Relazione di
Algoritmi Avanzati

Nicola Carlesso - matricola 1237782
Federico Brian - matricola 1243422

A.A. 2019/2020

Indice

1 Introduzione **1**

2 Scelta del linguaggio di programmazione **2**

3 Scelte implementative **3**
3.1 Modello 3
3.2 Algoritmi 4
3.3 Main 5
3.4 Test 6
3.5 Documentazione 6

4 Risultati degli algoritmi **7**
4.1 Specifiche Hardware dei calcolatori utilizzati 7
4.2 Prim 7
4.3 NaiveKruskal 10
4.4 Kruskal 13

5 Conclusioni **16**

Elenco delle figure

1 Ciclo principale dell'algoritmo `Prim`. 4
2 Ciclo principale dell'algoritmo di `NaiveKruskal`. 5
3 Ciclo principale dell'algoritmo `Kruskal`. 5
4 Performance dell'algoritmo Prim sui due sistemi. 7
5 Performance dell'algoritmo Naive Kruskal sui due sistemi. 10
6 Performance dell'algoritmo Kruskal sui due sistemi. 13
7 Performance dei tre algoritmi a confronto nella macchina di Nicola Carlesso. 16
8 Performance dei tre algoritmi a confronto nella macchina di Federico Brian. 16

Elenco delle tabelle

1 Specifiche dei calcolatori utilizzati. Fonte: `http://intel.com`. 7
2 Risultati dell'algoritmo di Prim 10
3 Risultati dell'algoritmo di Naive Kruskal 12
4 Risultati dell'algoritmo di Kruskal 15

1 Introduzione

Il presente documento descrive le scelte architetturali ed implementative del primo elaborato di laboratorio del corso di Algoritmi Avanzati. Di seguito, verrà offerta una panoramica sul lavoro svolto dagli studenti Nicola Carlesso e Federico Brian, riguardante lo studio ed il confronto dei tre diversi algoritmi visti a lezione per il calcolo del *Minimum Spanning Tree*[1]:

* l'algoritmo di Prim (in seguito: `Prim`) che utilizza la struttura dati *Heap* e che, quindi, assegna una complessità asintotica pari a $\mathcal{O}\big(m \log n\big)$;
* l'algoritmo di Kruskal:
  - con un'implementazione *naïve*, chiamato `NaiveKruskal`, in cui si utilizza l'algoritmo *Depht-First Search*[2] per determinare la presenza di cicli all'interno dello stesso. La sua complessità asintotica, quindi, risulta essere $\mathcal{O}\big(mn\big)$;
  - con un'implementazione che utilizza la struttura dati *Disjoint Set* per determinare la presenza o meno di ciclicità, chiamato `Kruskal`. Questo porta la sua complessità asintotica a $\mathcal{O}\big(m \log n\big)$.

Infine, verranno esposti ed adeguatamente discussi i risultati ottenuti.

[1] d'ora in poi MST
[2] d'ora in poi DFS

2 Scelta del linguaggio di programmazione

Per lo svolgimento di questo *assignment* è stato scelto, come linguaggio di programmazione, Java nella sua versione 8. La scelta è derivata, principalmente, da due fattori:

* è stato sia studiato durante il percorso di laurea triennale, sia approfondito autonomamente da entrambi;
* in Java, è possibile utilizzare riferimenti ad oggetti piuttosto che oggetti stessi. Questo ha permesso un'implementazione degli algoritmi che si potrebbe definire "accademica", perché coerente con la complessità dichiarata e semanticamente vicina allo pseudocodice visto a lezione.

Questo ultimo punto ha bisogno di essere sviluppato ulteriormente per risultare chiaro. In una prima implementazione degli algoritmi utilizzando l'approccio *object-oriented* senza l'utilizzo di riferimenti, ci siamo accorti che il codice aggiungeva complessità, anche abbastanza pesanti, rispetto allo pseudocodice illustrato a lezione. Questo accadeva perché inizialmente sono stati utilizzati costruttori di copia profonda che, oltre a raddoppiare l'utilizzo di memoria, aggiungevano una complessità rispetto al numero dei lati, al numero dei nodi oppure ad entrambe.

Ad esempio, in una prima implementazione dell'algoritmo `NaiveKruskal`, ad ogni iterazione del ciclo principale veniva creato un nuovo grafo, copiando il grafo che era stato ottenuto aggiungendo iterativamente un lato alla volta. Il costruttore di copia profonda provvedeva a creare due nuove liste: una di nodi ed una di lati, entrambi aventi le medesime caratteristiche delle liste del grafo da cui sono stati copiati.

Questo ci ha portato a riflettere sul significato dello pseudocodice dei tre diversi algoritmi e li ha guidati verso uno sviluppo di un codice che:

* mantenesse la caratteristica di facile leggibilità propria della programmazione ad oggetti;
* fosse coerente con le complessità dichiarate a lezione.

Questi obiettivi sono stati raggiunti agendo su riferimenti di oggetti piuttosto che su oggetti stessi.

3 Scelte implementative

Come specificato nel precedente paragrafo, nell'implementazione dei tre algoritmi si è 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: in una prima implementazione dell'algoritmo `NaiveKruskal` serviva più di un'ora per trovare il peso del MST dei grafi, ora invece sono necessari "solamente" 17 minuti circa[3].

Benché il codice sia stato adeguatamente commentato[4], di seguito è riportata una *summa* delle caratteristiche di ogni classe implementata che non compaiono nella documentazione, ripartita per `package`.

3.1 Modello

Le componenti del modello, vale a dire le classi presenti all'interno del `package` chiamato `lab1.model`, comprendono tutte le strutture dati utilizzate nella risoluzione dei tre problemi assegnati.

* `Node`: oltre ai campi `ID` e `Father`, sono presenti campi dati usati solo in alcuni algoritmi:
  - `weight`: attributo usato esclusivamente dall'algoritmo `Prim` che indica il peso minimo del lato che collega il nodo al MST creato iterativamente fino a quel momento dall'algoritmo;
  - `visited`: attributo booleano usato solo dagli algoritmi `Kruskal` e `Naive Kruskal` che può essere *true* se il nodo risulta essere già stato visitato, *false* altrimenti;
  - `adjacencyList`: lista che non contiene i nodi adiacenti al nodo selezionato come ci si potrebbe aspettare, ma contiene i riferimenti ai lati che hanno come estremo il nodo selezionato. È stata fatta tale scelta perché così, accedendo ad un elemento di `adjacencyList`, si reperiscono immediatamente le informazioni dei lati adiacenti ad un nodo. Questo fatto è utile, ad esempio, per l'algoritmo `Prim` e, in caso di bisogno, è possibile reperire il nodo opposto al nodo selezionato chiamando semplicemente la funzione `edge.getOpposite(node)` in tempo costante.
* `Edge`: oltre ai riferimenti dei nodi agli estremi del lato, è presente anche il campo `label`, utilizzato dall'algoritmo DFS per determinare la presenza di ciclicità in un grafo. Il campo `label` può avere due valori possibili:
  - `DISCOVERY_EDGE` se il lato in questione è stato percorso per estendere il grafo con un nuovo nodo, mantenendo la proprietà di essere aciclico;
  - `BACK_EDGE` se, invece, si tratta di un lato non percorso anche se i nodi agli estremi risultano visitati. La presenza di un lato con tale etichetta è considerata la prova della ciclicità dello stesso.
* `Graph`: presenta una lista di nodi ed una lista di lati. Nella costruzione del grafo non v'è alcun controllo sull'inserimento di un lato già inserito, oppure di uno che condivide gli stessi nodi di un altro lato ma con peso diverso, perciò un grafo può avere diversi lati che collegano gli stessi vertici, anche con diversi pesi. È stata fatta tale scelta perché la costruzione del grafo risulta più veloce: si evita un controllo su tutti i lati del grafo quando se ne aggiunge uno, riuscendo a mantenere comunque la correttezza degli algoritmi. È stato altresì implementato l'algoritmo per effettuare la DFS, necessaria per l'algoritmo `NaiveKruskal`;

[3] nella macchina di Federico Brian, le cui specifiche hardware saranno illustrate di seguito

[4] come si può vedere dal `Javadoc`, automaticamente generato e accessibile all'interno della cartella `JavaLab1/doc/`, aprendo il file `index.html` con il browser preferito

* `MinHeap`: questa è una classe *Generics* che gestisce un "minheap", cioè un albero binario con la seguente caratteristica: il nodo radice di un sotto-albero qualsiasi contiene un dato che, secondo una certa relazione d'ordine, è minore o uguale rispetto al suo sotto-albero destro e sinistro. Questa relazione d'ordine può essere definita, generalmente, in due modi:

  1. staticamente, utilizzando l'*override* del metodo `compareTo`: una volta definito, non è più possibile modificarlo senza apporre modifiche al codice precedentemente scritto;
  2. dinamicamente, definendo un oggetto derivato dalla classe `Comparator<T>` e avendo la possibilità di intercambiare il criterio di ordinamento del minheap senza dover modificare la il metodo `compareTo` della classe-parametro `T` di `MinHeap`.

  Noi abbiamo deciso di implementare una relazione d'ordine statica, sovrascrivendo cioè il metodo `compareTo`, poiché la relazione che ordina gli oggetti all'interno del minheap non cambia. L'unico criterio che è usato per ordinare gli elementi è, infatti, il peso dei nodi o dei lati. Per motivazioni di completezza, abbiamo ritenuto opportuno implementare anche la modalità dinamica di ordinamento della classe `MinHeap`. Quest'ultima sceglierà l'ordinamento dinamico se, al momento dell'istanziazione, è stato fornito al costruttore un `Comparator<T>` adeguato, altrimenti sceglierà l'ordinamento statico.

  Al mero scopo didattico, sono state create le classi `SortNodesByWeight` e `SortEdgesByWeight`, contrassegnate con l'*annotation* `@deprecated` ed utilizzate per le classi di test.

* `DisjointSet`: questa struttura dati gestisce partizioni di oggetti, rappresentati con un numero intero che li identifica. Ogni oggetto può stare in una sola delle partizioni degli insiemi disgiunti presenti. La struttura dati è utilizzata all'interno dell'algoritmo `Kruskal` e, come si potrà vedere, porta notevoli vantaggi dal punto di vista del *computational time*, seppur con la stessa complessità asintotica di `Prim`. Questo argomento sarà trattato nella sezione 5, dedicata alle conclusioni.

3.2 Algoritmi

Il package `lab1.algorithm` contiene un'unica classe, `MinimumSpanningTreeFinding`, che permette di trovare i MST utilizzando gli algoritmi `Prim`, `NaiveKruskal` e `Kruskal`.

* `Prim`: l'algoritmo risulta essere molto leggibile, da come si può evincere dal seguente *snippet* che riporta il codice del ciclo principale:

```java
int cost = 0;
//O(n(3log n)) = O(nlog n)
while(!Q.isEmpty()) {//O(n)
    Node lightNode = Q.extractMin();//O(log n)
    lightNode.setVisited(true);
    cost += lightNode.getWeight();

    for(Edge edge : lightNode.getAdjacencyList()){//O(m)
        Node opposite = edge.getOpposite(lightNode);
        if(!opposite.isVisited()
                && edge.getWeight().compareTo(opposite.getWeight()) < 0) {
            Q.remove(opposite);//O(log n)
            opposite.setFather(lightNode);
            opposite.setWeight(edge.getWeight());
            Q.insert(opposite);//O(log n)
        }
    }
}
```

Figura 1: Ciclo principale dell'algoritmo `Prim`.

Dove `Q` è il minheap contenente i nodi, ordinati in modo crescente rispetto al valore del loro campo `weight`;

* `NaiveKruskal`: il codice del ciclo principale dell'algoritmo è così formulato:

```
int cost = 0;
//O(mn)
for (Edge edge : edges) {//O(m)
    Node node1 = A.getNodeByID(edge.getNode1().getID()); //O(1)
    Node node2 = A.getNodeByID(edge.getNode2().getID()); //O(1)
    Edge edgeToInsert = new Edge(node1, node2, edge.getWeight());
    //O(1)
    A.addEdge(edgeToInsert);
    //O(n+m)
    if(!A.hasCycle())
        cost += edge.getWeight();
    else{
        /*removes the edge that caused a cycle from the graph
         * and updates the nodes' adjacency lists
         */
        A.getEdges().remove(A.getEdges().size() - 1);
        node1.getAdjacencyList().remove(node1.getAdjacencyList().size() - 1);
        if(!node1.getID().equals(node2.getID()))
            node2.getAdjacencyList().remove(node2.getAdjacencyList().size() - 1);
    }
}
```

Figura 2: Ciclo principale dell'algoritmo di `NaiveKruskal`.

Dove `edges` rappresenta la lista di riferimenti ai lati del grafo, ordinata in modo crescente rispetto al valore dell'attributo `weight` ed `A` è il grafo costruito iterativamente aggiungendo il lato con peso minore estratto da `Q`;

* `Kruskal`: il codice del ciclo principale dell'algoritmo è così formulato:

```
for(Edge edge : edges){//O(m)
    if(ds.find(edge.getNode1().getID() - 1) !=
            ds.find(edge.getNode2().getID() - 1)){//O(log n)
        Edge edgetmp = new Edge(A.getNodeByID(edge.getNode1().getID()),
                A.getNodeByID(edge.getNode2().getID()),
                    edge.getWeight());

        A.addEdge(edgetmp);

        ds.union(edge.getNode1().getID() - 1,
                edge.getNode2().getID() - 1);//O(log n)

        cost += edge.getWeight();
    }
}
```

Figura 3: Ciclo principale dell'algoritmo `Kruskal`.

Dove `cost` è un intero che rappresenta il costo del MST ed `A` è il grafo costruito iterativamente aggiungendo il lato con peso minore.

3.3 Main

Il package `lab1.main` contiene la classe `Main`, responsabile dell'esecuzione degli algoritmi. All'interno vi sono tre funzioni:

* la funzione `Main` che fa partire il calcolo oppure il test del costo del MST secondo l'algoritmo desiderato;
* la funzione `compute` che si occupa di calcolare il costo del MST di ogni grafo presente nel dataset, di salvarlo in un file di testo e di proseguire al test dello stesso.
  Per utilizzare/testare i tre diversi algoritmi, inserire all'interno della funzione `compute` una tra le seguenti stringhe:
  - `prim` per l'algoritmo di Prim con heap;
  - `naivekruskal` per l'algoritmo di Kruskal con la sua implementazione *naïve*;
  - `kruskal` per l'algoritmo di Kruskal con la sua implementazione con disjoint set.
* la funzione `test`, chiamata passando il nome dell'algoritmo desiderato con le stesse *keyword* presentate qui sopra, prosegue con il test sui cost dei MST riportati nel file di testo dalla funzione `compute`.

3.4 Test

Il package `lab1.test` contiene due classi:

* `TestAlgorithm` il cui scopo è di testare la bontà delle soluzioni ritornate con i tre algoritmi sviluppati;
* `TestMinHeap` che è servita per testare il funzionamento della classe `MinHeap` mettendola a confronto con la struttura dati `PriorityQueue`.

È bene sottolineare il fatto che i test fanno uso di `assert`, che occorre abilitare con l'opzione `-ea`.

3.5 Documentazione

Il codice, opportunamente commentato, possiede una documentazione auto-generata con la funzionalità `javadoc`: la si può consultare accedendo alla directory `JavaProject/doc` ed aprendo il file `index.html` con il proprio browser preferito.

4 Risultati degli algoritmi

Questa sezione risponderà alla Domanda 1: verranno riportati i grafici del tempo impiegato in funzione della dimensione del grafo, le performance ed il costo dei MST calcolati dagli algoritmi `Prim`, `Kruskal` e `NaiveKruskal`.

4.1 Specifiche Hardware dei calcolatori utilizzati

Dato il considerevole divario di prestazioni ottenute dalle due diverse macchine, abbiamo ritenuto opportuno riportare le differenti componenti hardware su cui abbiamo lavorato.

Caratteristica	**PC di Nicola**	**PC di Federico**
Nome processore	Intel i5-7300HQ	Intel i7-8750H
Numero core	4	6
Numero thread	4	12
Range velocità di clock [GHz]	2.50 - 3.50	2.20 - 4.10
Dimensione cache L1 [KiB]	256	384
Dimensione cache L2 [MiB]	1	1.5
Dimensione cache L3 [MiB]	6	9
Dimensione RAM [GiB]	8	31.2

Tabella 1: Specifiche dei calcolatori utilizzati.
Fonte: http://intel.com.

4.2 Prim

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

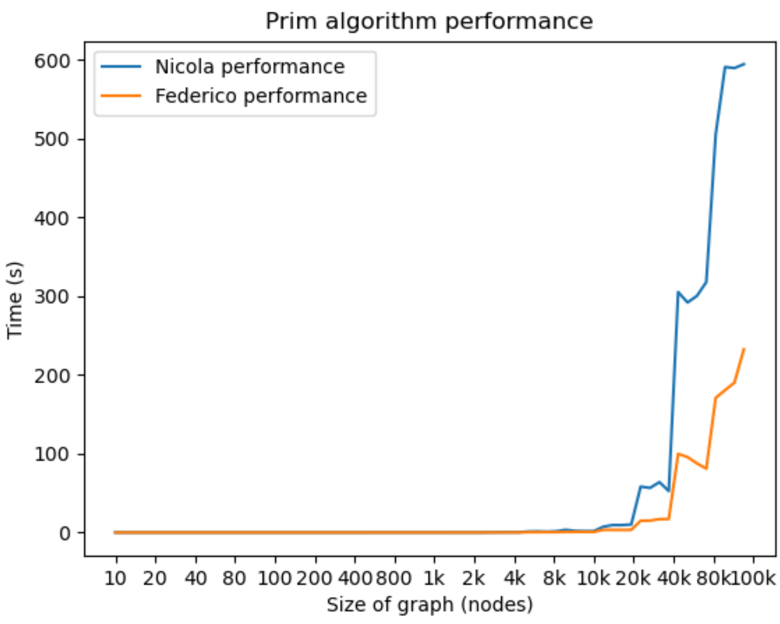

Figura 4: Performance dell'algoritmo Prim sui due sistemi.

L'algoritmo è decisamente performante per grafi fino a 10K nodi, successivamente inizia ad essere relativamente lento per grafi da 80K nodi, impiegando ~1 minuto e mezzo, fino ad arrivare a grafi con 100K nodi impiegando ~4 minuti, nel caso della macchina di Federico Brian.

Di seguito sono riportati il tempo computazionale ed i pesi dei MST calcolati.

N.	Graph Size (nodes)	Time Federico (s)	Time Nicola (s)	MST cost
1	10	0.001516935	0.005691	29316
2	10	0.000139756	0.0002781	2126
3	10	0.000149669	0.0002143	-44765
4	10	0.000115304	0.0001572	20360
5	20	0.000201304	0.002932	-32021
6	20	0.000148581	0.0002687	18596
7	20	0.000130301	0.0005665	-42560
8	20	9.9989e-05	0.000397	-37205
9	40	0.000255545	0.0004462	-122078
10	40	0.000227131	0.001578	-37021
11	40	0.000223172	0.0012624	-79570
12	40	0.000190513	0.0004704	-79741
13	80	0.000461337	0.0022566	-139926
14	80	0.000358958	0.0004229	-211345
15	80	0.000312896	0.0004965	-110571
16	80	0.000331155	0.0055763	-233320
17	100	0.000222887	0.0003762	-141960
18	100	0.000186166	0.0003986	-271743
19	100	0.00020321	0.0053806	-288906
20	100	0.000191939	0.0003545	-232178
21	200	0.000556103	0.0008466	-510185
22	200	0.000539168	0.0012841	-515136
23	200	0.000626964	0.0007046	-444357
24	200	0.000466611	0.0007651	-393278
25	400	0.001343472	0.0081984	-1122919
26	400	0.001259603	0.0062876	-788168
27	400	0.001300368	0.0024222	-895704
28	400	0.001263466	0.0022616	-733645
29	800	0.004210081	0.0156534	-1541291
30	800	0.004169337	0.0170512	-1578294
31	800	0.004372859	0.0111666	-1675534

Continua nella pagina seguente

Tabella 2 – *continuazione della pagina precedente*

N.	Graph Size (nodes)	Time Federico (s)	Time Nicola (s)	MST cost
32	800	0.004226525	0.012685	-1652119
33	1k	0.006369881	0.0271609	-2091110
34	1k	0.00644721	0.0261404	-1934208
35	1k	0.006475871	0.0232982	-2229428
36	1k	0.006468983	0.0134004	-2359192
37	2k	0.025558904	0.0526213	-4811598
38	2k	0.02420279	0.0500484	-4739387
39	2k	0.024796339	0.0477902	-4717250
40	2k	0.025268448	0.0457845	-4537267
41	4k	0.103823489	0.1973502	-8722212
42	4k	0.103343875	0.2438554	-9314968
43	4k	0.112767785	0.2912372	-9845767
44	4k	0.102867612	0.2805648	-8681447
45	8k	0.466010458	1.1943197	-17844628
46	8k	0.457934774	1.4036635	-18800966
47	8k	0.45301853	1.025199	-18741474
48	8k	0.463987517	1.5333734	-18190442
49	10k	0.745876695	3.3735423	-22086729
50	10k	0.748228046	1.9356491	-22338561
51	10k	0.752410272	1.6555519	-22581384
52	10k	0.741559471	1.7339627	-22606313
53	20k	3.366825299	7.2241058	-45978687
54	20k	3.317811526	9.442905599	-45195405
55	20k	3.31394422	9.409197101	-47854708
56	20k	3.35028821	10.1661279	-46420311
57	40k	14.907811327	58.2513275	-92003321
58	40k	15.069201141	56.572616	-94397064
59	40k	16.985234135	63.9437065	-88783643
60	40k	17.136569027	52.5889188	-93017025
61	80k	99.832574806	305.1820434	-186834082
62	80k	95.712610549	292.0056872	-185997521
63	80k	87.760898325	300.2967593	-182065015
64	80k	81.084580784	317.9336533	-180803872
65	100k	170.88136533	505.2773482	-230698391

Continua nella pagina seguente

Tabella 2 – *continuazione della pagina precedente*

N.	Graph Size (nodes)	Time Federico (s)	Time Nicola (s)	MST cost
66	100k	180.640790693	590.8584786	-230168572
67	100k	190.185899105	589.5148767	-231393935
68	100k	232.491690129	594.4183702	-231011693

Tabella 2: Risultati dell'algoritmo di Prim

4.3 NaiveKruskal

Anche per questo algoritmo non abbiamo fatto variazioni rispetto all'implementazione studiata a lezione: la complessità finale, infatti, risulta essere $\mathcal{O}(mn)$. D'altro canto, la funzione presente nel ciclo principale dell'algoritmo `graph.hasCycle()`, che controlla se nel grafo selezionato è presente un ciclo, ha complessità $O(m+n)$. Questo potrebbe portare a pensare che, dunque, l'algoritmo abbia una complessità $\mathcal{O}(m(m+n))$. Tale affermazione risulta essere falsa, in quanto la funzione viene invocata solo dal MST che sappiamo avere la seguente proprietà: $m = n - 1$, con $m = |E|$ il numero dei lati ed $n = |V|$ il numero dei nodi. Dunque, la complessità dell'algoritmo risulta essere proprio $\mathcal{O}(mn)$.

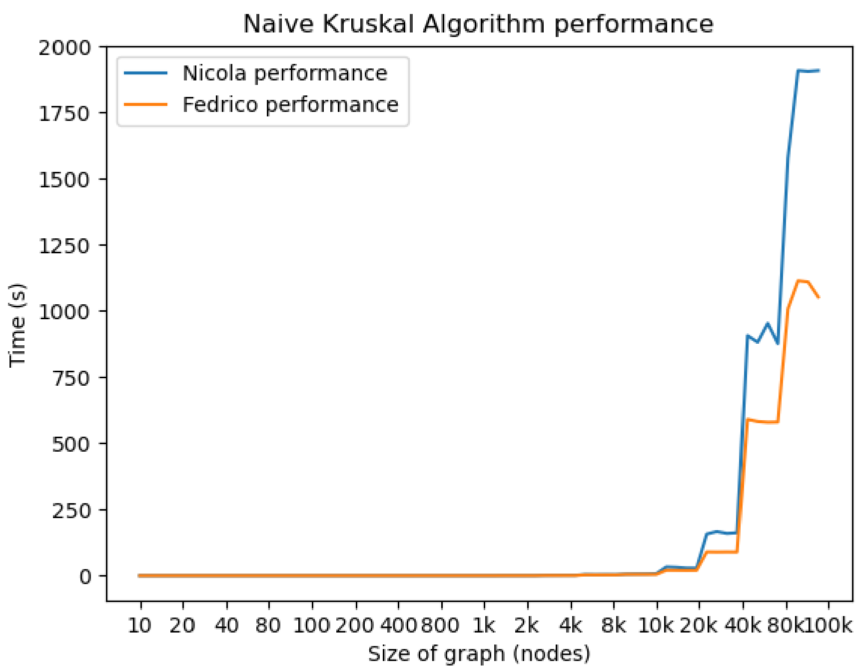

Figura 5: Performance dell'algoritmo Naive Kruskal sui due sistemi.

`NaiveKruskal` è molto efficiente con grafi fino a 4K nodi, mentre già a 40K nodi inizia a mostrare rallentamenti, richiedendo un tempo di ~1 minuto e mezzo, fino ad un tempo di ~17 minuti e mezzo per grafi di 100K nodi[5]. Di seguito sono riportati il tempo computazionale ed i pesi dei MST calcolati.

[5] I tempi fanno riferimento all'algoritmo eseguito nella macchina Federico Brian

N.	Graph Size (nodes)	Time Federico (s)	Time Nicola (s)	MST cost
1	10	0.000560029	0.0010596	29316
2	10	8.5044e-05	0.0002246	2126
3	10	0.000109528	0.0001755	-44765
4	10	7.0003e-05	0.0001161	20360
5	20	0.000217742	0.0003615	-32021
6	20	0.000205306	0.0002788	18596
7	20	0.000251243	0.0002375	-42560
8	20	0.000140594	0.0002041	-37205
9	40	0.000530655	0.0019728	-122078
10	40	0.000425697	0.0003499	-37021
11	40	0.00024775	0.0003325	-79570
12	40	0.000196186	0.0002773	-79741
13	80	0.000737096	0.0014311	-139926
14	80	0.000802938	0.0008544	-211345
15	80	0.000663801	0.0009532	-110571
16	80	0.00075189	0.0020361	-233320
17	100	0.001142012	0.0022148	-141960
18	100	0.001048694	0.001367	-271743
19	100	0.00106485	0.0011847	-288906
20	100	0.001062188	0.0008953	-232178
21	200	0.003979332	0.0022246	-510185
22	200	0.00237902	0.0018707	-515136
23	200	0.002511942	0.0019732	-444357
24	200	0.002382755	0.0025881	-393278
25	400	0.008904292	0.0069684	-1122919
26	400	0.004054465	0.0055755	-788168
27	400	0.004215928	0.0089353	-895704
28	400	0.004001736	0.0060748	-733645
29	800	0.017189365	0.0252481	-1541291
30	800	0.01754511	0.0258438	-1578294
31	800	0.017843113	0.0269498	-1675534
32	800	0.016382689	0.0250428	-1652119
33	1k	0.02666359	0.0396759	-2091110
34	1k	0.026889521	0.0365509	-1934208
35	1k	0.027331791	0.0376809	-2229428

Continua nella pagina seguente

Tabella 3 – *continuazione della pagina precedente*				
N.	**Graph Size (nodes)**	**Time Federico (s)**	**Time Nicola (s)**	**MST cost**
36	1k	0.028779077	0.040684	-2359192
37	2k	0.141345103	0.1869853	-4811598
38	2k	0.133419127	0.1927233	-4739387
39	2k	0.138542937	0.1956191	-4717250
40	2k	0.135108372	0.2264677	-4537267
41	4k	0.707476168	0.9198784	-8722212
42	4k	0.689289844	1.0673374	-9314968
43	4k	0.697196581	1.176725	-9845767
44	4k	0.700634409	1.177549	-8681447
45	8k	3.102323985	4.3709961	-17844628
46	8k	3.096770812	3.8505744	-18800966
47	8k	3.081142442	4.152138	-18741474
48	8k	3.137697114	3.9908528	-18190442
49	10k	4.896234627	5.846408	-22086729
50	10k	4.913670002	6.316151	-22338561
51	10k	4.940323414	6.526546	-22581384
52	10k	4.935964403	7.4906781	-22606313
53	20k	20.618231825	33.2681333	-45978687
54	20k	20.443345376	31.7596484	-45195405
55	20k	20.155556454	28.8519555	-47854708
56	20k	20.351124712	28.6587404	-46420311
57	40k	89.807105168	157.6931358	-92003321
58	40k	89.372149459	167.1818835	-94397064
59	40k	89.697158306	160.0550749	-88783643
60	40k	89.39603408	162.5589424	-93017025
61	80k	590.700453947	908.1871984	-186834082
62	80k	582.496115085	882.8654037	-185997521
63	80k	579.599313318	954.7231673	-182065015
64	80k	580.37768589	877.5672819	-180803872
65	100k	1007.230509086	1577.2524851	-230698391
66	100k	1114.130091458	1909.1328731	-230168572
67	100k	1109.768834091	1905.8766097	-231393935
68	100k	1052.638226358	1908.9756497	-231011693

Tabella 3: Risultati dell'algoritmo di Naive Kruskal

4.4 Kruskal

Abbiamo implementato l'algoritmo come indicato a lezione senza variazione alcuna. Dunque, la sua complessità risulta essere $\mathcal{O}(m \log n)$.

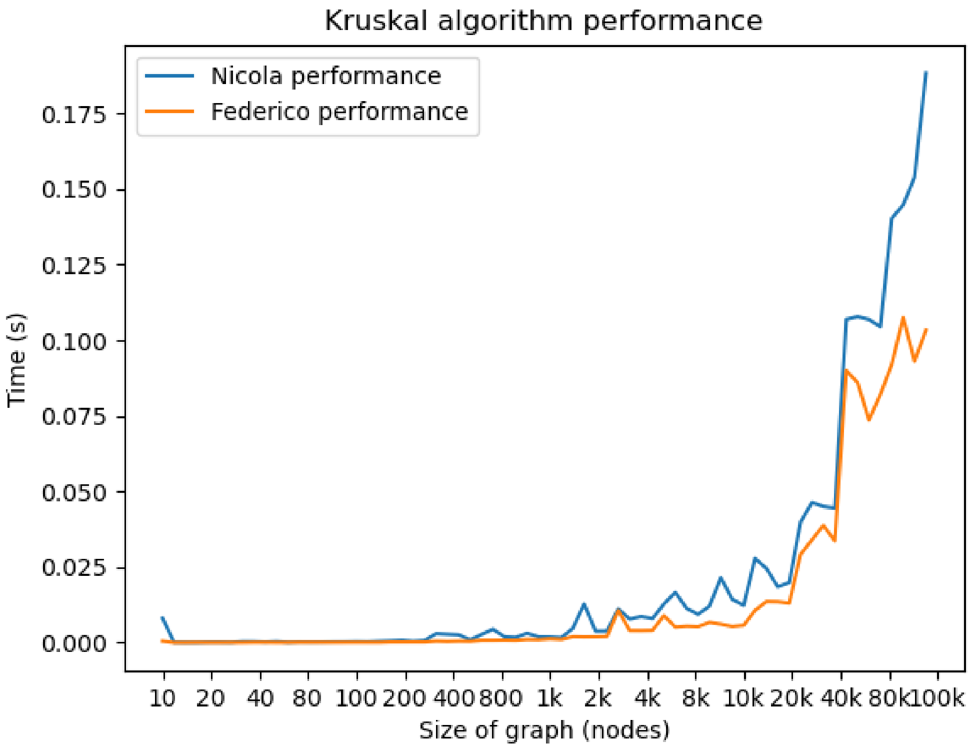

Figura 6: Performance dell'algoritmo Kruskal sui due sistemi.

Di seguito sono riportati il tempo computazionale, assieme ai pesi dei MST calcolati.

N.	Graph Size (nodes)	Time Federico (s)	Time Nicola (s)	MST cost
1	10	0.000475016	0.0080559	29316
2	10	2.4539e-05	0.0001189	2126
3	10	2.4995e-05	8.96e-05	-44765
4	10	6.3777e-05	7.92e-05	20360
5	20	4.2758e-05	0.0001614	-32021
6	20	4.0177e-05	0.0001641	18596
7	20	5.1368e-05	0.0001285	-42560
8	20	5.3807e-05	0.0003505	-37205
9	40	0.000101506	0.0003404	-122078

Continua nella pagina seguente

Tabella 4 – *continuazione della pagina precedente*

N.	**Graph Size (nodes)**	**Time Federico (s)**	**Time Nicola (s)**	**MST cost**
10	40	0.000101048	0.0002097	-37021
11	40	6.0438e-05	0.000361	-79570
12	40	6.1835e-05	0.0001309	-79741
13	80	0.000104485	0.0002288	-139926
14	80	0.000102939	0.0002083	-211345
15	80	0.000101184	0.0002255	-110571
16	80	9.7308e-05	0.0002586	-233320
17	100	0.0001047	0.0002707	-141960
18	100	0.000111879	0.0003329	-271743
19	100	0.000104892	0.000289	-288906
20	100	0.000104441	0.0004452	-232178
21	200	0.000213534	0.0005749	-510185
22	200	0.000213031	0.0007544	-515136
23	200	0.000216838	0.000455	-444357
24	200	0.000213391	0.0007035	-393278
25	400	0.000478752	0.0029743	-1122919
26	400	0.000378236	0.0027706	-788168
27	400	0.000478894	0.0025862	-895704
28	400	0.000426838	0.0009081	-733645
29	800	0.000757029	0.0027124	-1541291
30	800	0.000759682	0.0044353	-1578294
31	800	0.000797339	0.0019433	-1675534
32	800	0.000742282	0.0017496	-1652119
33	1k	0.00101101	0.0031046	-2091110
34	1k	0.000950238	0.0019217	-1934208
35	1k	0.001235389	0.0018778	-2229428
36	1k	0.001024273	0.0016544	-2359192
37	2k	0.001983391	0.0047124	-4811598
38	2k	0.001905485	0.0127673	-4739387
39	2k	0.001940465	0.0038127	-4717250
40	2k	0.002034407	0.0038368	-4537267
41	4k	0.010617613	0.0111621	-8722212
42	4k	0.004021979	0.0077927	-9314968
43	4k	0.004002429	0.0086298	-9845767

Continua nella pagina seguente

Tabella 4 – *continuazione della pagina precedente*

N.	Graph Size (nodes)	Time Federico (s)	Time Nicola (s)	MST cost
44	4k	0.004069054	0.0079643	-8681447
45	8k	0.008941901	0.0127837	-17844628
46	8k	0.005141779	0.016622	-18800966
47	8k	0.005433211	0.0113138	-18741474
48	8k	0.005279905	0.0093715	-18190442
49	10k	0.006703729	0.0120966	-22086729
50	10k	0.006112203	0.0215072	-22338561
51	10k	0.005335733	0.014305	-22581384
52	10k	0.005733082	0.0123635	-22606313
53	20k	0.010668536	0.0278435	-45978687
54	20k	0.013601353	0.0245053	-45195405
55	20k	0.01353099	0.0184613	-47854708
56	20k	0.013051076	0.0198698	-46420311
57	40k	0.029222432	0.0399019	-92003321
58	40k	0.034022169	0.0463569	-94397064
59	40k	0.038760754	0.0450415	-88783643
60	40k	0.033626421	0.0444899	-93017025
61	80k	0.090007024	0.1068958	-186834082
62	80k	0.086002076	0.1078061	-185997521
63	80k	0.073560867	0.106871	-182065015
64	80k	0.082105309	0.1044562	-180803872
65	100k	0.092029677	0.1403172	-230698391
66	100k	0.10750741	0.1447736	-230168572
67	100k	0.093028701	0.1539362	-231393935
68	100k	0.103353746	0.1883876	-231011693

Tabella 4: Risultati dell'algoritmo di Kruskal

5 Conclusioni

In questa sezione risponderemo alla Domanda 2: si metteranno a confronto i risultati ottenuti dai tre algoritmi, accompagnati da un adeguato commento.

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

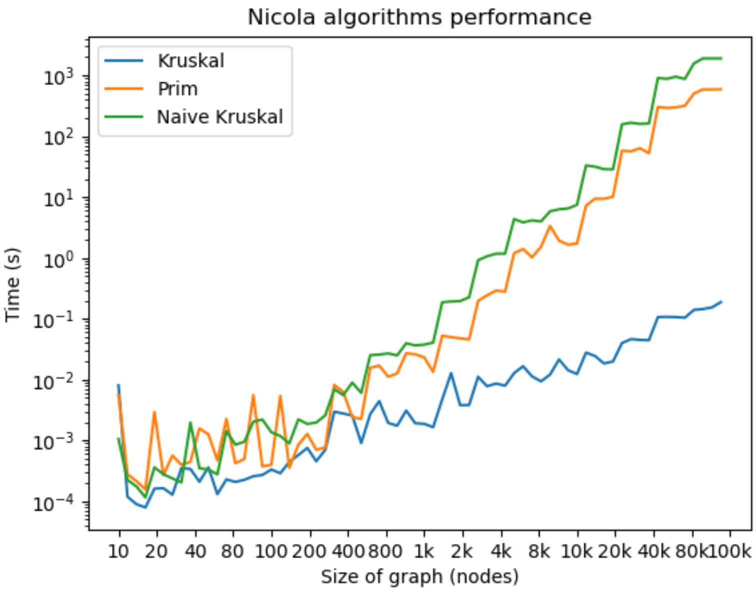

Figura 7: Performance dei tre algoritmi a confronto nella macchina di Nicola Carlesso.

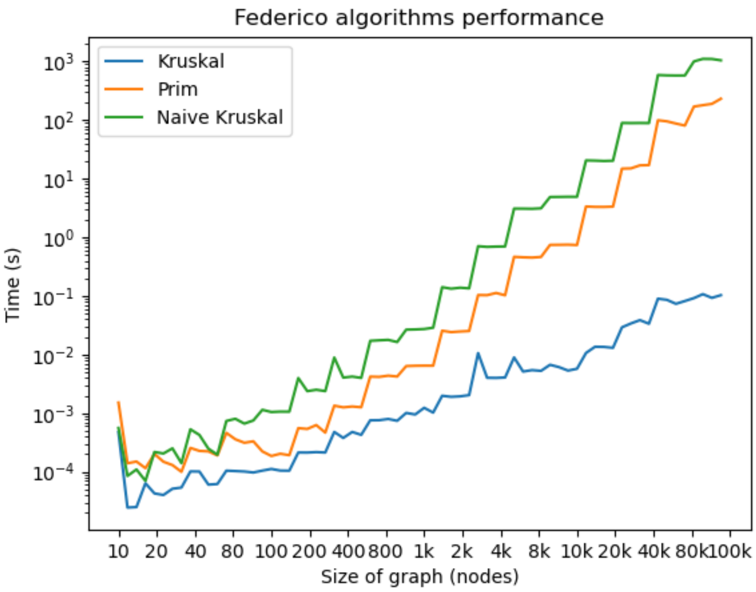

Figura 8: Performance dei tre algoritmi a confronto nella macchina di Federico Brian.

Inizialmente `Prim` e `NaiveKruskal` delineano un andamento simile, tuttavia è possibile osservare come, già a partire dai grafi con 20K nodi, i tempi impiegati da `NaiveKruskal` risultino essere significativamente superiori a quelli di `Prim`. Questo perché, all'aumentare dei nodi, il tempo di esecuzione dell'algoritmo `NaiveKruskal`, con complessità $\mathcal{O}(mn)$, cresce asintoticamente più rapidamente di quello impiegato dall'algoritmo `Prim`, avente complessità $\mathcal{O}(m \log n)$.

D'altra parte, l'algoritmo `Kruskal` riesce sempre ad impiegare meno tempo degli altri algoritmi: è infatti possibile vedere come, anche nei grafi da 100K nodi, impieghi solamente ~0.1-0.2 secondi per calcolare il costo del MST. Questo potrebbe essere considerato un comportamento sospetto, poiché i due algoritmi `Prim` e `Kruskal` impiegano tempi d'esecuzione molto distanti pur avendo la stessa complessità asintotica.
In realtà, il concetto di complessità asintotica è molto diverso dal *computational time*: i due concetti non vanno confusi ma considerati come due entità distinte. Il primo concetto, infatti, assume:

* che ogni tipologia di istruzione abbia il medesimo tempo;
* che questo tempo rimanga costante nel tempo.

Infatti, per il calcolo della complessità, viene assunto che ogni istruzione elementare abbia complessità $\mathcal{O}(1)$. Si tratta, però, di un calcolo puramente teorico che serve per dare l'idea del comportamento asintotico dell'algoritmo all'aumentare della dimensione dell'input.

Questo concetto si scontra con il *computational time*: nella realtà, infatti, ogni istruzione atomica può impiegare tempi molto diversi. Inoltre, non si tiene conto di alcune operazioni necessarie che il processore deve compiere, come ad esempio il *context switch* della cache. Anche la politica di scrittura utilizzata per mantenere la coerenza della memoria può occupare il bus di sistema e, conseguentemente, causare ritardi nei tempi di esecuzione.

Un altro fatto che potrebbe destare qualche perplessità è la sostanziale differenza tra le tempistiche misurare dal calcolatore di Federico rispetto a quelle misurate dal calcolatore di Nicola. Anche qui si può vedere come, in realtà, il *computational time* non sia un indicatore valido della complessità di un algoritmo: i tempi misurati utilizzando un determinato processore sono quasi la metà rispetto ai tempi misurati con un altro processore, a riprova del fatto che complessità asintotica e *computational time* siano due cose completamente diverse.