Corso di Algoritmi Avanzati

Laboratorio 1: Minimum Spanning Tree

10 luglio 2020

Busin Lorenzo	1237580
Tartaggia Nicolò	1237784
Voinea Ciprian	1237294

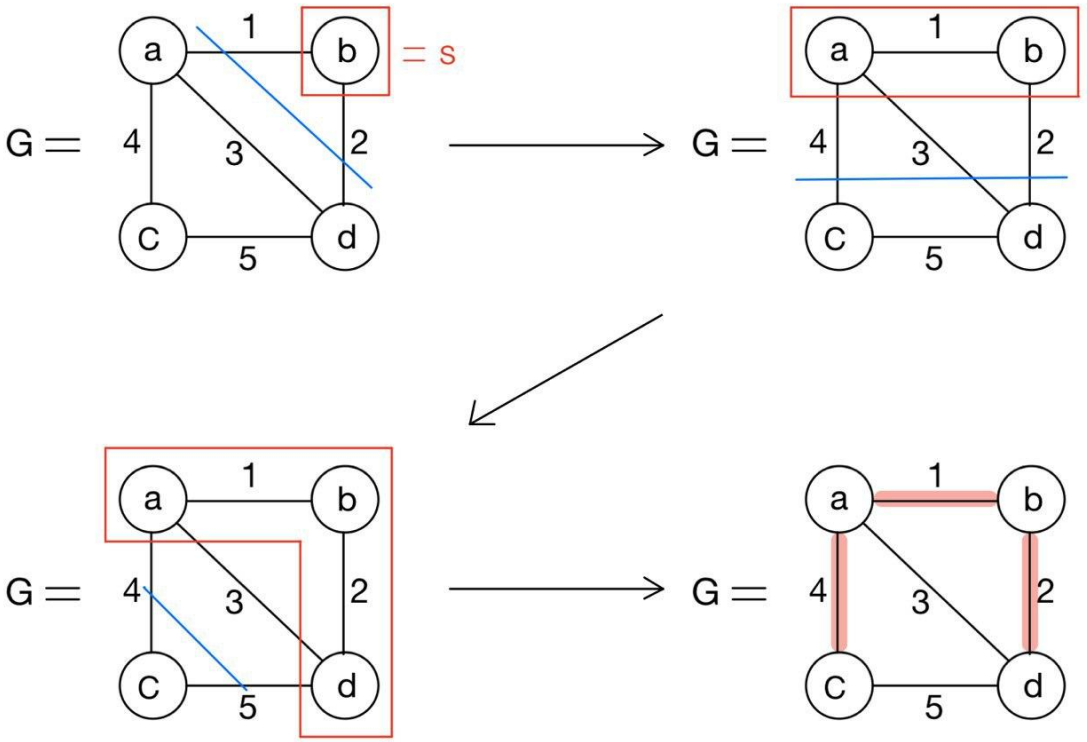
Indice

1 Introduzione

2 Algoritmo di Prim

```
PRIM (G, s)
    X = {s} //set of vertexes included in the MST
    A = ∅ //set of edges included in the MST
    while there is an edge (u, v) with u ∈ X and v ∉ X do
        (u*, v*) = a minimum cost such edge //light edge
        add vertex v* to X
        add edge (u*, v*) to A
    return A
```

L'algoritmo di Prim costruisce un MST a partire da un vertice di iniziale s, aggiungendoci un lato alla volta:



Ad ogni iterazione, si considerano i lati attraversati dal taglio (raffigurato in blu) che separa X dal resto del grafo. Successivamente, di questi viene scelto il lato (u, v) di peso minimo. Infine, il lato selezionato viene aggiunto all'insieme A e il vertice v viene aggiunto all'insieme X.

2.1 Strutture dati

2.1.1 MinHeap

La struttura dati utilizzata per l'implementazione dell'algoritmo è la struttura *min-heap*. Un *MinHeap* è un oggetto composto da un array che può essere considerato come un albero binario quasi completo. Ogni nodo dell'albero corrisponde a un elemento dell'array. Tutti i livelli sono completamente riempiti, tranne eventualmente l'ultimo che può essere riempito da sinistra fino ad un certo punto.
Un array `A` che rappresenta un heap è un oggetto *ArrayHeap*, il quale estende la classe *List* e presenta attributi:

- `A.length`: numero degli elementi dell'array;
- `A.heap-size`: numero degli elementi dell'heap che sono registrati nell'array.

Cioè, anche se ci possono essere dei numeri memorizzati in tutto l'array `A[1 ... A.length]`, soltanto i numeri in `[1 ... A.heap-size]`, dove $0 \leq$ `A.heap-size` $\leq$ `A.length`, sono elementi validi dell'heap.
A partire dall'indice i di un nodo è possibile accedere all'indice del padre, del figlio sinistro e del figlio destro nel seguente modo:

- `parent(i)`: $\lfloor i/2 \rfloor$
- `left(i)`: $2 * i$
- `right(i)`: $2 * i + 1$

Un *min-heap* presenta la cosiddetta proprietà del *min-heap*, secondo la quale, per ogni nodo diverso dalla radice, si ha che `A[parent(i)]` $\leq$ `A[i]`. Di conseguenza il più piccolo elemento in un min-heap è nella radice. Le altre informazioni relative all'oggetto *MinHeap* sono le seguenti:

- `minHeapify(i)`: metodo che permette di mantenere la proprietà del min-heap a partire dall'indice `i`;
- `bubbleUp(i)`: metodo che permette di mantenere la proprietà del min-heap, riposizionando l'elemento di indice `i` dell'array-heap nella posizione corretta, risalendo di padre in padre;
- `extractMin()`: metodo che estrae il minimo dall'array-heap, cioè la radice dell'albero, e ristabilisce la proprietà del min-heap tramite la chiamata a `minHeapify(i)`.

2.1.2 Node

L'oggetto *Node* rappresenta un vertice del grafo. Esso comprende i seguenti campi dati:

- `tag`: intero che identifica un vertice;
- `key`: chiave del vertice, il suo valore di default è `None`;
- `parent`: padre del vertice, il suo valore di default è `None`;

- `isPresent`: variabile booleana che memorizza la presenza di un nodo nell'array che rappresenta l'heap. Il suo valore di default è `True`;
- `index`: indice dell'array del min-heap associato al vertice;
- `adjacencyList`: lista di adiacenza del vertice, il suo valore di default è la lista vuota.

2.1.3 Graph

L'oggetto *Graph* permette di gestire la creazione e la costruzione di un grafo secondo le caratteristiche specificate nel file `.txt` dato in input. Il suo unico campo dati è `nodes`, un dizionario di oggetti di tipo `Node`. Il tipo dizionario utilizzato è `defaultDict` di Python. La scelta di utilizzare questo tipo di dizionario è dovuta al fatto che l'oggetto viene inizializzato con una funzione che fornisce un valore di default nel caso si utilizzi una chiave non esistente. In questo modo non vengono sollevate eccezioni del tipo *"KeyError"*.
Le altre informazioni relative all'oggetto *Graph* sono le seguenti:

- `createNodes(n)`: metodo che instanzia `n` vertici nel dizionario `nodes`;
- `addNode(u, v, cost)`: metodo che aggiorna le liste di adiacenza dei nodi `u` e `v`, inserendo inoltre, in entrambe, il peso dell'arco che li collega;
- `buildGraph(n)`: metodo principale per la costruzione del grafo. Esso prima chiama il metodo `createNodes(n)` e, successivamente, il metodo `addNode(u, v, cost)` per ogni vertice di input.

2.2 Implementazione

La soluzione del problema è stata implementata nel seguente modo:

- Viene inizializzato il grafo G attraverso il metodo `buildGraph()`, al quale viene passato il grafo di input come parametro. La funzione esegue due passaggi in sequenza:
  1. Tramite l'apposito costruttore, vengono inizializzati nel dizionario del grafo un numero di vertici uguale a quanto indicato nella prima riga dell'input. La chiave per accedere ad un vertice è il suo `tag`;
  2. Per ogni tripla ($vertice_1\_arco_i$, $vertice_2\_arco_i$, $peso\_arco_i$) dell'input, vengono aggiornate le liste di adiacenza di entrambi i vertici poiché il grafo è indiretto:

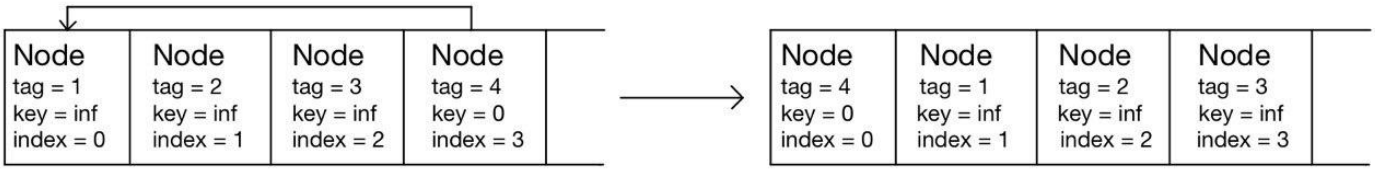
     ```
     nodes[tag_vertice1].adjacencyList.append([nodes[tag_vertice2], peso])
     nodes[tag_vertice2].adjacencyList.append([nodes[tag_vertice1], peso])
     ```

     Ogni lista di adiacenza, quindi, contiene un numero di sottoliste di due elementi pari al numero di vertici adiacenti. I due elementi sono il vertice adiacente e il peso dell'arco che connette i due vertici. In questo modo si riesce ad accedere sia alle informazioni del vertice adiacente sia al peso in maniera immediata.
     L'inizializzazione al punto 1 tutti i possibili vertici del grafo assicura che il vertice che viene aggiunto alla lista di adiacenza di un altro vertice sia un oggetto definito.

Inoltre ogni vertice presente in una lista di adiacenza è un riferimento all'oggetto vero e proprio. In questo modo non vengono effettuate copie inutili;

- Una volta creato il grafo, questo viene fornito come input all'algoritmo `MSTPrim` insieme al nodo di partenza. L'algoritmo esegue i seguenti passi:

  1. Assegna al campo `key` di ogni vertice il valore ∞ `(math.inf)`;
  2. Assegna al campo `key` del nodo di partenza il valore 0;
  3. Viene inizializzata la struttura min-heap a partire dai nodi del grafo. L'inizializzazione tiene conto del nodo di partenza. Infatti, seguendo la struttura del file di input, la lista di vertici che viene fornita come input alla struttura dati è una lista ordinata a partire dal vertice con `tag = 1`. Di conseguenza possono verificarsi due casi:
     - Se il vertice di partenza è il vertice con `tag = 1` quest'ultimo sarà il nodo radice dell'albero min-heap, in quanto il suo campo `key`, utilizzato per confrontare i nodi, sarà il più piccolo. In questo caso l'array-heap sarà costruito utilizzando la lista di input;
     - Se il vertice di partenza ha `tag != 1` l'array-heap viene costruito in modo analogo a prima. Dopodiché il nodo di partenza viene posto in testa all'array-heap e tutti i campi `index` di tutti vertici vengono aggiornati.



Posizionare il nodo di partenza in testa all'array-heap assicura che la proprietà del min-heap venga rispettata. Il compromesso è l'aggiornamento di tutti i campi `index`, eseguibile in tempo lineare nel numero di vertici;

  4. A questo punto comincia la fase iterativa dell'algoritmo:
     (a) Viene estratto il minimo dalla struttura min-heap tramite la procedura `extractMin()`. Per fare ciò essa estrae la radice dell'albero, effettua gli aggiornamenti necessari per mantenere la proprietà del min-heap attraverso il metodo `minHeapify()`, assegna al campo `isPresent` il valore `false` poiché il vertice non farà più parte dell'array-heap ed infine ritorna la radice;

     (b) Per ogni vertice `v` nella lista di adiacenza del vertice appena estratto si controlla che i) appartenga all'array-heap e ii) che il peso dell'arco tra i due sia minore del valore attuale del campo `key` del vertice estratto. In caso positivo, si procede all'aggiornamento dei campi `parent` e `key` di `v`;

     (c) Viene invocato il metodo `bubbleUp()` per mantenere la proprietà del min-heap. Esso colloca nella posizione esatta dell'albero il vertice `v` in quanto il suo campo `key` è stato precedentemente aggiornato.

2.3 Complessità

La complessità dell'algoritmo è $O(m \log(n))$, dove m indica il numero totale di archi e n il numero totale di vertici:

- L'inizializzazione della struttura min-heap, nel caso peggiore, richiede un tempo di esecuzione $O(n)$;
- L'operazione `extractMin()` ha un tempo di esecuzione di $O(\log(n))$ e viene ripetuta n volte all'interno del ciclo `while`. Il costo totale quindi è $O(n \log(n))$;
- Il ciclo `for` viene eseguito $O(m)$ volte, in quanto la somma delle lunghezze delle liste di adiacenza è $2|m|$. All'interno del ciclo `for`, il test che verifica l'appartenenza di un vertice adiacente alla struttura min-heap è eseguito in tempo costante $O(1)$. Infine, l'operazione `bubbleUp()` per il ripristino della proprietà del min-heap è eseguita in tempo $O(\log(n))$. Il costo totale quindi è $O(m \log(n))$;

Sommando questi dati il tempo totale dell'algoritmo è $O(n \log(n) + m \log(n))) = O(m \log(n))$.

3 Algoritmo di Kruskal con implementazione "naive"

```
1 KRUSKAL (G)
2 A = ∅
3 sort edges of G by cost
4 for each edge e, in non decreasing order of cost do
5    if A ∪ {e} is acyclic then
6       A = A ∪ {e}
7 return A
```

Kruskal "naive" è un algoritmo *greedy* che non utilizza particolari strutture dati e che costruisce un MST aggiungendo ad ogni iterazione un nuovo lato di costo minimo all'insieme A, se ciò non comporta il verificarsi di un ciclo.

3.1 Strutture dati

3.1.1 Graph

L'oggetto *Graph* contiene le informazioni relative ad un grafo con:

- `n`: campo dati che indica numero di vertici;
- `Le`: campo dati che contiene la lista degli archi;
- `adj`: campo dati che contiene le liste di adiacenza tramite un dizionario;
- `addEdge(u, v)`: metodo che dati due vertici in input ne aggiunge il collegamento nelle liste di adiacenza di entrambi i vertici;
- `isConnected(s, t)`: metodo che dati due vertici in input ritorna `True` se essi sono connessi o `False` se non lo sono;
- `buildGraph(input)`: metodo che dato in input il file che contiene la struttura di un grafo suddiviso per linee costruisce un oggetto `Graph()`.

3.1.2 Edge

L'oggetto *Edge* contiene le informazioni relative ad un arco:

- `v1`: vertice di partenza;
- `v2`: vertice di arrivo;
- `weight`: peso dell'arco.

N.B. Non vi è una vera e propria distinzione tra vertice di partenza e vertice di arrivo dato che il grafo non è orientato.

3.2 Implementazione

La soluzione del problema è stata implementata nel seguente modo:

- Viene inizializzato il grafo G attraverso il costruttore `Graph()`, al quale viene passato in input il numero di vertici estratti dalla prima riga del file di input in formato `.txt`;
- Viene costruito il grafo G tramite il metodo `buildGraph()`, al quale viene passato il grafo di input come parametro. La funzione, per ogni tripla ($vertice_1\_arco_i$, $vertice_2\_arco_i$, $peso\_arco_i$) in input, aggiorna le liste di adiacenza di entrambi i vertici attraverso il metodo `addEdge()` ed inserisce l'oggetto arco nella lista degli archi;
- Una volta creato il grafo, esso viene fornito come input all'algoritmo `Kruskal` che esegue i seguenti passi:
  1. Viene inizializzato il grafo A con lo stesso numero di vertici del grafo di input. Esso ha lo scopo di tenere traccia degli archi aggiunti al MST;
  2. Viene ordinata la lista contenente gli archi del grafo di input in maniera crescente in base al peso attraverso l'algoritmo *MergeSort*;
  3. Per ogni arco viene verificato, attraverso il metodo `isConnected()` della classe `Graph`, che la sua aggiunta al grafo A non porti alla creazione di un ciclo all'interno del medesimo grafo e, in caso positivo, l'arco viene aggiunto all'insieme A. In questo caso l'algoritmo non verifica la non presenza di un ciclo, ma verifica che non ci sia già un cammino che collega i due vertici in input, in quanto se così fosse l'aggiunta di un ulteriore cammino andrebbe sicuramente a costituire un ciclo in quel grafo. Il metodo `isConnected()` è stato implementato con una variante iterativa di una *BFS* che, dati in input due nodi s e t, inizialmente considera tutti i vertici, a parte quello di partenza s, come non visitati e, tramite l'utilizzo di una coda, vengono man mano visitati i nodi adiacenti fino ad arrivare al nodo t, se questo è raggiungibile da s. Per ottimizzare l'algoritmo è possibile fermare il ciclo quando A contiene $n-1$ lati, dove n indica il numero di vertici.

3.3 Complessità

La complessità dell'algoritmo è $O(mn)$, dove m indica il numero totale di archi e n il numero totale di vertici:

- L'ordinamento degli archi in base al peso viene eseguito con l'algoritmo MergeSort che ha complessità $O(m\log(m))$;
- Il ciclo `for` viene eseguito in $O(m)$ iterazioni, ma è possibile fermare il ciclo quando l'insieme A raggiunge $n-1$ lati. Ad ogni iterazione viene chiamata la funzione `isConnected()` per verificare se il lato in esame può essere aggiunto alla soluzione che ha complessità lineare nel numero di vertici $O(n)$.

4 Algoritmo di Kruskal con Union-Find

```
KRUSKAL(G):
    A = ∅
    For each vertex v ∈ G.V:
        MAKE–SET(v)
    sort edges of G by cost
    for each edge e, in non decreasing order of cost do
        if FIND–SET(u) ≠ FIND–SET(v):
            A = A ∪ {(u, v)}
            UNION(u, v)
    return A
```

Come per la sua versione naive, l'implementazione di Kruskal tramite l'uso della struttura dati Union Find è un algoritmo greedy che costruisce un MST aggiungendo ad ogni iterazione un nuovo lato di costo minimo all'insieme A se i vertici di questo non fanno già parte dello stesso set.

4.1 Strutture dati

4.1.1 Graph ed Edge

L'implementazione di queste due strutture dati sono equivalenti a quelle utilizzate in 3.1.1 e 3.1.2 per l'algoritmo di Kruskal naive.

4.1.2 Union Find

La struttura dati *Union Find* (o *Disjoint-Set*) mantiene una collezione di insiemi dinamici disgiunti (chiamati anche *componenti connesse*) $S = \{s_1, s_2, ..., s_n\}$. Viene implementata tramite un array e visualizzabile come un insieme di alberi diretti, in quanto c'è una relazione padre-figlio dove ciascun nodo figlio punta al proprio padre.
Per ogni elemento di questo array viene salvato un campo `parent` che, al momento dell'inizializzazione, contiene l'indice dell'array dell'oggetto `x`.
Per ciascun insieme dell'oggetto Union Find c'è un elemento che lo rappresenta, detto anche *root*. La profondità di questi insiemi è pari al numero di elementi compresi tra l'ultimo nodo figlio e la root.
Le operazioni permesse sugli elementi di questa struttura dati sono:

- `Initialize(x)` o `Make-Set(x)`: metodo che, dato un insieme di oggetti $X = \{x_1, x_2, ..., x_n\}$, crea una struttura dati Union Find nella quale per ciascun elemento $x_i \in X$ viene creato un set con root x_i. Questa operazione ha complessità lineare sul numero di elementi in X, quindi $O(n)$;
- `Find(x)`: metodo che ritorna la radice dell'insieme in cui è contenuto l'elemento x passato in input, risalendo di padre in padre fino a trovare un *self loop*, ovvero trova

un elemento che punta a se stesso. La sua complessità è proporzionale alla profondità della radice di x, quindi nel caso peggiore è $O(n)$;

- `Union(x, y)`: metodo che, prende in input due oggetti e, se questi fanno parte di due insiemi disgiunti diversi, unisce le due componenti connesse che li contengono facendo puntare la radice dell'albero con profondità minore a quella del secondo albero, minimizzando quindi la profondità totale.
  La complessità di questa operazione è $O(\log n)$ e viene dominata dalla complessità della precedente funzione `Find`, che viene richiamata al suo interno.

4.2 Implementazione

La soluzione al problema è stata implementata nel seguente modo:

- viene inizializzato il grafo G attraverso il costruttore `Graph()`;
- viene costruito il grafo G tramite il metodo `buildGraph()`, al quale viene passato il grafo di input come parametro. La funzione, per ogni tripla ($vertice_1\_arco_i$, $vertice_2\_arco_i$, $peso\_arco_i$) dell'input, aggiorna le liste di adiacenza di entrambi i vertici attraverso il metodo `addEdge()` ed inserisce l'oggetto arco nella lista degli archi;
- una volta creato il grafo, esso viene fornito come input all'algoritmo `KruskalUF` che esegue i seguenti passi:
  1. viene definito A un insieme vuoto, a questo vengono aggiunti ad ogni iterazione i lati che andranno a costituire la soluzione;
  2. viene inizializzata la struttura di insiemi disgiunti U passando l'insieme di nodi del grafo al suo costruttore, questo crea un insieme per ciascun nodo del grafo;
  3. utilizzando l'algoritmo MergeSort, viene ordinata la lista contenente gli archi del grafo di input in maniera crescente;
  4. viene iterata la lista di lati ordinata e per ciascuno di questi viene controllato se gli insiemi dei suoi nodi coincidono confrondandone la root. Se queste sono diverse significa i nodi sono in due componenti connesse distinte, in tal caso viene aggiungo il lato al grafo di soluzione A aggiornando la situazione delle componenti connesse.

4.3 Complessità

Per calcolare la complessità totale dell'algoritmo bisogna tenere in considerazione:

- il costo $O(n)$ dell'inizializzazione della struttura dati Union Find;
- il MergeSort con complessità $O(m\log(m))$, dove m indica il numero di archi;
- le operazioni del ciclo, effettuate al massimo m volte:
  - le due operazioni di `Find` ($2m$ in totale) con complessità $O(m\log(n))$

- le $n-1$ operazioni di `Union` (eseguita ogni volta che viene aggiunto un nuovo lato) con complessità $O(n \log(n))$
- l'aggiornamento di A ha un costo lineare, quindi con complessità $O(m)$

La complessità dell'intero algoritmo è quindi $O(m \log(n))$.

5 Risultati

5.1 Tabella con MST calcolati

Input file	num_vertici	num_archi	MST
input_random_01_10.txt	10	9	29316
input_random_02_10.txt	10	13	2126
input_random_03_10.txt	10	14	-44765
input_random_04_10.txt	10	11	20360
input_random_05_20.txt	20	25	-32021
input_random_06_20.txt	20	25	18596
input_random_07_20.txt	20	29	-42560
input_random_08_20.txt	20	26	-37205
input_random_09_40.txt	40	57	-122078
input_random_10_40.txt	40	51	-37021
input_random_11_40.txt	40	50	-79570
input_random_12_40.txt	40	52	-79741
input_random_13_80.txt	80	108	-139926
input_random_14_80.txt	80	101	-211345
input_random_15_80.txt	80	104	-110571
input_random_16_80.txt	80	115	-233320
input_random_17_100.txt	100	137	-141960
input_random_18_100.txt	100	129	-271743
input_random_19_100.txt	100	137	-288906
input_random_20_100.txt	100	133	-232178
input_random_21_200.txt	200	268	-510185
input_random_22_200.txt	200	269	-515136
input_random_23_200.txt	200	269	-444357
input_random_24_200.txt	200	267	-393278
input_random_25_400.txt	400	541	-1122919

Input file	num_vertici	num_archi	MST
input_random_26_400.txt	400	518	-788168
input_random_27_400.txt	400	539	-895704
input_random_28_400.txt	400	526	-733645
input_random_29_800.txt	800	1026	-1541291
input_random_30_800.txt	800	1059	-1578294
input_random_31_800.txt	800	1078	-1675534
input_random_32_800.txt	800	1050	-1652119
input_random_33_1000.txt	1000	1301	-2091110
input_random_34_1000.txt	1000	1313	-1934208
input_random_35_1000.txt	1000	1328	-2229428
input_random_36_1000.txt	1000	1345	-2359192
input_random_37_2000.txt	2000	2699	-4811598
input_random_38_2000.txt	2000	2654	-4739387
input_random_39_2000.txt	2000	2652	-4717250
input_random_40_2000.txt	2000	2677	-4537267
input_random_41_4000.txt	4000	5361	-8722212
input_random_42_4000.txt	4000	5316	-9314968
input_random_43_4000.txt	4000	5340	-9845767
input_random_44_4000.txt	4000	5369	-8681447
input_random_45_8000.txt	8000	10706	-17844628
input_random_46_8000.txt	8000	10672	-18800966
input_random_47_8000.txt	8000	10662	-18741474
input_random_48_8000.txt	8000	10758	-18190442
input_random_49_10000.txt	10000	13302	-22086729
input_random_50_10000.txt	10000	13342	-22338561
input_random_51_10000.txt	10000	13287	-22581384
input_random_52_10000.txt	10000	13287	-22606313
input_random_53_20000.txt	20000	26671	-45978687

Input file	num_vertici	num_archi	MST
input_random_54_20000.txt	20000	26826	-45195405
input_random_55_20000.txt	20000	26674	-47854708
input_random_56_20000.txt	20000	26671	-46420311
input_random_57_40000.txt	40000	53415	-92003321
input_random_58_40000.txt	40000	53447	-94397064
input_random_59_40000.txt	40000	53243	-88783643
input_random_60_40000.txt	40000	53319	-93017025
input_random_61_80000.txt	80000	106587	-186834082
input_random_62_80000.txt	80000	106634	-185997521
input_random_63_80000.txt	80000	106587	-182065015
input_random_64_80000.txt	80000	106555	-180803872
input_random_65_100000.txt	100000	133395	-230698391
input_random_66_100000.txt	100000	133214	-230168572
input_random_67_100000.txt	100000	133525	-231393935
input_random_68_100000.txt	100000	133463	-231011693

5.2 Grafico delle performance dell'algoritmo di Prim

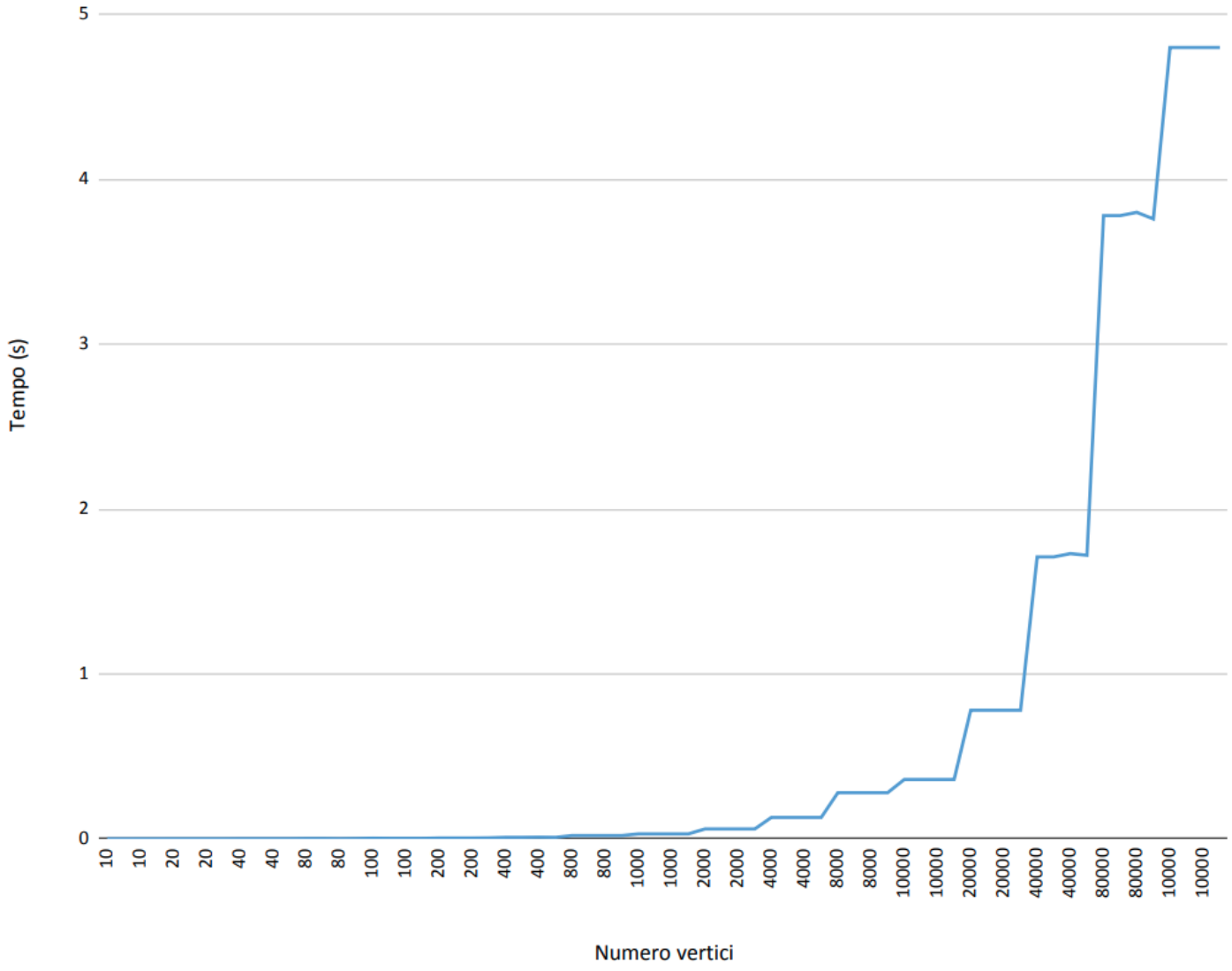


Figura 1: Performance dell'algoritmo di Prim

5.3 Grafico delle performance dell'algoritmo di Kruskal "naive"

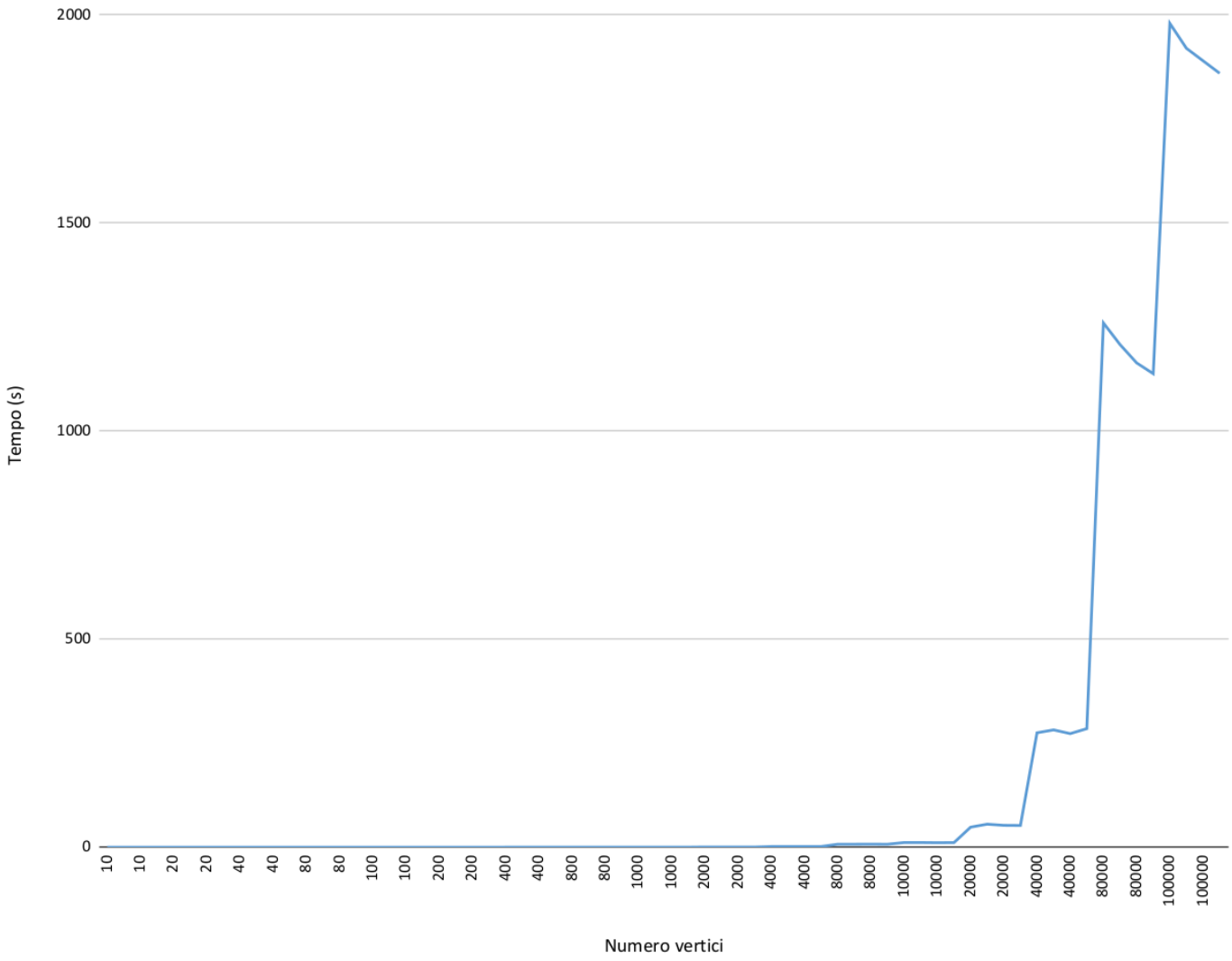


Figura 2: Performance dell'algoritmo di Kruskal "naive"

5.4 Grafico delle performance dell'algoritmo di Kruskal Union Find

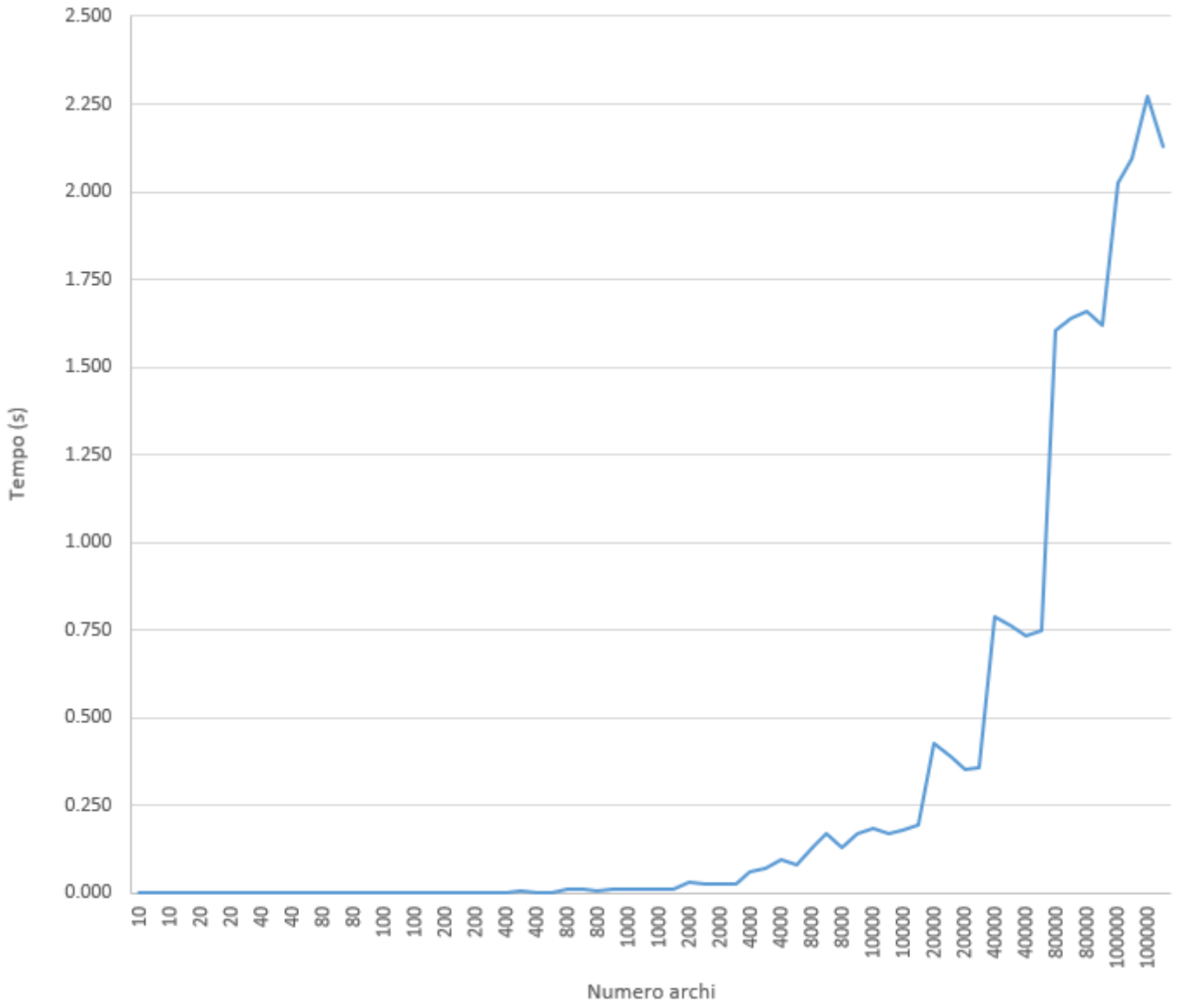


Figura 3: Performance dell'algoritmo di Kruskal con Union Find

6 Conclusione

Una volta raccolte le informazioni relative alle performance dei tre algoritmi eseguiti nel dataset di input, queste sono state collezionate per costruire un grafico che li mettesse a confronto.

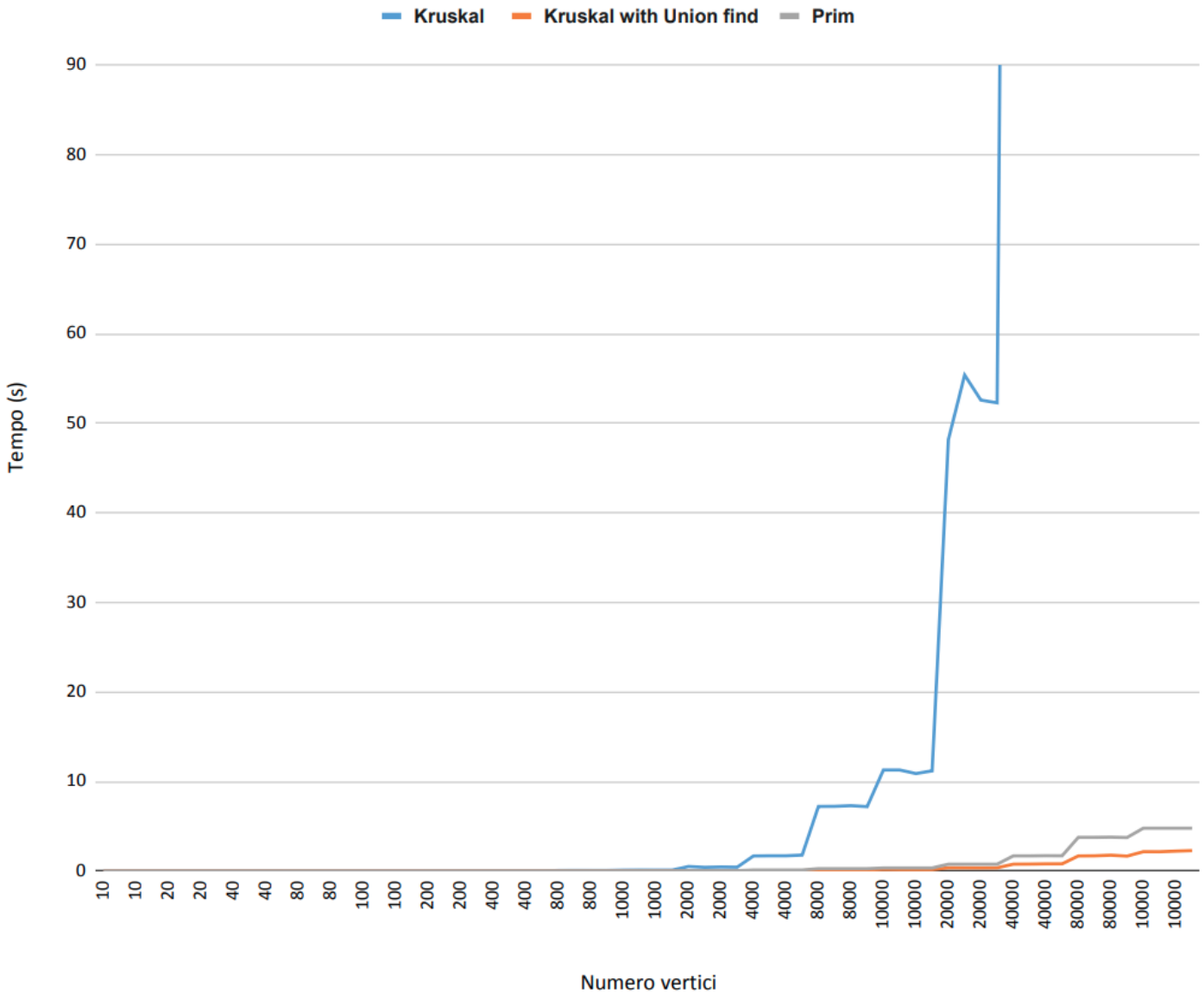


Figura 4: Confronto tra le performance dei tre algoritmi

Dalla Fig. 4 è possibile osservare che per grafi di piccole dimensioni, quindi con un numero di vertici circa pari o inferiore a 1000, la differenza di prestazioni dei diversi algoritmi non è molto evidente. Quando le dimensioni dei grafi di input aumentano, i tempi di esecuzione dell'algoritmo di Kruskal con implementazione "naive" crescono in maniera esponenziale fino

ad impiegare svariati minuti per effettuare il calcolo del MST per i grafi più grandi presenti nel dataset. Gli altri due algoritmi, Prim e Kruskal con Union find, hanno tempi di esecuzione più simili tra loro: pochi secondi per il calcolo del MST anche su grafi di grandi dimensioni. Questi risultati sono coerenti con le complessità degli algoritmi, infatti quello con la complessità più alta è Kruskal in versione "naive". Gli altri due, invece, presentano la stessa complessità algoritmica sebbene sia possibile affermare che l'algoritmo di Kruskal implementato con Union find presenta dei tempi di esecuzione inferiori rispetto a Prim.