Confronto tra algoritmi per il calcolo di MST

di Costantino Marco

1 Introduzione

Gli algoritmi implementati sono:

- L'algoritmo di Prim;
- L'algoritmo di Kruskal, versione naive (senza union find);
- L'algoritmo di Kruskal (usando union find).

Il linguaggio scelto per l'implementazione degli algoritmi è *python*.

Nonostante gli svantaggi in termini di prestazioni ho scelto python per i seguenti motivi:

- Per acquisire ulteriore esperienza con il linguaggio che è centrale in un altro corso che sto frequentando (Deep Learning);
- Per la velocità che permette in fase di sviluppo, che ho ritenuto importante volendo affrontare l'esercizio da solo.

Le strutture dati usate (Heap e Union Find) sono state implementate usando semplice python. Per migliorare le prestazioni ho usato, al posto dell'interprete standard di python, *pypy* un compilatore just-in-time per codice python.

Per verificare la correttezza delle implementazioni, ho eseguito il codice su degli altri grafi a presenti nella repository github **standford-algs** di cui sono a disposizione anche i pesi degli MST.

Come illustrerò nelle successive sezioni, le prestazioni ottenute sono consistenti con le analisi di complessità degli algoritmi, tuttavia sospetto che l'implementazione in python influisca sensibilmente sulle performance. Un possibile miglioramento al quale ho pensato è quello di utilizzare le strutture dati della libreria *numpy* al posto di liste e dizionari standard di python.

1.1 La struttura dati Graph

La struttura dati che ho usato per rappresentare i grafi consiste in:

- La lista dei vertici del grafo
- La lista dei lati del grafo
- Un dizionario che rappresenta la matrice di adiacenza del grafo (chiave: vertice, valore: dizionario dei vertici adiacenti, con chiave vertice, valore peso del lato)

All'inizializzazione vengono rimossi self loop e se un lato è duplicato viene tenuto quello di peso minimo.

1.2 L'algoritmo di Prim

Ho implementato l'algoritmo di Prim a partire dallo pseudo-codice visto a lezione. La struttura dati Heap implementata per l'algoritmo di Prim prevede che gli elementi di Heap siano coppie di elementi. Questa scelta deriva dal fatto che l'algoritmo ha necessità di ottenere ad ogni iterazione il vertice dal costo minore. Le coppie in Heap quindi sono coppie `[costo, vertice]`. Queste coppie sono salvate in una lista. Per garantire una maggior velocità all'accesso dei costi, Heap contiene inoltre un dizionario, in cui la chiave è il vertice e il valore è il costo associato. Questo dizionario inoltre mi consente di verificare con complessità media $O(1)$ se un vertice è contenuto nell'Heap.

1.3 L'algoritmo di Kruskal (naive)

Entrambe le versioni dell'algoritmo di Kruskal utilizzano l'Heap precedentemente implementato per ottenere ad ogni iterazione il lato di costo minimo. La versione naive dell'algoritmo di Kruskal utilizza una funzione `is_acyclic(edges, new_edge)`, per verificare se l'aggiunta di un vertice all'MST risulta in un grafo che presenta un ciclo, rendendo quindi l'MST non più tale. La funzione riceve in input un dizionario che rappresenta l'MST, in cui le chiavi corrispondono ai vertici e i valori sono liste di vertici adiacenti alla chiave. La funzione termina immediatamente se la coppia di vertici che rappresenta il lato da inserire non sono presenti nell'MST. Altrimenti aggiungendo quel lato potrebbe esserci un ciclo. Per verificare questa possibilità la funzione esplora gli adiacenti (e i loro adiacenti) del primo vertice del lato, fino a trovare un adiacente all'altro vertice del lato, in tal caso c'è un ciclo, oppure esaurirli, in tal caso non vi è un ciclo.

1.4 L'algoritmo di Kruskal (con Union Find)

Il punto di forza dell'algoritmo di Kruskal è l'uso intelligente che fa della struttura dati Union Find. L'implementazione della struttura dati è molto semplice seguendo lo pseudo-codice visto a lezione. Ho usato 2 dizionari, uno per tenere traccia del set di appartenenza di ogni elemento, uno per tenere traccia della dimensione dei diversi set. Questo algoritmo è stato di gran lunga il più semplice da sviluppare. È sorprendente che un approccio così semplice sia anche quello che da luogo alle migliori prestazioni.

2 Domanda 1 - I risultati

I risultati sono stati raccolti usando la funzione di python `time.time()`, prima e dopo l'esecuzione dei diversi algoritmi e sottraendo i valori ottenuti, questi sono stati salvati in dei file `.csv`, insieme ai pesi degli MST ottenuti, verificati essere uguali tra i 3 algoritmi a runtime.

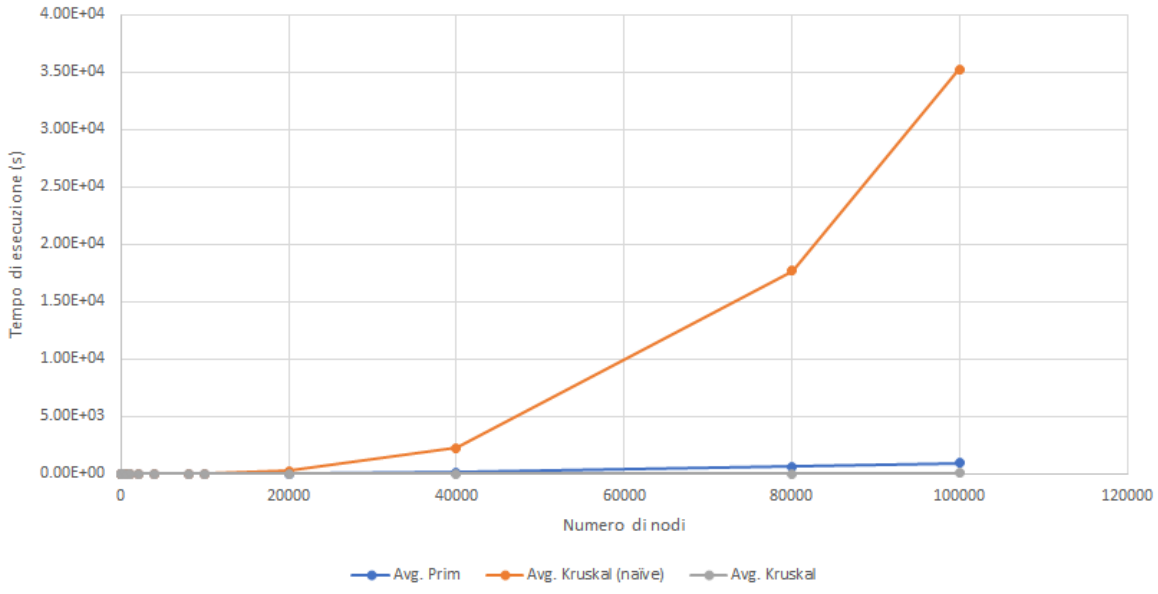
I grafi sono identificati da id e numero di nodi, i tempi sono misurati in secondi e scritti in notazione scientifica.

Id_#Vertici	Prim	Kruskal (n)	Kruskal	MST
01_10	0.00E+00	0.00E+00	0.00E+00	29316
02_10	0.00E+00	0.00E+00	0.00E+00	2126
03_10	9.97E-04	0.00E+00	9.97E-04	-44765
04_10	0.00E+00	0.00E+00	0.00E+00	20360
05_20.	0.00E+00	0.00E+00	1.28E-02	-32021
06_20.	0.00E+00	0.00E+00	0.00E+00	18596
07_20	1.57E-02	0.00E+00	0.00E+00	-42560
08_20	0.00E+00	0.00E+00	0.00E+00	-37205
09_40	3.28E-02	0.00E+00	6.02E-03	-122078
10_40	2.25E-02	0.00E+00	6.56E-03	-37021
11_40	2.73E-02	0.00E+00	0.00E+00	-79570
12_40	1.57E-02	0.00E+00	1.53E-03	-79741
13_80	3.13E-02	1.56E-02	1.94E-02	-139926
14_80	2.16E-02	0.00E+00	1.70E-02	-211345
15_80	2.08E-02	0.00E+00	2.06E-03	-110571
16_80	1.57E-02	1.56E-02	3.66E-03	-233320
17_100	1.56E-02	1.56E-02	1.19E-02	-141960

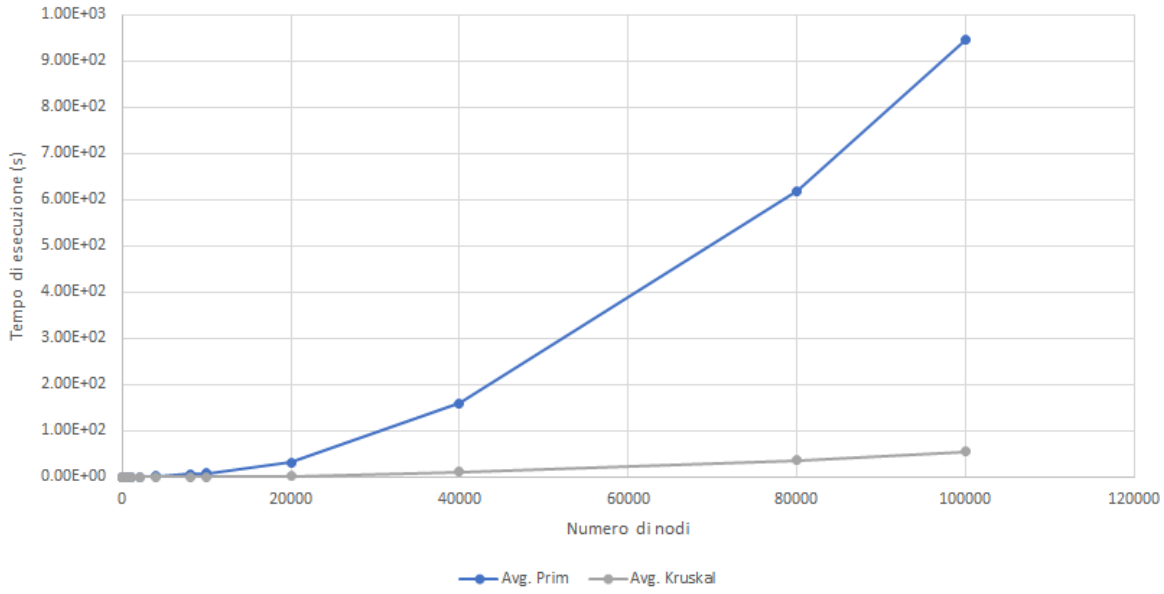
Id_#Vertici	Prim	Kruskal (n)	Kruskal	MST
18_100	2.03E-02	0.00E+00	1.73E-02	-271743
19_100	1.86E-02	1.56E-02	1.58E-02	-288906
20_100	1.63E-02	1.56E-02	9.97E-04	-232178
21_200	3.13E-02	1.56E-02	2.13E-02	-510185
22_200	3.22E-02	1.56E-02	3.83E-02	-515136
23_200	6.41E-03	1.56E-02	1.57E-02	-444357
24_200	3.27E-02	0.00E+00	1.69E-02	-393278
25_400	3.99E-02	1.56E-02	2.95E-02	-1122919
26_400	4.66E-02	1.56E-02	1.22E-02	-788168
27_400	4.69E-02	1.56E-02	3.30E-02	-895704
28_400	4.50E-02	3.12E-02	1.57E-02	-733645
29_800	9.97E-02	6.25E-02	1.72E-02	-1541291
30_800	6.80E-02	4.69E-02	2.99E-02	-1578294
31_800	6.62E-02	4.69E-02	1.67E-02	-1675534
32_800	7.34E-02	4.68E-02	1.07E-02	-1652119
33_1000	1.00E-01	7.81E-02	1.57E-02	-2091110
34_1000	1.51E-01	7.81E-02	3.55E-02	-1934208
35_1000	1.33E-01	6.25E-02	1.26E-02	-2229428
36_1000	1.17E-01	6.25E-02	1.80E-02	-2359192
37_2000	4.66E-01	4.22E-01	4.20E-02	-4811598
38_2000	4.47E-01	4.06E-01	4.86E-02	-4739387
39_2000	4.41E-01	4.53E-01	4.34E-02	-4717250
40_2000	5.14E-01	4.37E-01	4.59E-02	-4537267
41_4000	1.81E+00	2.86E+00	8.96E-02	-8722212
42_4000	1.88E+00	3.28E+00	9.03E-02	-9314968
43_4000	1.83E+00	2.84E+00	1.14E-01	-9845767
44_4000	1.82E+00	3.11E+00	8.91E-02	-8681447
45_8000	7.39E+00	2.09E+01	2.78E-01	-17844628
46_8000	6.00E+00	2.06E+01	1.56E-01	-18800966
47_8000	4.56E+00	2.05E+01	1.56E-01	-18741474
48_8000	4.69E+00	1.80E+01	1.56E-01	-18190442
49_10000	7.59E+00	3.42E+01	2.50E-01	-22086729
50_10000	7.69E+00	3.38E+01	2.19E-01	-22338561
51_10000	7.50E+00	3.32E+01	2.34E-01	-22581384
52_10000	7.56E+00	3.59E+01	2.34E-01	-22606313

Id_#Vertici	Prim	Kruskal (n)	Kruskal	MST
53_20000	3.14E+01	2.87E+02	1.84E+00	-45978687
54_20000	3.13E+01	2.83E+02	1.81E+00	-45195405
55_20000	3.10E+01	2.90E+02	1.78E+00	-47854708
56_20000	3.20E+01	2.82E+02	1.80E+00	-46420311
57_40000	1.68E+02	2.26E+03	1.09E+01	-92003321
58_40000	1.54E+02	2.29E+03	9.43E+00	-94397064
59_40000	1.47E+02	2.22E+03	9.67E+00	-88783643
60_40000	1.67E+02	2.25E+03	1.19E+01	-93017025
61_80000	6.98E+02	1.80E+04	3.52E+01	-186834082
62_80000	6.00E+02	1.70E+04	3.47E+01	-185997521
63_80000	5.92E+02	1.78E+04	3.65E+01	-182065015
64_80000	5.85E+02	1.81E+04	3.56E+01	-180803872
65_100000	9.49E+02	3.54E+04	5.62E+01	-230698391
66_100000	9.36E+02	3.50E+04	5.43E+01	-230168572
67_100000	9.58E+02	3.55E+04	5.66E+01	-231393935
68_100000	9.38E+02	3.51E+04	5.42E+01	-231011693

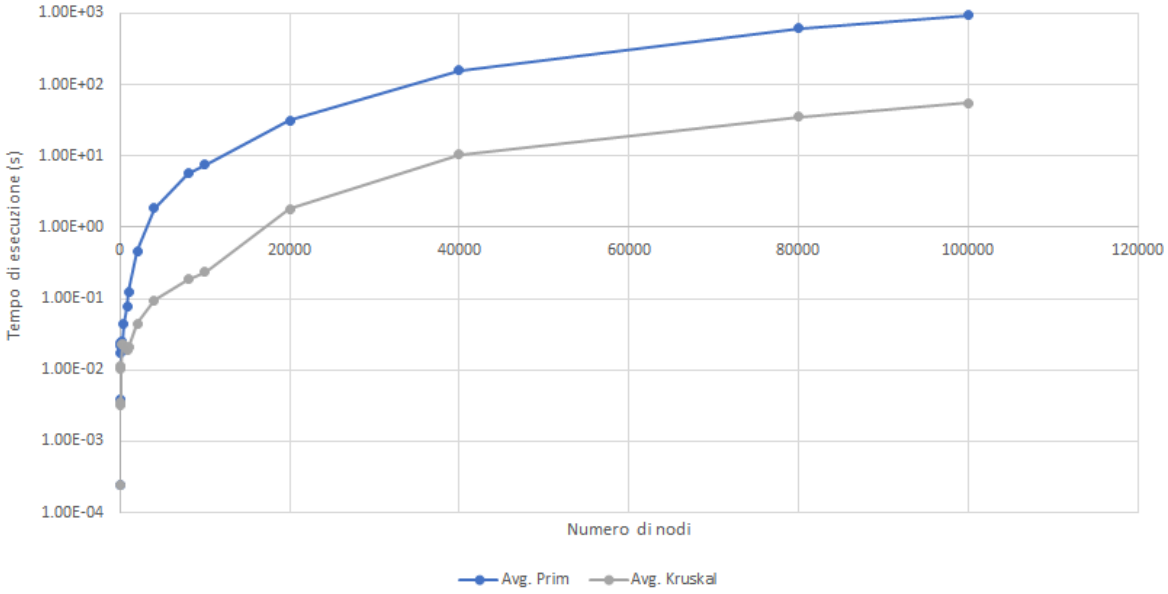
Vediamo i risultati in forma grafica:



È evidente come la differenza di complessità tra gli algoritmi; infatti la complessità di Kruskal naive è $O(mn)$ contro la complessità degli altri 2 algoritmi: $O(m \log n)$. Vediamo più da vicino l'algoritmo di Prim e la versione con Union Find dell'algoritmo di Kruskal:



Usiamo la scala logaritmica per vedere meglio cosa succede quando i grafi sono piccoli:



3 Domanda 2 – Conclusioni

Dai risultati è evidente la differenza di complessità temporale tra la versione naive dell'algoritmo di Kruskal $O(mn)$ e gli altri due algoritmi $O(m \log n)$. Infatti, per un grafo con 100k nodi, l'algoritmo di Kruskal versione naive impiega circa 9.7 ore mentre la versione dell'algoritmo che usa Union Find ci mette 54 secondi.

Mi aspetto un buon margine di miglioramento usando un linguaggio di programmazione più orientato alle prestazioni.

Confrontando l'algoritmo di Kruskal e l'algoritmo di Prim, l'algoritmo di Kruskal si comporta consistentemente meglio dal punto di vista temporale; l'algoritmo di Kruskal che usa Union Find è il più efficente (temporalmente) dei tre.

Tuttavia, poichè ho usato Heap allo stesso modo in tutti e 3 gli algoritmi, almeno per la mia implementazione è possibile dire che l'algoritmo di Kruskal necessita di più memoria rispetto all'algoritmo di Prim.

Dato un grafo con 100k nodi, l'algoritmo di Prim impiega circa 0.27 ore per ottenere l'MST sul grafo. È possibile immaginare delle applicazioni a memoria limitata (ad esempio la navigazione di un rover su marte) in cui le limitazioni a livello temporale non sono altrettanto stringenti e l'algoritmo di Prim potrebbe essere più appropriato dell'algoritmo di Kruskal.