Naive Kruskal

Anche per questo algoritmo non abbiamo fatto variazioni rispetto all'implementazione studiata a lezione, l'algoritmo infatti risulta avere una complessità finale di  $O(mn)$ . Se si osserva con attenzione però la complessità di ogni operazione all'interno dell'algoritmo, è possibile notare che dentro il ciclo *for* è presente la funzione *Graph.hasCycle()* che controlla se nel grafo selezionato è presente un ciclo. Tale funzione ha complessità  $O(m + n)$  e farebbe dunque pensare che dunque l'algoritmo, nel totale, abbia una complessità  $O(m(m + n))$ ; siccome però tale funzione viene invocata solo per MST, sappiamo che  $m = n - 1$ , dunque la complessità dell'algoritmo infine è proprio  $O(mn)$ .

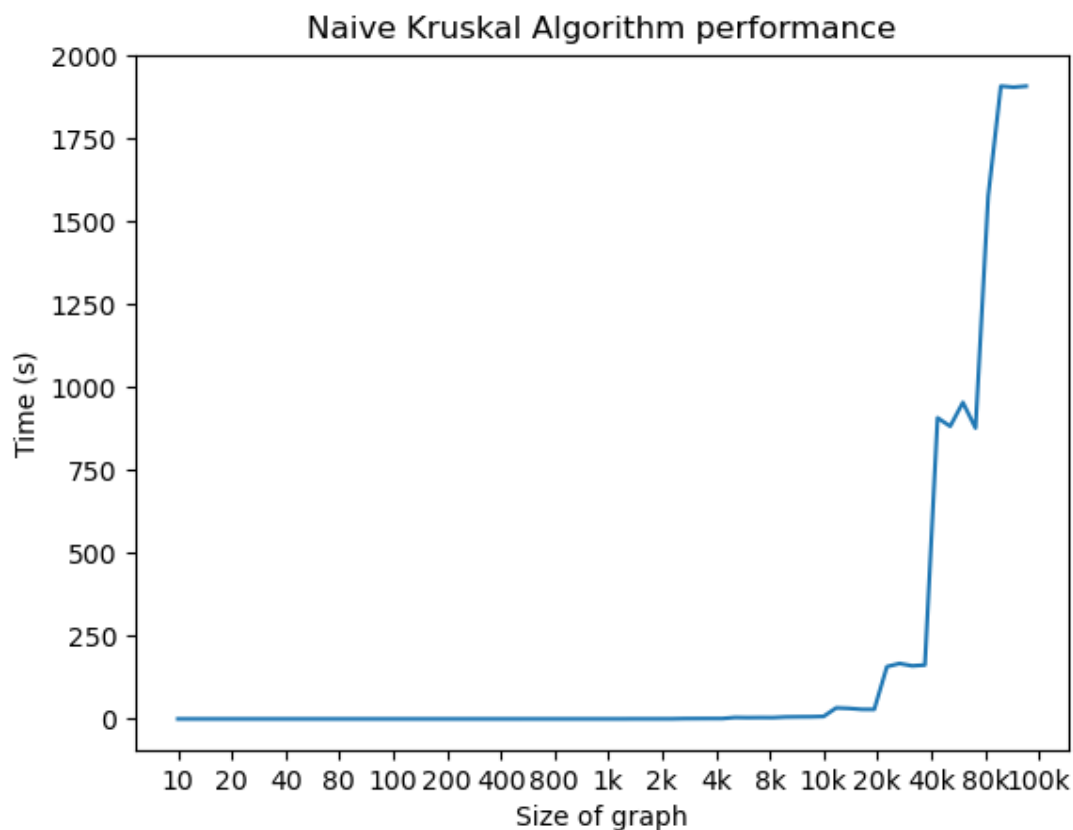


Figura 2: Performance dell'algoritmo Naive Kruskal.

*Naive Kruskal* è molto efficiente con grafi fino a 4k nodi, mentre già a 20k nodi inizia a mostrare rallentamenti, richiedendo un tempo di 30 secondi, fino ad un tempo di 30 minuti per grafi di 100k nodi. Inoltre, già a partire dai grafi di dimensione di 200 nodi, raddoppiare la grandezza del grafo richiede un aumento del tempo di risoluzione di un fattore 4 o 5.



N.	Graph Size	Time (s)	MST cost
1	10	0.0010596	29316
2	10	0.0002246	2126
3	10	0.0001755	-44765
4	10	0.0001161	20360
5	20	0.0003615	-32021
6	20	0.0002788	18596
7	20	0.0002375	-42560
8	20	0.0002041	-37205
9	40	0.0019728	-122078
10	40	0.0003499	-37021
11	40	0.0003325	-79570
12	40	0.0002773	-79741
13	80	0.0014311	-139926
14	80	0.0008544	-211345
15	80	0.0009532	-110571
16	80	0.0020361	-233320
17	100	0.0022148	-141960
18	100	0.001367	-271743
19	100	0.0011847	-288906
20	100	0.0008953	-232178
21	200	0.0022246	-510185
22	200	0.0018707	-515136
23	200	0.0019732	-444357
24	200	0.0025881	-393278
25	400	0.0069684	-1122919
26	400	0.0055755	-788168
27	400	0.0089353	-895704
28	400	0.0060748	-733645
29	800	0.0252481	-1541291
30	800	0.0258438	-1578294
31	800	0.0269498	-1675534
32	800	0.0250428	-1652119

Tabella 4: Risultati dell'algoritmo *Naive Kruskal* (1 di 3)

N.	Graph Size	Time (s)	MST cost
33	1k	0.0396759	-2091110
34	1k	0.0365509	-1934208
35	1k	0.0376809	-2229428
36	1k	0.040684	-2359192
37	2k	0.1869853	-4811598
38	2k	0.1927233	-4739387
39	2k	0.1956191	-4717250
40	2k	0.2264677	-4537267
41	4k	0.9198784	-8722212
42	4k	1.0673374	-9314968
43	4k	1.176725	-9845767
44	4k	1.177549	-8681447
45	8k	4.3709961	-17844628
46	8k	3.8505744	-18800966
47	8k	4.152138	-18741474
48	8k	3.9908528	-18190442
49	10k	5.846408	-22086729
50	10k	6.316151	-22338561
51	10k	6.526546	-22581384
52	10k	7.4906781	-22606313
53	20k	33.2681333	-45978687
54	20k	31.7596484	-45195405
55	20k	28.8519555	-47854708
56	20k	28.6587404	-46420311
57	40k	157.6931358	-92003321
58	40k	167.1818835	-94397064
59	40k	160.0550749	-88783643
60	40k	162.5589424	-93017025
61	80k	908.1871984	-186834082
62	80k	882.8654037	-185997521
63	80k	954.7231673	-182065015
64	80k	877.5672819	-180803872

Tabella 5: Risultati dell'algoritmo *Naive Kruskal* (2 di 3)

N.	Graph Size	Time (s)	MST cost
65	100k	1577.2524851	-230698391
66	100k	1909.1328731	-230168572
67	100k	1905.8766097	-231393935
68	100k	1908.9756497	-231011693

Tabella 6: Risultati dell'algoritmo *Naive Kruskal* (3 di 3)

### 2.3 Kruskal

Abbiamo implementato l'algoritmo come indicato a lezione senza variazioni alcuna.

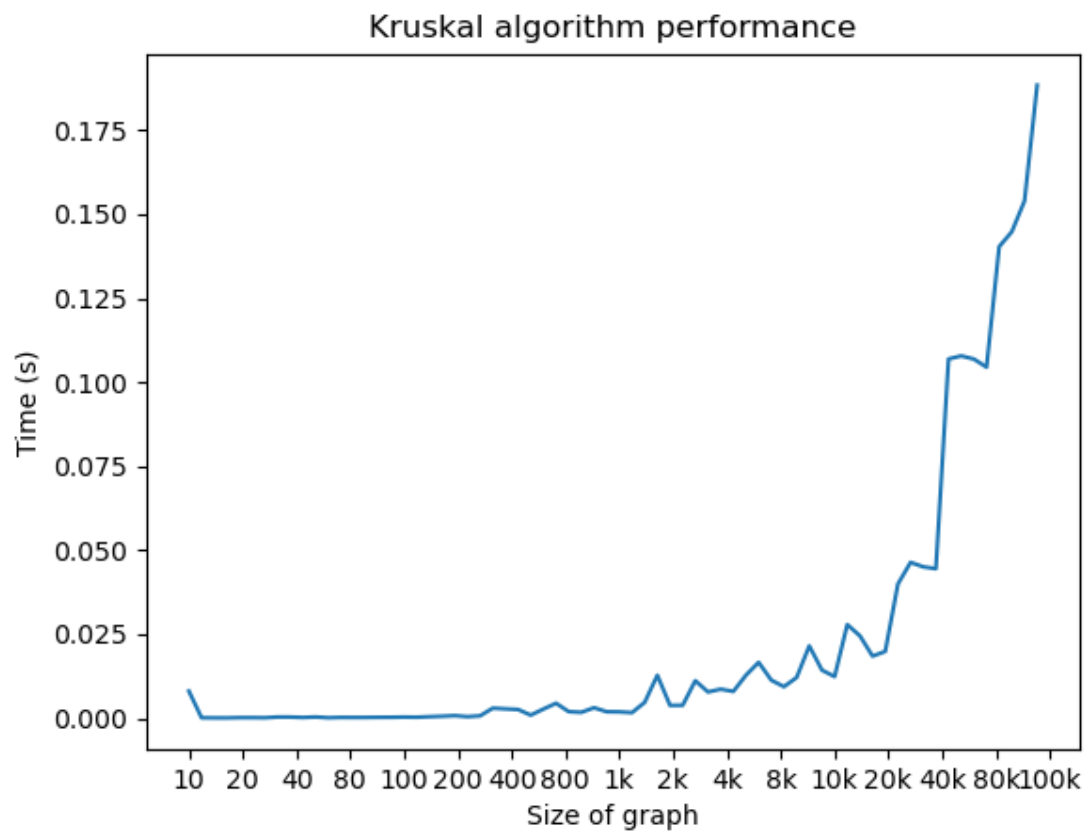


Figura 3: Performance dell'algoritmo Kruskal.

L'algoritmo è molto performante anche per grafi con 100k, impiegando infatti 0.2 secondi. Raddoppiando la dimensione del grafo, al massimo, il tempo di risoluzione del grafo raddoppia.

N.	Graph Size	Time (s)	MST cost
1	10	0.0080559	29316
2	10	0.0001189	2126
3	10	8.96e-05	-44765
4	10	7.92e-05	20360
5	20	0.0001614	-32021
6	20	0.0001641	18596
7	20	0.0001285	-42560
8	20	0.0003505	-37205
9	40	0.0003404	-122078
10	40	0.0002097	-37021
11	40	0.000361	-79570
12	40	0.0001309	-79741
13	80	0.0002288	-139926
14	80	0.0002083	-211345
15	80	0.0002255	-110571
16	80	0.0002586	-233320
17	100	0.0002707	-141960
18	100	0.0003329	-271743
19	100	0.000289	-288906
20	100	0.0004452	-232178
21	200	0.0005749	-510185
22	200	0.0007544	-515136
23	200	0.000455	-444357
24	200	0.0007035	-393278
25	400	0.0029743	-1122919
26	400	0.0027706	-788168
27	400	0.0025862	-895704
28	400	0.0009081	-733645
29	800	0.0027124	-1541291
30	800	0.0044353	-1578294
31	800	0.0019433	-1675534
32	800	0.0017496	-1652119

Tabella 7: Risultati algoritmo di *Kruskal* (1 di 3)

N.	Graph Size	Time (s)	MST cost
33	1k	0.0031046	-2091110
34	1k	0.0019217	-1934208
35	1k	0.0018778	-2229428
36	1k	0.0016544	-2359192
37	2k	0.0047124	-4811598
38	2k	0.0127673	-4739387
39	2k	0.0038127	-4717250
40	2k	0.0038368	-4537267
41	4k	0.0111621	-8722212
42	4k	0.0077927	-9314968
43	4k	0.0086298	-9845767
44	4k	0.0079643	-8681447
45	8k	0.0127837	-17844628
46	8k	0.016622	-18800966
47	8k	0.0113138	-18741474
48	8k	0.0093715	-18190442
49	10k	0.0120966	-22086729
50	10k	0.0215072	-22338561
51	10k	0.014305	-22581384
52	10k	0.0123635	-22606313
53	20k	0.0278435	-45978687
54	20k	0.0245053	-45195405
55	20k	0.0184613	-47854708
56	20k	0.0198698	-46420311
57	40k	0.0399019	-92003321
58	40k	0.0463569	-94397064
59	40k	0.0450415	-88783643
60	40k	0.0444899	-93017025
61	80k	0.1068958	-186834082
62	80k	0.1078061	-185997521
63	80k	0.106871	-182065015
64	80k	0.1044562	-180803872

Tabella 8: Risultati algoritmo di *Kruskal* (2 di 3)

N.	Graph Size	Time (s)	MST cost
65	100k	0.1403172	-230698391
66	100k	0.1447736	-230168572
67	100k	0.1539362	-231393935
68	100k	0.1883876	-231011693

Tabella 9: Risultati algoritmo di *Kruskal* (3 di 3)

### 3 Conclusioni

Mettendo a confronto i tre algoritmi è subito evidente come ci sia una evidente differenza di performance tra *Kruskal* e *Prim* con *Naive Kruskal*.

*Prim* e *Naive Kruskal* evidenziano un andamento simile, solo che *Naive Kruskal* inizia a subire un significativo incremento del tempo di risoluzione prima di *Prim*, questo perché... Mentre *Kruskal* non richiede neanche mezzo secondo per la risoluzione anche per i grafi più grandi perché...

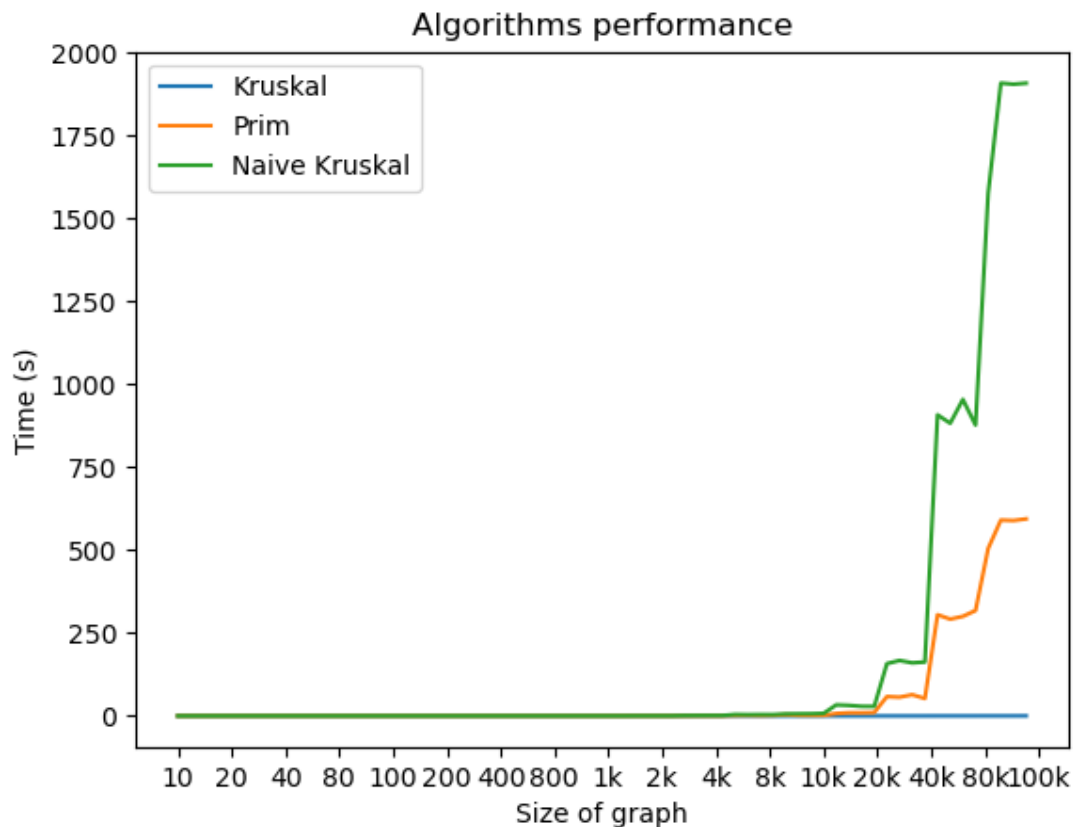


Figura 4: Performance dei tre algoritmi a confronto.