Algoritmi Avanzati - Laboratorio 1

Minimum Spanning Tree

Lucchetta Bryan	Parolari Luca	Schiabel Alberto
1237584	1236601	1236598

18 aprile 2020

Indice

1 Abstract **2**

2 Linguaggio di programmazione scelto **2**
2.1 IDE e compilatore 2

3 Benchmark **3**

4 Struttura del codice **4**

5 Scelte implementative **4**
5.1 Codice templatizzato 4
5.2 Rappresentazione del grafo 4
5.3 Algoritmi 6

6 Analisi delle performance **8**
6.1 Confronto di runtime tra programmi 8
6.2 Domanda #1 8
6.3 Domanda #2 16

7 Test **17**

8 Estensioni e originalità **17**
8.1 Prim con k-ary Heap 17
8.2 Kruskal con Disjoint-Set e path-compression 18

9 Conclusioni **18**

1 Abstract

Questo primo homework di laboratorio di Algoritmi Avanzati ha lo scopo di implementare e confrontare algoritmi per il calcolo del Minimum Spanning Tree di un grafo. Sono stati considerati solo grafi non diretti pesati.

Gli algoritmi implementati sono 3:

1. Algoritmo di Prim implementato con la struttura dati MinHeap;
2. Algoritmo di Kruskal nella sua implementazione "naive" di complessità O(mn);
3. Algoritmo di Kruskal implementato con la struttura dati Disjoint-Set (Union-Find).

Abbiamo considerato anche alcune estensioni originali rispetto agli algoritmi visti in classe; esse sono discusse e presentate nella sezione Estensioni e originalità.

Il codice è scritto in C++17 ed è opportunamente commentato per facilitarne la comprensione. Non è stata usata alcuna libreria esterna.

Le risposte alle 2 domande principali dell'homework sono riportate nella sezione Analisi delle performance.

2 Linguaggio di programmazione scelto

Abbiamo scelto di usare C++17 per implementare questo homework. Le ragioni sono principalmente dovute al fatto che:

- è un linguaggio compilato e fortemente orientato ai tipi;
- è privo di garbage collector, il che velocizza l'esecuzione del codice;
- supporta i tipi generici (*template*);
- permette di creare personalizzazioni a compile-time senza alcun impatto a runtime;
- ha un'ampia libreria standard consolidata e ben mantenuta;
- se opportunamente usato, permette di limitare al minimo il consumo di memoria;
- la sua sintassi è intuitiva, leggibile e facile da seguire, anche se un po' più prolissa rispetto ad altri linguaggi;
- supporta la *move semantics* e altre ottimizzazioni che permettono al programmatore di evitare inutili copie di oggetti e overhead in memoria.

2.1 IDE e compilatore

Poiché il nostro sistema operativo di sviluppo è Windows 10, abbiamo usato l'IDE Visual Studio 2019 Community e il suo compilatore *MSVC v142 x64/x86*.

Nell'archivio allegato a questa relazione abbiamo incluso un *Makefile* per permettere la compilazione su altri sistemi operativi usando *g++9.3*.

3 Benchmark

Abbiamo deciso di rendere il processo di misurazione del tempo di esecuzione dei nostri algoritmi esterno al codice dei programmi sviluppati. Il nostro benchmark tiene quindi conto di tutto, ovvero:

- Tempo necessario a leggere il file di input in un vettore temporaneo;
- Tempo per trasformare il vettore temporaneo in una lista di adiacenza;
- Tempo per eseguire l'algoritmo vero e proprio per il calcolo dell'**MST**;
- Tempo per scorrere l'**MST** ritornato dall'algoritmo per calcolarne il peso totale.

Lo script Powershell *benchmark.ps1*, che richiama a sua volta lo script *time.ps1*, si occupa di misurare il tempo richiesto a ogni programma sviluppato per questo homework per calcolare il peso del Minimum Spanning Tree di ogni dataset di input. Al termine dell'esecuzione, esso genera un file CSV per ogni programma di cui è stato fatto il benchmark. Il nome di ogni file CSV generato contiene il nominativo del programma misurato e il timestamp in cui è stato creato.

Le colonne dei file CSV sopracitati sono le seguenti:

- **ms**: tempo in millisecondi per eseguire il programma su un singolo file di input;
- **output**: risultato del programma, ovvero peso dell'**MST** del grafo letto in input;
- **n**: numero di nodi del grafo letto;
- **m**: numero di archi del grafo letto;
- **filename**: nome del file di input letto.

Per rendere i risultati del benchmark quanto più stabili e affidabili possibile, abbiamo preso le seguenti precauzioni:

- Abbiamo usato sempre lo stesso computer per misurare il tempo di esecuzione dei programmi implementati;
- Abbiamo chiuso tutti i programmi in foreground e disabilitato quanti più servizi possibile in background;
- Abbiamo disabilitato la connessione Internet del computer scelto
- Abbiamo fatto più misurazioni, e abbiamo tenuto la media delle misurazioni effettuate.

4 Struttura del codice

Abbiamo strutturato il codice come un'unica soluzione Visual Studio contenente molteplici progetti, uno per ogni algoritmo implementato. Una soluzione Visual Studio può essere vista come un macro-progetto che contiene più sottomoduli.

Abbiamo creato una cartella *Shared* per contenere le classi usate per rappresentare i grafi come liste di adiacenza (*AdjListGraph.h*), le strutture dati create "from scratch" (*BinaryHeap.h*, *PriorityQueue.h* e *DisjointSet.h*), la classe che individua cicli in un grafo usando Depth-First-Search (*DFSCycleDetection.h*) e delle utilities usate in tutti i progetti.

Abbiamo configurato Visual Studio per importare automaticamente i file di header salvati in *Shared* durante la compilazione di ogni sottoprogetto. Analogamente, tale cartella è referenziata nell'opzione *-i* di g++ nel *Makefile*.

5 Scelte implementative

5.1 Codice templatizzato

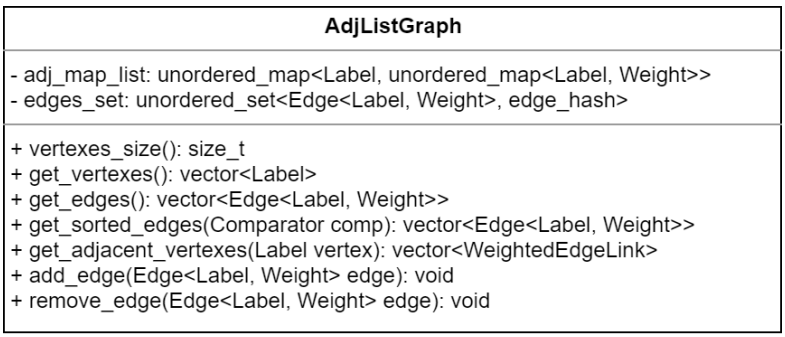
5.2 Rappresentazione del grafo

Le due possibilità standard per rappresentare un grafo sono

- **Matrici di adiacenza**
- **Liste di adiacenza**

In questa implementazione abbiamo scelto di utilizzare una lista di adiacenza affiancata da una struttura dati aggiuntiva unordered_set per migliorare i tempi di alcune operazioni fondamentali. Il relativo diagramma di classe è possibile ri-trovarlo nella figura 1 dove vengono riportati gli attributi e i metodi offerti dalla classe.

Figura 1: Diagramma di classe per AdjListGraph



Nonostante la lista di adiacenza sarebbe già sufficiente per salvare tutte le informazioni necessarie alla rappresentazione di un grafo, in alcune operazioni risulta essere lenta.

L'esempio di operazione che ci ha spinto ad inserire una nuova struttura dati (unordered_set in questo caso) è il metodo per ritornare tutti gli archi di un grafo in quanto tale metodo se implementato avendo a disposizione la sola lista di adiacenza prevederebbe in maniera semplice la scansione dell'intera lista di adiacenza, con l'inserimento degli archi che si ritrovano in ogni vertice. Tale semplice implementazione porterebbe alla creazione di un vettore in cui se sussiste un arco tra un vertice 2 e un vertice 3, il vettore conterebbe sia l'arco $2 \rightarrow 3$ e $3 \rightarrow 2$ per costruzione della lista di adiacenza.

Le cose si complicano maggiormente se si restringe la richiesta di non restituire gli archi doppi come abbiamo già osservato. A tale proposito si potrebbe ricercare l'eventuale presenza di tali archi doppi ed eliminarli di conseguenza, ma il tutto richiederebbe maggiori operazioni e dunque un maggior tempo di esecuzione.

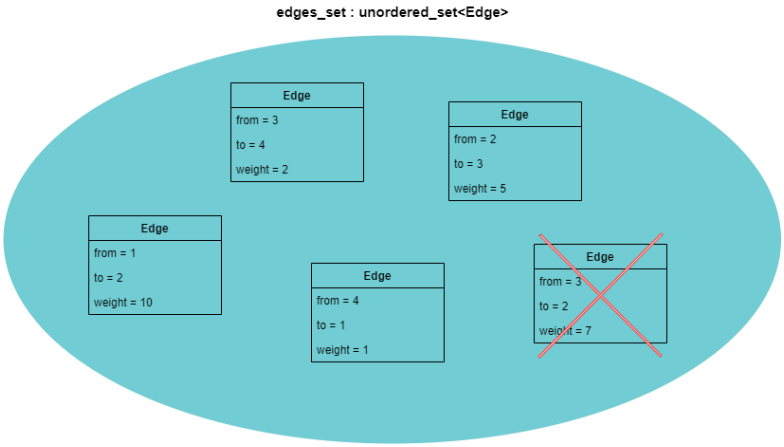
Per tali ragioni è stato deciso di affiancare alla lista di adiacenza, un'apposita struttura dati che permettesse di tenere traccia degli archi e di avere tempi di esecuzione costanti per le operazioni di aggiunta, rimozione e ricerca. Così facendo è possibile garantire che l'operazione di restituzione degli archi singoli di un grafo abbia tempo O(m) se si richiede di restituire un vettore, o addirittura O(1) se si richiede la restituzione di un puntatore a tale struttura. E' stato scelto dunque di utilizzare come struttura dati da affiancare unordered_set perchè oltre ad avere le caratteristiche richieste, se appositamente configurata permette di garantire l'assenza di archi doppi, compresi quelli visti nell'esempio precedente (ossia $2 \rightarrow 3$ e $3 \rightarrow 2$). Per fare ciò è stato dunque opportunamente configurato l'operatore di uguaglianza e la funzione di hash, in modo che archi equivalenti abbiamo la stessa funzione di hash e siano riconosciuti come uguali, evitando così l'inserimento di un arco doppio visto che che unordered_set non prevede elementi doppi al suo interno.

Una rappresentazione astratta di tale struttura è possibile visualizzarla nella figura 2, dove è possibile notare che se viene richiesto l'inserimento di un arco $3 \rightarrow 2$ ove già presente un arco $2 \rightarrow 3$ questo non viene inserito da unordered_set.

L'ultimo problema non ancora affrontato rigurda la possibilità di inserire un arco doppio tra due nodi uguali (come richiesto da problema), con la differenza che questi 2 archi hanno un peso diverso. Siccome la nostra implementazione non prevede la presenza di tali archi doppi, la funzione add_eddge() si occupa di verificare l'eventuale presenza di un arco già inserito e di conseguenza i relativi pesi, per andare a modificare il peso qualora il peso del nuovo arco sia inferiore al peso dell'arco precedentemente già inserito. Per fare questo ad ogni aggiunta di un nuovo arco si va a verificare nella lista di adiacenza l'esistenza di un arco per quei due vertici:

- **se l'arco era già stato aggiunto:** si confrontano i due pesi e si aggiorna il peso dell'arco solo nell'eventualità in cui il nuovo peso sia inferiore a quello già presente nella lista di adiacenza. Dopodichè se c'è stato un aggiornamento si aggiorna anche il relativo unordered_set, eliminando l'arco precendente e si riaggiunge l'arco con il nuovo peso.

Figura 2: Visualizzazione astratta di edges_set



- **se l'arco non era già stato inserito:** si inserisce l'arco sia nella lista di adiacenza che nell'unordered_set.

Tutte le operazioni di aggiunta, per come sono implementati unordered_set e unordered_map in C++ hanno tempo costante e dunque l'operazione di aggiunta richiede tempo costante.

5.2.1 Strutture Dati

5.2.2 Binary Heap

5.2.3 Priority Queue

5.2.4 Disjoint Set

5.3 Algoritmi

5.3.1 Costruzione del Grafo

In questa parte si va a leggere il file di input per poi darlo in pasto alla funzione di costruzione della rappresentazione del grafo come spiegato nella sezione precedente.....

5.3.2 Prim con Binary Heap

5.3.3 Kruskal Naive

L'algoritmo Kruskal Naive è stato implementato a partire dallo pseudo codice visto in classe. A partire da questo ci si è subito accorti che una delle peculiarità di tale algoritmo e la necessità di verificare ad ogni inserimento di un arco di peso minimo la presenza di un ciclo nell'MST calcolato. Per ottenere tale funzionalità abbiamo deciso di modificare DFS in maniera da rilevare la presenza di un ciclo in un grafo. Per fare questo è stato creata la classe DFSCycleDetection, che è possibile ritrovare nella cartella "Shared", che

data la rappresentazione del grafo e richiamando l'apposito metodo has_cycle() è in grado di rilevare la presenza di un ciclo all'interno di esso usando per l'appunto una visita in profondità.

E' stato pertanto necessario modificare DFS nel seguente modo:

- non è necessario avere una label per ogni arco che etichetti quell'arco come "DISCOVERY EDGE" o "BACK EDGE"
- quando un arco verrebbe etichettato come "BACK EDGE" nello pseudo codice visto in classe possiamo affermare che nel grafo è presente un ciclo. Non è necessario riscostruire tutto il ciclo come visto in un ulteriore modifica di DFS in classe.
- al posto di tener traccia di ogni arco se è stato visitato o meno attraverso l'attributo ID visto in clase, è stato deciso di creare un set di archi già visitati dove l'inseriemento e la ricerca avviene in tempo costante.
- per essere sicuri di non prendere in considerazione il vertice da cui viene lanciata ricorsivamente la DFS (ossia per non ritrovare il nodo da cui siamo venuti) è stata passata alla ricorsione il vertice padre, in modo che quando si scansiona la lista di adiacenza del vertice in considerazione non si consideri il vertice padre, simulando così il comportamento della funzione "opposite(v,e)" nel pseudo codice.

Per risolvere il problema di più di un eventuale componenente connessa nel grafo, vengono sempre scansionati tutti i nodi non ancora visitati attraverso un ciclo for su tutti i nodi come visto in classe.

Un ulteriore piccola ottimizzazione aggiunta è la verifica della presenza di soli 2 vertici in cui non è neccessario lanciare una visita in profondita, in quanto un ciclo non può sussistere tra 2 vertici se gli archi non sono diretti e non esistono self-loop TODO: VERIFICARE QUESTA SUPPOSIZIONE.

Un altra ulteriore ottimizzazione di tale algoritmo è la verifica ad ogni aggiunta di un nuovo arco che non crea un ciclo nel grafo, della soglia massima di archi che un MST può avere, che come visto in classe è pari al numero di nodi totali del grafo - 1. Pertanto non appena l'MST che si sta calcolando raggiunge la soglia prestabilità e possibile ritornare immediatamente l'MST.

Tale algoritmo risulta avere complessita dunque complessità O(mn) come visto in classe.

5.3.4 Kruskal con Disjoint Set

6 Analisi delle performance

6.1 Confronto di runtime tra programmi

	8k	10k	20k	20k	80k	100k
KruskalNaive	7431.57	11939.8	59257	366313	1.99612e+06	3.12073e+06
KruskalUnionFind	204.429	254.504	599.704	1249.7	3010.99	4050.49
Differenza	7227.15	11685.3	58657.3	365064	1.99311e+06	3.11668e+06
Miglioramento %	97.25	97.87	98.99	99.66	99.85	99.87

Tabella 1: Confronto tra KruskalNaive e KruskalUnionFind.

	8k	10k	20k	20k	80k	100k
KruskalNaive	7431.57	11939.8	59257	366313	1.99612e+06	3.12073e+06
KruskalUnionFindCompressed	204.429	256.287	599.704	1271.32	3062.76	4050.49
Differenza	7227.15	11683.5	58657.3	365042	1.99306e+06	3.11668e+06
Miglioramento %	97.25	97.85	98.99	99.65	99.85	99.87

Tabella 2: Confronto tra KruskalNaive e KruskalUnionFindCompressed.

	8k	10k	20k	20k	80k	100k
KruskalNaive	7431.57	11939.8	59257	366313	1.99612e+06	3.12073e+06
PrimBinaryHeap	227.5	273.581	646.039	1365.97	3197.83	4372.45
Differenza	7204.07	11666.2	58610.9	364947	1.99293e+06	3.11636e+06
Miglioramento %	96.94	97.71	98.91	99.63	99.84	99.86

Tabella 3: Confronto tra KruskalNaive e PrimBinaryHeap.

	8k	10k	20k	20k	80k	100k
KruskalUnionFind	204.429	254.504	599.704	1249.7	3010.99	4050.49
KruskalUnionFindCompressed	204.429	256.287	599.704	1271.32	3062.76	4050.49
Differenza	0	-1.783	0	-21.615	-51.773	0
Miglioramento %	0	-0.7	0	-1.73	-1.72	0

Tabella 4: Confronto tra KruskalUnionFind e KruskalUnionFindCompressed.

6.2 Domanda #1

Eseguite i tre algoritmi che avete implementato (Prim, Kruskal naive e Kruskal efficiente) sui grafi del dataset. Misurate i tempi di calcolo dei tre algoritmi e create un grafico che mostri la variazione dei tempi di calcolo al

	8k	10k	20k	20k	80k	100k
PrimBinaryHeap	227.5	273.581	646.039	1365.97	3197.83	4372.45
KruskalUnionFind	204.429	254.504	599.704	1249.7	3010.99	4050.49
Differenza	23.071	19.077	46.335	116.265	186.843	321.954
Miglioramento %	10.14	6.97	7.17	8.51	5.84	7.36

Tabella 5: Confronto tra PrimBinaryHeap e KruskalUnionFind.

	8k	10k	20k	20k	80k	100k
PrimBinaryHeap	227.5	273.581	646.039	1365.97	3197.83	4372.45
KruskalUnionFindCompressed	204.429	256.287	599.704	1271.32	3062.76	4050.49
Differenza	23.071	17.294	46.335	94.65	135.07	321.954
Miglioramento %	10.14	6.32	7.17	6.93	4.22	7.36

Tabella 6: Confronto tra PrimBinaryHeap e KruskalUnionFindCompressed.

variare del numero di vertici nel grafo. Per ognuna delle istanze del problema, riportate il peso del minimum spanning tree ottenuto dagli algoritmi.

Il risultato del minimum spanning tree ottenuto dagli algoritmi è illustrato nelle tabelle 7, 8, 9 e 10.

Tabella 7: Risultati di KruskalNaive

ms	MST	File di input
7.75	29316	input_random_01_10.txt
6.416	2126	input_random_02_10.txt
16.695	-44765	input_random_03_10.txt
6.39	20360	input_random_04_10.txt
6.863	-32021	input_random_05_20.txt
7.121	18596	input_random_06_20.txt
6.359	-42560	input_random_07_20.txt
6.331	-37205	input_random_08_20.txt
6.69	-122078	input_random_09_40.txt
16.272	-37021	input_random_10_40.txt
6.751	-79570	input_random_11_40.txt
6.552	-79741	input_random_12_40.txt
17.146	-139926	input_random_13_80.txt
7.214	-211345	input_random_14_80.txt
17.171	-110571	input_random_15_80.txt
7.783	-233320	input_random_16_80.txt
8.013	-141960	input_random_17_100.txt
17.662	-271743	input_random_18_100.txt
7.848	-288906	input_random_19_100.txt

Tabella 7 continuata dalla pagina precedente

ms	MST	File di input
17.703	-232178	input_random_20_100.txt
23.274	-510185	input_random_21_200.txt
12.838	-515136	input_random_22_200.txt
14.016	-444357	input_random_23_200.txt
14.392	-393278	input_random_24_200.txt
42.18	-1122919	input_random_25_400.txt
27.25	-788168	input_random_26_400.txt
28.474	-895704	input_random_27_400.txt
29.456	-733645	input_random_28_400.txt
91.985	-1541291	input_random_29_800.txt
79.016	-1578294	input_random_30_800.txt
79.626	-1675534	input_random_31_800.txt
79.294	-1652119	input_random_32_800.txt
130.378	-2091110	input_random_33_1000.txt
112.244	-1934208	input_random_34_1000.txt
117.363	-2229428	input_random_35_1000.txt
118.55	-2359192	input_random_36_1000.txt
475.061	-4811598	input_random_37_2000.txt
455.737	-4739387	input_random_38_2000.txt
458.535	-4717250	input_random_39_2000.txt
475.395	-4537267	input_random_40_2000.txt
1874.333	-8722212	input_random_41_4000.txt
1830.927	-9314968	input_random_42_4000.txt
1830.862	-9845767	input_random_43_4000.txt
1913.81	-8681447	input_random_44_4000.txt
7462.598	-17844628	input_random_45_8000.txt
7379.777	-18800966	input_random_46_8000.txt
7754.247	-18741474	input_random_47_8000.txt
7431.575	-18190442	input_random_48_8000.txt
11876.697	-22086729	input_random_49_10000.txt
11939.798	-22338561	input_random_50_10000.txt
11978.208	-22581384	input_random_51_10000.txt
12009.818	-22606313	input_random_52_10000.txt
57232.711	-45978687	input_random_53_20000.txt
59256.966	-45195405	input_random_54_20000.txt
54916.396	-47854708	input_random_55_20000.txt
56220.966	-46420311	input_random_56_20000.txt
383362.394	-92003321	input_random_57_40000.txt
366313.256	-94397064	input_random_58_40000.txt
371668.9	-88783643	input_random_59_40000.txt
407042.457	-93017025	input_random_60_40000.txt

Tabella 7 continuata dalla pagina precedente

ms	MST	File di input
1996123.987	-186834082	input_random_61_80000.txt
1936952.48	-185997521	input_random_62_80000.txt
1878317.784	-182065015	input_random_63_80000.txt
2021332.994	-180803872	input_random_64_80000.txt
3259681.887	-230698391	input_random_65_100000.txt
3163894.525	-230168572	input_random_66_100000.txt
3120732.732	-231393935	input_random_67_100000.txt
3252430.566	-231011693	input_random_68_100000.txt

Tabella 8: Risultati di KruskalUnionFind

ms	MST	File di input
6.25	29316	input_random_01_10.txt
6.116	2126	input_random_02_10.txt
5.994	-44765	input_random_03_10.txt
5.671	20360	input_random_04_10.txt
5.565	-32021	input_random_05_20.txt
5.708	18596	input_random_06_20.txt
5.549	-42560	input_random_07_20.txt
5.941	-37205	input_random_08_20.txt
6.219	-122078	input_random_09_40.txt
6.079	-37021	input_random_10_40.txt
6.05	-79570	input_random_11_40.txt
6.047	-79741	input_random_12_40.txt
6.399	-139926	input_random_13_80.txt
6.391	-211345	input_random_14_80.txt
6.326	-110571	input_random_15_80.txt
6.443	-233320	input_random_16_80.txt
6.702	-141960	input_random_17_100.txt
6.553	-271743	input_random_18_100.txt
6.749	-288906	input_random_19_100.txt
6.569	-232178	input_random_20_100.txt
9.304	-510185	input_random_21_200.txt
9.402	-515136	input_random_22_200.txt
9.37	-444357	input_random_23_200.txt
9.178	-393278	input_random_24_200.txt
15.026	-1122919	input_random_25_400.txt
14.475	-788168	input_random_26_400.txt
14.658	-895704	input_random_27_400.txt

Tabella 8 continuata dalla pagina precedente

ms	MST	File di input
14.569	-733645	input_random_28_400.txt
24.81	-1541291	input_random_29_800.txt
24.52	-1578294	input_random_30_800.txt
25.023	-1675534	input_random_31_800.txt
24.412	-1652119	input_random_32_800.txt
29.177	-2091110	input_random_33_1000.txt
29.452	-1934208	input_random_34_1000.txt
29.694	-2229428	input_random_35_1000.txt
29.968	-2359192	input_random_36_1000.txt
55.244	-4811598	input_random_37_2000.txt
54.85	-4739387	input_random_38_2000.txt
54.434	-4717250	input_random_39_2000.txt
55.186	-4537267	input_random_40_2000.txt
109.601	-8722212	input_random_41_4000.txt
104.416	-9314968	input_random_42_4000.txt
105.753	-9845767	input_random_43_4000.txt
105.872	-8681447	input_random_44_4000.txt
204.737	-17844628	input_random_45_8000.txt
222.969	-18800966	input_random_46_8000.txt
204.002	-18741474	input_random_47_8000.txt
204.429	-18190442	input_random_48_8000.txt
254.498	-22086729	input_random_49_10000.txt
254.504	-22338561	input_random_50_10000.txt
253.591	-22581384	input_random_51_10000.txt
254.146	-22606313	input_random_52_10000.txt
597.346	-45978687	input_random_53_20000.txt
599.704	-45195405	input_random_54_20000.txt
596.978	-47854708	input_random_55_20000.txt
604.93	-46420311	input_random_56_20000.txt
1230.469	-92003321	input_random_57_40000.txt
1249.701	-94397064	input_random_58_40000.txt
1440.58	-88783643	input_random_59_40000.txt
1254.999	-93017025	input_random_60_40000.txt
3010.987	-186834082	input_random_61_80000.txt
3020.693	-185997521	input_random_62_80000.txt
3002.066	-182065015	input_random_63_80000.txt
3030.772	-180803872	input_random_64_80000.txt
3925.031	-230698391	input_random_65_100000.txt
3953.948	-230168572	input_random_66_100000.txt
4050.495	-231393935	input_random_67_100000.txt
3969.782	-231011693	input_random_68_100000.txt

Tabella 9: Risultati di KruskalUnionFindCompressed

ms	MST	File di input
6.25	29316	input_random_01_10.txt
6.187	2126	input_random_02_10.txt
5.994	-44765	input_random_03_10.txt
5.671	20360	input_random_04_10.txt
5.565	-32021	input_random_05_20.txt
5.708	18596	input_random_06_20.txt
5.549	-42560	input_random_07_20.txt
5.941	-37205	input_random_08_20.txt
6.274	-122078	input_random_09_40.txt
6.294	-37021	input_random_10_40.txt
6.05	-79570	input_random_11_40.txt
6.047	-79741	input_random_12_40.txt
6.399	-139926	input_random_13_80.txt
6.391	-211345	input_random_14_80.txt
6.326	-110571	input_random_15_80.txt
6.529	-233320	input_random_16_80.txt
6.71	-141960	input_random_17_100.txt
6.617	-271743	input_random_18_100.txt
6.749	-288906	input_random_19_100.txt
6.68	-232178	input_random_20_100.txt
9.455	-510185	input_random_21_200.txt
9.484	-515136	input_random_22_200.txt
9.536	-444357	input_random_23_200.txt
9.224	-393278	input_random_24_200.txt
15.204	-1122919	input_random_25_400.txt
14.535	-788168	input_random_26_400.txt
14.717	-895704	input_random_27_400.txt
14.637	-733645	input_random_28_400.txt
26.847	-1541291	input_random_29_800.txt
24.52	-1578294	input_random_30_800.txt
25.122	-1675534	input_random_31_800.txt
24.472	-1652119	input_random_32_800.txt
29.576	-2091110	input_random_33_1000.txt
29.452	-1934208	input_random_34_1000.txt
29.845	-2229428	input_random_35_1000.txt
30.366	-2359192	input_random_36_1000.txt
55.63	-4811598	input_random_37_2000.txt
54.85	-4739387	input_random_38_2000.txt
54.573	-4717250	input_random_39_2000.txt

Tabella 9 continuata dalla pagina precedente

ms	MST	File di input
55.388	-4537267	input_random_40_2000.txt
109.601	-8722212	input_random_41_4000.txt
104.596	-9314968	input_random_42_4000.txt
107.273	-9845767	input_random_43_4000.txt
105.872	-8681447	input_random_44_4000.txt
204.737	-17844628	input_random_45_8000.txt
222.969	-18800966	input_random_46_8000.txt
204.002	-18741474	input_random_47_8000.txt
204.429	-18190442	input_random_48_8000.txt
254.498	-22086729	input_random_49_10000.txt
256.287	-22338561	input_random_50_10000.txt
254.163	-22581384	input_random_51_10000.txt
254.146	-22606313	input_random_52_10000.txt
597.346	-45978687	input_random_53_20000.txt
599.704	-45195405	input_random_54_20000.txt
606.613	-47854708	input_random_55_20000.txt
610.513	-46420311	input_random_56_20000.txt
1230.469	-92003321	input_random_57_40000.txt
1271.316	-94397064	input_random_58_40000.txt
1440.58	-88783643	input_random_59_40000.txt
1258.242	-93017025	input_random_60_40000.txt
3062.76	-186834082	input_random_61_80000.txt
3041.329	-185997521	input_random_62_80000.txt
3024.002	-182065015	input_random_63_80000.txt
3050.915	-180803872	input_random_64_80000.txt
3955.09	-230698391	input_random_65_100000.txt
3961.982	-230168572	input_random_66_100000.txt
4050.495	-231393935	input_random_67_100000.txt
3991.774	-231011693	input_random_68_100000.txt

Tabella 10: Risultati di PrimBinaryHeap

ms	MST	File di input
5.926	29316	input_random_01_10.txt
5.875	2126	input_random_02_10.txt
5.753	-44765	input_random_03_10.txt
6.121	20360	input_random_04_10.txt
6.39	-32021	input_random_05_20.txt
5.925	18596	input_random_06_20.txt

Tabella 10 continuata dalla pagina precedente

ms	MST	File di input
5.928	-42560	input_random_07_20.txt
6.074	-37205	input_random_08_20.txt
6.193	-122078	input_random_09_40.txt
6.118	-37021	input_random_10_40.txt
5.945	-79570	input_random_11_40.txt
6.173	-79741	input_random_12_40.txt
6.609	-139926	input_random_13_80.txt
6.378	-211345	input_random_14_80.txt
6.494	-110571	input_random_15_80.txt
6.583	-233320	input_random_16_80.txt
6.766	-141960	input_random_17_100.txt
6.774	-271743	input_random_18_100.txt
7.076	-288906	input_random_19_100.txt
6.806	-232178	input_random_20_100.txt
9.386	-510185	input_random_21_200.txt
9.438	-515136	input_random_22_200.txt
9.39	-444357	input_random_23_200.txt
9.424	-393278	input_random_24_200.txt
15.109	-1122919	input_random_25_400.txt
14.75	-788168	input_random_26_400.txt
16.088	-895704	input_random_27_400.txt
14.464	-733645	input_random_28_400.txt
25.368	-1541291	input_random_29_800.txt
25.535	-1578294	input_random_30_800.txt
25.997	-1675534	input_random_31_800.txt
25.288	-1652119	input_random_32_800.txt
30.225	-2091110	input_random_33_1000.txt
31.26	-1934208	input_random_34_1000.txt
31.055	-2229428	input_random_35_1000.txt
30.876	-2359192	input_random_36_1000.txt
58.444	-4811598	input_random_37_2000.txt
58.364	-4739387	input_random_38_2000.txt
56.956	-4717250	input_random_39_2000.txt
56.965	-4537267	input_random_40_2000.txt
117.941	-8722212	input_random_41_4000.txt
109.843	-9314968	input_random_42_4000.txt
112.947	-9845767	input_random_43_4000.txt
128.115	-8681447	input_random_44_4000.txt
221.552	-17844628	input_random_45_8000.txt
215.404	-18800966	input_random_46_8000.txt
216.47	-18741474	input_random_47_8000.txt

Tabella 10 continuata dalla pagina precedente

ms	MST	File di input
227.5	-18190442	input_random_48_8000.txt
271.074	-22086729	input_random_49_10000.txt
273.581	-22338561	input_random_50_10000.txt
270.186	-22581384	input_random_51_10000.txt
268.648	-22606313	input_random_52_10000.txt
659.905	-45978687	input_random_53_20000.txt
646.039	-45195405	input_random_54_20000.txt
634.594	-47854708	input_random_55_20000.txt
657.896	-46420311	input_random_56_20000.txt
1324.008	-92003321	input_random_57_40000.txt
1365.966	-94397064	input_random_58_40000.txt
1337.124	-88783643	input_random_59_40000.txt
1390.265	-93017025	input_random_60_40000.txt
3197.83	-186834082	input_random_61_80000.txt
3254.522	-185997521	input_random_62_80000.txt
3265.034	-182065015	input_random_63_80000.txt
3255.252	-180803872	input_random_64_80000.txt
4253.77	-230698391	input_random_65_100000.txt
4355.529	-230168572	input_random_66_100000.txt
4372.449	-231393935	input_random_67_100000.txt
4292.354	-231011693	input_random_68_100000.txt

6.3 Domanda #2

> Commentate i risultati che avete ottenuto: come si comportano gli algoritmi rispetti alle varie istanze? C'è un algoritmo che riesce sempre a fare meglio degli altri? Quale dei tre algoritmi che avete implementato è più efficiente?

Dalle tabelle di confronto 1 e 5 è evidente che **KruskalUnionFind** è il più efficiente tra gli algoritmi di cui era richiesta l'implementazione. Inoltre, osservando la tabella di confronto 4, è possibile notare che l'utilizzo della tecnica *path-compression* nella struttura dati **Disjoint-Set** in realtà non abbia portato benefici, e anzi abbia peggiorato leggermenete le performance (che sono comunque migliori di *PrimBinaryHeap*, vedasi la tabella di confronto 6).

In teoria, KruskalUnionFindCompressed avrebbe dovuto essere l'algoritmo più efficiente di tutti (complessità temporale: **TODO**). PrimBinaryHeap e KruskalUnionFind hanno la stessa complessità temporale teorica, ma in realtà la complessità delle operazioni del BinaryHeap ha un coefficiente più elevato rispetto a quelle della struttura dati Disjoint-Set.

Infine, la complessità teorica di KruskalNaive (**TODO**) c'avevano già fatto immaginare che i tempi di calcolo nella pratica sarebbero esplosi a partire da grafi con poche centinaia di nodi e archi, e così è stato.

Non abbiamo un'idea certa del perché Kruskal implementato con *DisjointSetCompressed* sia poco meno performante di *DisjointSet*. La nostra ipotesi è che, per gli input forniti ai programmi, il *cache behaviour* di KruskalUnionFind sia molto migliore di quello di KruskalUnionFindCompressed.

7 Test

Abbiamo usato i dataset di *stanford-algs* per confrontare i risultati delle nostre implementazioni degli algoritmi richiesti con gli output attesi.
Nella cartella *test* sono presenti i 68 file di input e i relativi 68 file di output atteso di tali dataset. Abbiamo usato gli script Powershell *testall.ps1* e *test.ps1* (presenti nella root del progetto consegnato) per automatizzare il testing dei nostri programmi.
Abbiamo inoltre usato lo strumento di Continuous Integration *Travis* per testare continuamente la solidità del codice nella nostra repository ad ogni push nel branch "master".

8 Estensioni e originalità

Oltre alle 3 implementazioni richieste dalla consegna dell'homework, abbiamo deciso di esplorare qualche altra estensione degli algoritmi per il calcolo del Minimum Spanning Tree visti a lezione.

8.1 Prim con k-ary Heap

Dal punto di vista teorico, le Fibonacci Heap hanno una complessità temporale migliore delle k-ary Heap. Tuttavia, dal punto di vista pratico, le k-ary Heap hanno una performance migliore perché la loro struttura permette loro di sfruttare la cache locality. Inoltre le Fibonaccy Heap hanno un coefficiente di complessità nascosto piuttosto elevato. Un altro motivo per cui abbiamo deciso di non implementare le Fibonacci Heap è che sono più complesse da implementare rispetto alle k-ary Heap.

Quando le k-ary heap sono usate per implementare code di priorità, l'operazione di aggiornamento del valore della chiave è più veloce rispetto ad una Binary Heap ($\mathcal{O}(\log_2(n))$ per le Binary Heap contro $\mathcal{O}(\log_k(n))$ per le k-ary Heap). L'operazione di rimozione dell'elemento con minore chiave, tuttavia, aumenta a $\mathcal{O}(k \log_k(n))$ rispetto a $\mathcal{O}(\log_k(n))$ delle Binary Heap. Ma visto che nell'algoritmo di Prim le operazioni di cambio valore delle chiavi sono più comuni delle operazioni di estrazioni del minimo elemento, le k-ary Heap sono comunque più efficienti delle Binary Heap per quell'algoritmo.

8.2 Kruskal con Disjoint-Set e path-compression

9 Conclusioni