Laboratorio 1

Algoritmi Avanzati

Enrico Buratto (2028620)
Mariano Sciacco (2007692)
Federico Zanardo (2020284)

Aprile 2021

Corso di Laurea Magistrale in Informatica, Dipartimento di Matematica, UniPD

Indice

1 Introduzione

1.1 Abstract

In questa relazione illustreremo dei confronti tra tre algoritmi per il calcolo del **Minimum Spanning Tree**: l'algoritmo di *Kruskal* nella versione naive, l'algoritmo di *Kruskal* con *Union-Find* e l'algoritmo di *Prim*.

1.2 Definizione di MST

Sia V l'insieme dei nodi che costituiscono il grafo pesato G e sia E la collezione dei lati di tale grafo. Ai fini delle analisi della complessità degli algoritmi, sia $|V| = n$ e $|E| = m$.

Un *minimum spanning tree* è un sottoinsieme dei lati E di un grafo G non orientato connesso e pesato sui lati che collega tutti i vertici insieme, senza alcun ciclo e con il minimo peso totale del lato possibile. Cioè, è uno spanning tree la cui somma dei pesi dei bordi è la più piccola possibile.

Un *minimum spanning tree* $T = (V, E')$ è un albero, il cui insieme dei lati E' è un sottoinsieme dei lati E di un grafo $G = (V, E)$ non orientato, connesso e pesato, che collega tutti i vertici V, la cui somma dei pesi dei lati è la minima.

L'algoritmo generico per determinare un MST è:

```
A = empty_set
while A doesn't form a spanning tree
    find an edge (u,v) that is safe for A
    A = A U {(u,v)}
return A
```

Si forniscono alcune definizioni per gli MST:

1. un **taglio** $(S, V \setminus S)$ di un grafo $G = (V, E)$ è una partizione di V;
2. un lato $(u, v) \in E$ **attraversa il taglio** $(S, V \setminus S)$ se $u \in S$ e $v \in V \setminus S$ o viceversa;
3. un taglio **rispetta** un insieme A di lati se nessun lato di A attraversa il taglio;
4. dato un taglio, il lato che lo attraversa di peso minimo si chiama **light edge**.

Per determinare se un lato è **safe**, si sfrutta il seguente teorema:

Teorema: Sia $G = (V, E)$ un grafo non diretto, connesso e pesato. Sia A un sottoinsieme di E incluso in una qualche MST di G, sia $(S, V \setminus S)$ un taglio che rispetta A, e sia (u, v) un *light edge* per $(S, V \setminus S)$. Allora (u, v) è *safe* per A.

2 Descrizione degli algoritmi

2.1 Struttura dati per il grafo

La struttura dati per il grafo è stata implementata nel seguente modo:

1. `V` è un insieme di nodi;
2. `E` è una lista di lati;
3. `graph` rappresenta la *lista di adiacenza*. Gli indici per accedere alla mappa sono rappresentati dai vertici. Ogni cella della mappa punta ad una lista di coppie di valori (il vertice a cui è collegato e il peso del lato che li congiunge).

I metodi implementati sono:

1. `add_vertex`: aggiunge un vertice al grafo;
2. `add_edge`: aggiunge un lato al grafo;
3. `remove_edge`: rimuove un lato dal grafo.

Queste operazioni sono state implementate in modo da avere una complessità computazionale costante.

2.2 Kruskal naive

2.2.1 Introduzione

La versione naive dell'algoritmo di Kruskal ha una complessità computazionale pari a $O(mn)$. L'idea alla base di questo algoritmo è quella di ordinare i lati in ordine crescente rispetto al loro peso w. Una volta ordinati, per ogni lato si controlla se aggiungendolo al grafo temporaneo questo crei un ciclo. Se crea un ciclo allora tale lato non verrà inserito, altrimenti il lato farà parte del MST T.

Ordinando i lati in base al loro peso e controllando che l'inserimento di un lato non crei un ciclo nel grafo, si otterrà un albero la cui somma dei suoi lati sarà minima.

Si illustra lo pseudo-codice:

```
Kruskal-Naive(G)
    A = Graph()
    for each vertex v in G.V
        G.add_vertex(v)
    sort the edges of G.E into increasing order by weight w
    for each edge (u, v) in G.E, taken in increasing order by weight
        if A U {(u, v)} is acyclic
            A = A U {(u, v)}
    return A
```

2.2.2 Implementazione

Nella nostra implementazione abbiamo utilizzato l'algoritmo di ordinamento *merge sort*, che ha una complessità computazionale pari a $\Theta(nlog(n))$, per ordinare i lati in ordine crescente rispetto al loro peso.

Il Minimum Spanning Tree viene salvato in una struttura dati `Graph`; il MST può essere esaminato andando ad esplorare la lista di adiacenza `graph` o la lista dei lati `E`.

Il controllo per la ciclicità viene implementato con una versione modificata della *Depth First Search* (*DFS*).

2.2.3 Ottimizzazioni implementate

Per ottimizzare l'algoritmo abbiamo apportato delle opportune modifiche. A partire dal metodo `kruskal_naive`, ad ogni iterazione del ciclo si controlla se il numero di lati del grafo A (quello che conterrà il MST) ha $n-1$ lati, dove n è il numero di vertici del grafo G, fornito in input. Quando si raggiunge questa uguaglianza significa che è stato costruito il MST T, pertanto l'algoritmo può terminare. Questa prima modifica permette di evitare di fare iterazioni inutili che non contribuiranno alla costruzione del MST.

La seconda ottimizzazione implementata riguarda il metodo `_is_acyclic`. Un primo controllo riguarda i vertici connessi dal lato: se i due vertici sono uguali significa che il lato in questione crea un *self-loop*. Con questo controllo siamo in grado eliminare i *self-loop* in tempo costante, senza dover effettuare una chiamata a DFS per verificare che l'aggiunta di tale lato crei un ciclo nel grafo A. Il secondo controllo che è stato implementato in questo metodo riguarda i vertici u e v che il lato, fornito in input, connette. Se almeno uno dei due lati non è presente nel grafo A, allora possiamo concludere che tale lato non introdurrà un ciclo nel grafo, in quanto il lato in questione permetterà di scoprire almeno un nuovo vertice del MST finale. Questa accortezza ci permette di risparmiare delle chiamate a DFS soprattutto nella fase di avvio dell'algoritmo. Se invece, sia u che v sono già presenti

nel grafo A, allora è necessario fare una chiamata a DFS per verificare che quel lato non introduca un ciclo nel grafo.

La terza ottimizzazione riguarda DFS: il metodo che è stato implementato non rispecchia la versione originale dell'algoritmo. Lo scopo di DFS è quello di andare a visitare *tutti* i vertici di un certo grafo; tuttavia, questo non rappresenta il nostro scopo. Il nostro intento è quello di cercare, se esiste, un cammino che congiunge u con v. Se tali vertici si trovano in due componenti connesse distinte (ad esempio nel momento in cui si stanno aggiungendo i lati al grafo A), significa che l'aggiunta del lato (u, v) al grafo A non introdurrà un ciclo. Avviando DFS dal vertice u sul grafo A, desidero verificare se esiste un cammino che mi porta fino a v. Se entrambi i vertici sono nella stessa componente connessa, allora tale cammino verrà trovato, altrimenti tale cammino non verrà trovato perché non esiste. Quindi, DFS verrà eseguita soltanto sulla componente connessa di u e non su tutti i vertici del grafo. Questa versione permette di risparmiare la visita di tutto il grafo e la sua complessità computazionale diventa $O(m)$.

Queste tre ottimizzazioni permettono di evitare di fare eccessive chiamate a DFS e di evitare di andare a visitare l'intero grafo anche quando un cammino da u a v non esiste. La complessità computazionale rimane sempre $O(mn)$, però l'esecuzione è sensibilmente molto più veloce rispetto ad un'implementazione senza queste accortezze.

Il metodo `is_there_a_path` serve soltanto per inizializzare il vettore di booleani che indica se un vertice è stato visitato o meno. Tale vettore è necessario a DFS per esplorare il grafo (o la componente connessa in cui vi è u).

2.3 Kruskal con Union-Find

2.3.1 Introduzione

La versione naive dell'algoritmo di Kruskal svolge ripetutamente l'operazione di controllo di aciclicità del grafo A. Questo continuo controllo è responsabile della complessità computazionale dell'algoritmo. Per ottimizzare le prestazioni, il controllo della ciclicità del grafo viene catturato tramite l'implementazione di una particolare struttura dati: gli **insiemi disgiunti** (o **disjoint sets**).

2.3.2 Struttura dati

La struttura dati per gli insiemi disgiunti è stata implementata come un *albero*. Inizialmente, ogni vertice viene rappresentato come un albero con un solo nodo, il cui *parent* è sè stesso. Man mano che si vanno a visitare i vari lati del grafo G, si controlla se i due vertici, connessi al lato in questione, appartengono allo stesso albero o meno. Se appartengono allo stesso albero significa che hanno lo stesso nodo radice, se invece appartengono ad alberi differenti allora avranno radici diverse. Per l'im-

plementazione di questo algoritmo non è importante né l'ordine dei nodi salvati nell'albero, né quale sia il nodo alla radice dell'albero. Se i vertici del lato (u, v) appartengono a due alberi diversi, allora si procede ad eseguire una *union* dei due alberi. Questa operazione è stata implementata come una *union-by-size*. Per poter implementare tale operazione nella struttura dati si tiene conto anche della *size* di ogni albero: ogni nodo memorizza la sua dimensione, che rappresenta il numero di discendenti (incluso sè stesso). Nel momento in cui bisogna unire i due alberi, l'albero con la dimensione maggiore diventa genitore dell'altro; nel caso in cui i due alberi abbiano la stessa dimensione, allora la scelta diventa casuale, a meno che non debbano essere mantenute determinate proprietà (non è questo il nostro caso).

I metodi implementati per tale struttura dati sono:

1. `make_set(x)`: crea un albero costituito da un solo nodo il cui valore è x, la *size* è impostata a 1 e il *parent* è sè stesso. Complessità computazionale: costante;
2. `find_set(x)`: ritorna il nodo radice dell'albero a cui appartiene il nodo che ha valore x. Complessità computazionale: $O(logn)$;
3. `union_by_size(x, y)`: dati i valori di due nodi, vengono determinati i nodi radice per ciascuno e se appartengono ad alberi diversi allora si procede nell'unire i due alberi, secondo le procedure definite precedentemente. Complessità computazionale: $O(log(n))$.

2.3.3 Implementazione

Lo pseudo-codice non si discosta molto dalla versione naive dell'algoritmo. Le uniche parti che variano sono:

1. A non è più un grafo, ma è una collezione di lati;
2. Non è più richiesto effettuare il controllo che il grafo sia aciclico.

Si illustra lo pseudo-codice:

```
Kruskal-Union-Find(G)
    A = empty_set
    for each vertex v in G.V
        make-set(v)
    sort the edges of G.E into increasing order by weight w
    for each edge (u, v) in G.E, taken in increasing order by weight
        if find-set(u) != find-set(v)
            A = A U {(u, v)}
```

```
            union(u, v)
    return A
```

Il controllo per la ciclicità del grafo viene sostituito con la verifica che i due vertici del lato (u, v) appartengano o meno a due alberi distinti. Se i due vertici hanno il nodo radice in comune significa che esiste già un percorso da u a v e quindi aggiungendo il lato (u, v) si creerebbe un ciclo. Se invece i due vertici appartengono a due alberi diversi, allora l'introduzione di tale arco non creerà un ciclo nel grafo, ma permetterà di scoprire dei nuovi vertici del MST T finale. Quindi, il lato (u, v) verrà aggiunto ad A e i due alberi verranno uniti in base alla loro *size*.

L'introduzione degli insiemi disgiunti riduce la complessità computazionale a $O(mlog(n))$. Analizzando l'algoritmo, si può osservare che:

1. vengono svolte n chiamate a `make_set(x)`;
2. vengono ordinati i lati in ordine crescente con *merge sort*, quindi si ha una complessità pari a $\Theta(mlog(m))$;
3. il ciclo for ha una complessità pari a $O((n + n)\alpha(n)))$, dove α è una funzione che cresce molto lentamente (è l'inversa della funzione di Ackermann). Assumendo che G sia connesso, le operazioni per gli insiemi disgiunti hanno una complessità pari a $O(m\alpha(n))$. Inoltre, possiamo dire che $\alpha(n) = O(log(n)) = O(log(m))$, quindi la complessità dell'algoritmo nel suo complesso diventerebbe $O(mlog(m))$.

   Si può osservare però che i grafi rispettano la diseguaglianza $m < n^2$. Quindi, $log(m) = O(log(n))$. Con questa osservazione, la complessità finale dell'algoritmo è $O(mlog(n))$.

2.4 Prim

2.4.1 Introduzione

La versione base dell'algoritmo di Prim ha una complessità computazionale pari a $\mathcal{O}(mn)$. L'idea alla base di questo algoritmo è quella di partire da un nodo arbitrario e scegliere, a ogni iterazione, l'arco che connetta con il minor peso possibile l'albero dei nodi del *minimum spanning tree* al nuovo vertice.
Nonostante la complessità sia polinomiale, e quindi efficiente, l'algoritmo può essere ottimizzato utilizzando la corretta struttura dati: l'implementazione dell'algoritmo di Prim con l'utilizzo di *Heap*, infatti, permette di calcolare il minimo nodo da aggiungere all'albero in tempo logaritmico; la complessità computazionale dell'algoritmo in questa implementazione, quindi, diventa
$\mathcal{O}(m + nlog(n))$.
Si illustra lo pseudocodice dell'algoritmo di Prim con l'utilizzo di *Heap*:

```
Prim(G,s)
    for each u in V do
        key[u] <- +inf
        parent[u] <- nil
    key[s] <- 0
    Q <- V
    while Q not empty do
        u <- extractMin(Q)
        for each v adjacent to u do
            if v in Q and w(u,v) < key[v] then
                parent[v] <- u
                key[v] <- w(u,v)
```

2.4.2 Struttura dati

Per la memorizzazione dei vertici del grafo di partenza, e per estrarne successivamente il nodo di peso minimo, abbiamo creato una classe `Heap` con i seguenti campi dati:

- `list`, ossia una lista contenente i nodi del grafo rappresentati da oggetti di tipo `Node`;
- `mapList`, ossia una mappa che permette di associare un nodo alla sua posizione in `list`. L'utilità di questa variabile viene discussa nel capitolo successivo;
- `currentSize`, ossia la dimensione del grafo.

La classe `Heap` mette a disposizione i metodi classici della struttura dati *Heap*; particolarmente importanti ai fini del suo utilizzo nell'algoritmo sono i seguenti metodi:

- `search` permette di sapere se un nodo è presente o meno nel grafo, ed è di particolare importanza per il controllo dell'esistenza di un nodo durante l'esecuzione dell'algoritmo;
- `extractMin` permette di estrarre il nodo di peso minimo in tempo logaritmico;
- `searchAndUpdateWeight` permette di aggiornare il peso di un nodo, ed è necessario per l'aggiornamento del grafo in seguito alla rimozione del nodo in testa al ciclo *while*.

2.4.3 Implementazione

La nostra implementazione ricalca l'implementazione standard dell'algoritmo di Prim con l'utilizzo di un *MinHeap* come struttura dati a supporto; essa non si discosta quindi dallo pseudocodice precedentemente mostrato.

2.4.4 Ottimizzazioni implementate

Anche per quanto riguarda l'algoritmo di Prim abbiamo apportato una opportuna modifica per ottimizzarne l'esecuzione.
Tale ottimizzazione consiste nell'utilizzo di una mappa, concretizzata in una variabile di tipo `defaultdict` in *Python*, che associa a ogni nodo la sua posizione in `list`, ossia la lista dei nodi. Questa implementazione permette di ricercare la presenza di un vertice nello *Heap* in tempo costante, e ciò è particolarmente utile perché la presenza del vertice deve essere verificata due volte ad ogni iterazione del ciclo *for* dell'algoritmo (nella condizione *if* e durante l'*update* dello *Heap*). Se non avessimo adottato tale ottimizzazione, queste operazioni avrebbero avuto una complessità lineare, andando così ad aumentare polinomialmente la complessità computazionale dell'algoritmo.

3 Caratteristiche del programma e Misurazioni

3.1 Caratteristiche del programma

3.1.1 Introduzione

Il programma è stato realizzato interamente in Python e si struttura in 4 cartelle principali che rappresentano: il modulo degli algoritmi (*algorithms*), il modulo delle strutture dati (*data_structures*), il modulo delle misurazioni (*measurements*) e il dataset contenente i grafi (*dataset*).

3.1.2 Installazione e requisiti

Prima di eseguire il programma è necessario installare le dipendenze presenti nel file *requirements.txt* eseguendo il seguente comando: `pip install -r requirements.txt`
Per l'esecuzione è richiesto l'uso di un terminale Windows, MacOs, Linux o BSD-like con Python 3.x+ e PIP installato.

3.1.3 Avvio del programma

Per l'esecuzione del programma è necessario spostarsi nella cartella di progetto ed eseguire il seguente comando: `python3 main.py [--version] [-h] <type> <dataset>` dove:

- **type:** rappresenta il tipo di algoritmo o di serie di esecuzioni che si vuole avviare, nello specifico uno tra i seguenti:
  - *all* (esegue e salva su file le misurazioni singole);
  - *all-quartet* (esegue e salva su file le misurazioni a quartetto);
  - *prim* (esegue le misurazioni con Prim);
  - *kruskal* (esegue le misurazioni con Kruskal Naive);
  - *kruskal-opt* (esegue le misurazioni con Kruskal Union Find).
- **dataset:** rappresenta una cartella contenente i file dei dataset formattati come da consegna o singolo file di dataset in input.
- **–version:** è opzionale e mostra la versione del programma in esecuzione.
- **-h:** è opzionale e mostra un aiuto per i comandi da utilizzare.

Un esempio di esecuzione di tutte le misurazioni singole e di un singolo file con kruskal naive è il seguente:

```
python3 main.py all dataset/
python3 main.py kruskal dataset/input_random_69_200000.txt
```

3.1.4 Considerazioni rilevanti sul programma

Inizialmente, data la mole di dati da dover analizzare, si è pensato di implementare in modalità *multiprocessing* il programma, così da favorire l'uso di tutti i thread a disposizione nei moderni processori per computer. Per questo motivo, le misurazioni sono state realizzate con l'idea di poter eseguire un *pool* di thread dove ciascun thread rappresentava l'avvio di un algoritmo.

Nel corso delle misurazioni si è notato però che l'esecuzione rallentava notevolmente, dal momento che monitorando l'uso della CPU solo un core veniva usato costantemente al 100%. Successivamente siamo venuti a conoscenza del fatto che nel caso specifico di **Python** non è possibile occupare in modo quasi esclusivo più di un thread della CPU (oltre al *main*), essendoci delle limitazioni interne dipendenti dal linguaggio.

Il codice delle misurazioni pertanto è stato riadattato in modo poter essere eseguito sequenzialmente, lasciando comunque la possibilità a un veloce riadattamento per l'esecuzione in *multiprocessing*. Alcuni degli aspetti positivi di questa modalità possono riguardare sicuramente l'esecuzione parallela per il calcolo dei risultati degli algoritmi, la cui computazione risulterebbe sicuramente di minore tempistica specialmente nel caso di Kruskal Naive.

3.1.5 Caratteristiche tecniche del computer per le misurazioni

L'esecuzione del programma e le relative misurazioni sono state effettuate sulla seguente macchina:

- **CPU:** Ryzen 9 3900x (12 core / 24 thread), 4.6ghz
- **RAM:** 32 GB DDR4
- **SSD:** NVME SSD 512 GB

Tutte le misurazioni sono state eseguite **sequenzialmente** monitorando l'uso di un core dedicato della CPU al 100% per tutta la durata dell'esecuzione.

3.2 Introduzione alle misurazioni

Le misurazioni sono state effettuate attraverso l'implementazione di due moduli che eseguono rispettivamente due tipi di misurazione:

- **Misurazione singola:** esecuzione dei tre algoritmi per ogni grafo del dataset con misurazione del tempo medio;
- **Misurazione a quartetto:** esecuzione dei tre algoritmi per ogni quartetto di grafi con lo stesso numero di vertici.

In entrambi i casi le misurazioni vengono effettuate dopo aver recuperato e caricato l'intero dataset nella struttura dati GRAPH, come menzionato precedentemente. Prima di ogni esecuzione dei vari algoritmi è stato inoltre disabilitato temporaneamente il *garbage collector* di Python, così da evitare rallentamenti anche minimi nel corso dell'esecuzione. Infine, ciascuno dei due moduli salva in tre file differenti - uno per ogni algoritmo - i risultati del programma.

3.3 Misurazione singola

3.3.1 Descrizione

La misurazione singola è organizzata sequenzialmente eseguendo in base al tipo di algoritmo l'intero dataset. Il metodo *executeSingleGraphCalculus* applica l'algoritmo richiesto al graph in input e procede nella seguente maniera:

1. Esegue una volta l'algoritmo applicato al grafo, disabilitando il *garbage collector* e salvando il tempo di esecuzione.
2. Verifica se il tempo di esecuzione è maggiore o meno a 1s.
   - Se il tempo è minore a 1s, esegue nuovamente k volte l'algoritmo, disabilitando il *garbage collector* e salvando il tempo di esecuzione medio sulle k esecuzioni, con k definito come segue:
     $$k = \left\lfloor \frac{10^9 \text{ ns}}{\text{tempo medio di esecuzione in ns}} \right\rfloor$$
     ossia il numero di ripetizioni applicate all'algoritmo la cui somma arriva a 1s.
   - Altrimenti, viene mantenuto il tempo rilevato precedentemente con una singola esecuzione.
3. Esegue il salvataggio in *append* nel file di output in formato CSV.

3.3.2 File di output

Il file di output viene salvato direttamente alla fine di ogni esecuzione dell'algoritmo mantenendo la seguente formattazione:

- Numero del dataset;
- Numero di vertici del grafo;
- Numero di archi del grafo;
- Tempo di esecuzione in nano secondi;
- Tempo di esecuzione in secondi;
- Peso finale calcolato;
- Esecuzioni totali effettuate dell'algoritmo.

3.4 Misurazione a quartetto

3.4.1 Descrizione

La misurazione a quartetto è organizzata sequenzialmente eseguendo in base al tipo di algoritmo l'intero dataset raggruppando a gruppi di 4 i grafi con lo stesso numero di vertici. Per questa particolare casistica applichiamo la premessa che il dataset in input abbia sempre 4 file adiacenti con lo stesso numero di vertici. Il metodo *executeSingleQuartetMeasurement* applica l'algoritmo richiesto al quartetto di grafi e procede nella seguente maniera:

1. Esegue una volta l'algoritmo applicato al quartetto di grafi, disabilitando il *garbage collector* e salvando il tempo di esecuzione medio.
2. Verifica se il tempo di esecuzione è maggiore o meno a 1s.
   - Se il tempo è minore a 1s, esegue nuovamente k volte l'algoritmo per ogni quartetto di grafi, disabilitando il *garbage collector* e salvando il tempo di esecuzione medio sulle k esecuzioni, con k definito come segue:
     $$k = \left\lfloor \frac{10^9 \text{ ns}}{\text{tempo medio di esecuzione in ns}} \right\rfloor$$
     ossia il numero di ripetizioni applicate all'algoritmo la cui somma arriva a 1s.
   - Altrimenti, viene mantenuto il tempo rilevato precedentemente, pari all'esecuzione media del quartetto di grafi.
3. Esegue il salvataggio in *append* nel file di output in formato CSV.

3.4.2 File di output

Il file di output viene salvato direttamente alla fine di ogni esecuzione del metodo mantenendo la seguente formattazione:

- Numero dei vertici;
- Numeri del dataset di interesse;
- Numeri di archi dei grafi;
- Numero medio di archi nei grafi;
- Tempo di esecuzione in nano secondi;
- Tempo di esecuzione in secondi;
- Esecuzioni totali effettuate per il quartetto.
- Esecuzioni effettuate per ogni grafo del quartetto.

4 Risposte alle domande

4.1 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 variare del numero di vertici nel grafo. Confrontate i tempi misurati con la complessità asintotica attesa dalla teoria. Per ognuna delle istanze del problema, riportate il peso del minimum spanning tree ottenuto dagli algoritmi.

Abbiamo implementato il codice in Python per l'esecuzione dei tre algoritmi su tutto il dataset fornito. I risultati, che sono stati riportati in modo dettagliato e con i relativi pesi anche nella sezione appendice, sono riportati di seguito.

4.1.1 Risultati con Kruskal naive

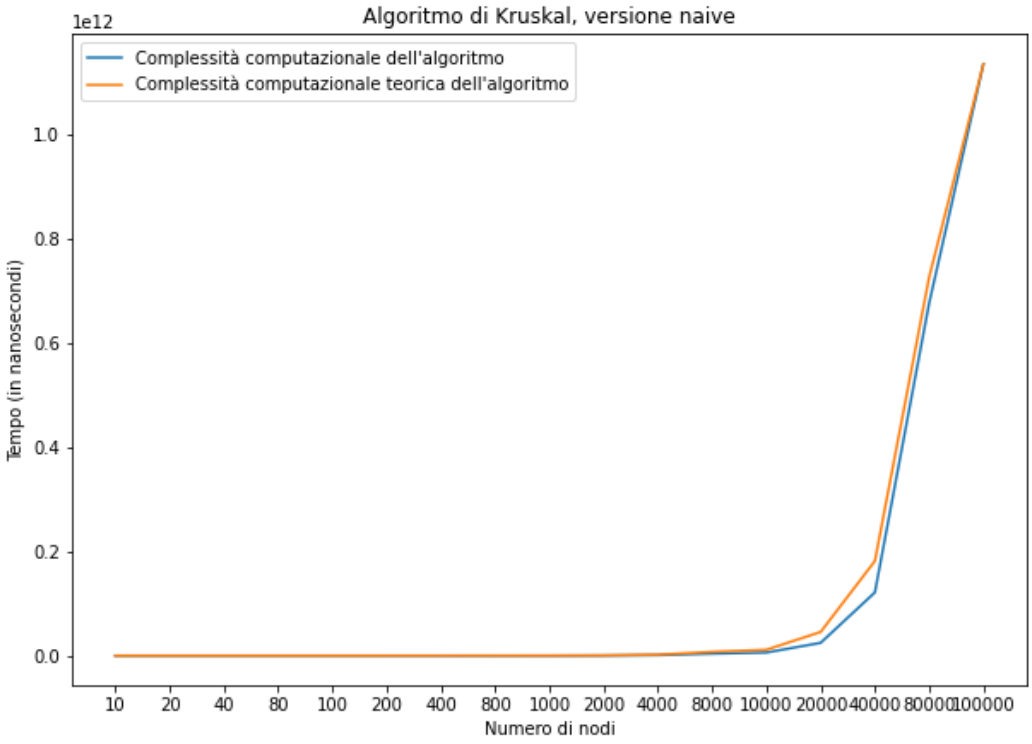

Figura 1: Complessità di Kruskal Naive con una esecuzione per ogni quartetto di grafi con uguale numero di nodi.

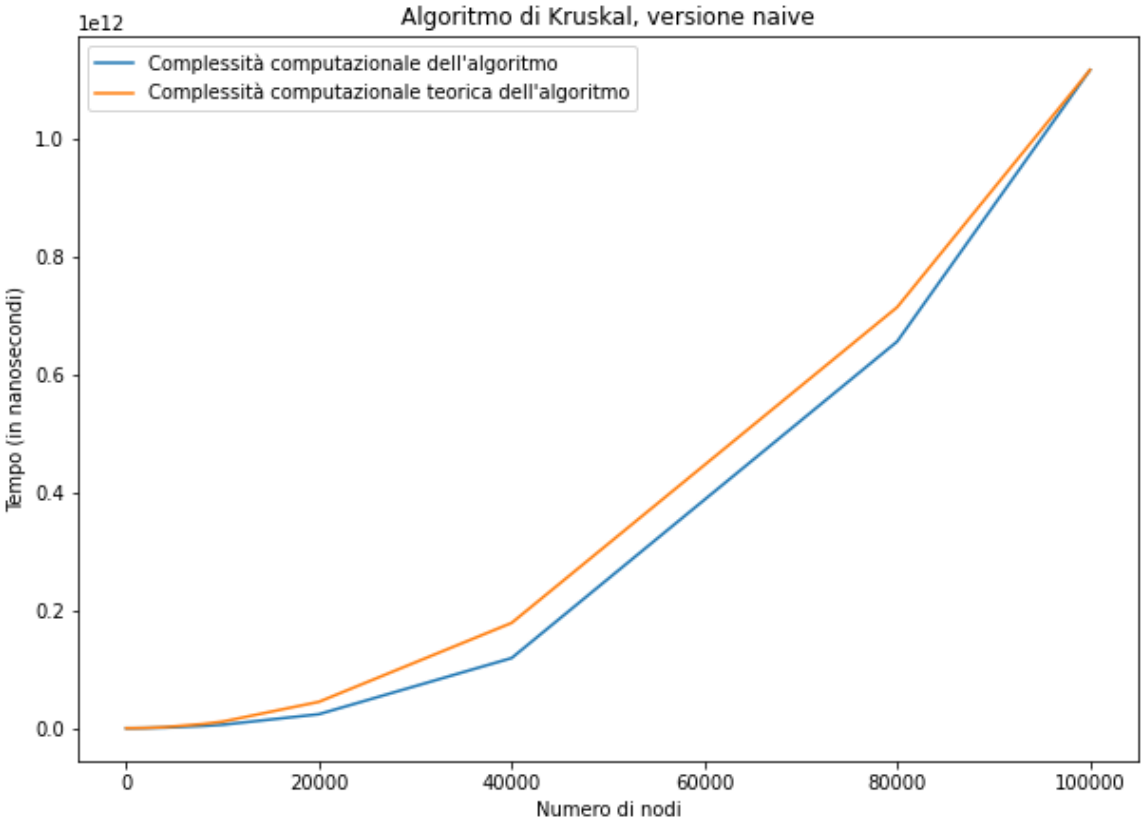

Figura 2: Complessità di Kruskal Naive con k esecuzioni ripetute per ogni quartetto di grafi con uguale numero di nodi.

Nel grafico appena illustrato (fig. 2) è riportata la complessità computazionale attesa (in giallo) ed effettiva (in blu) per l'algoritmo di Kruskal Naive con più esecuzioni dell'algoritmo. Come si può evincere dall'immagine, la curva della complessità effettiva rimane leggermente al di sotto della curva teorica, e pertanto le due complessità sono equiparabili.

4.1.2 Risultati con Kruskal Union Find

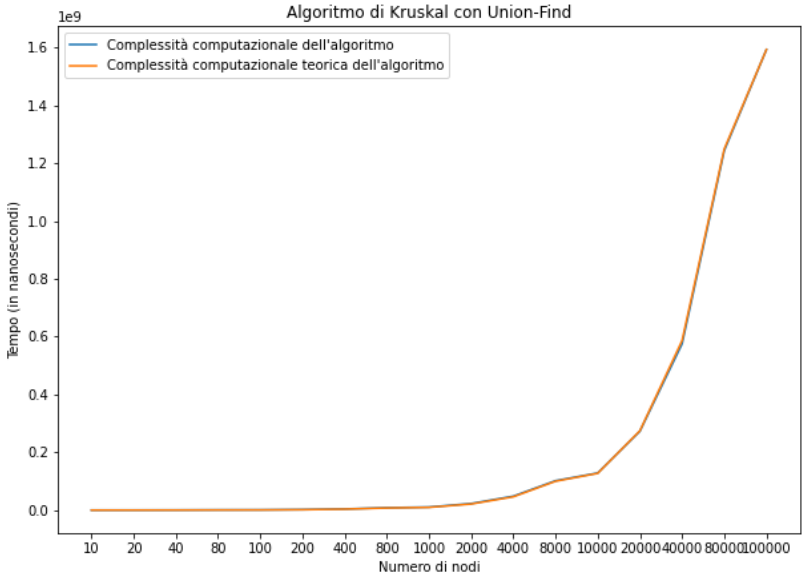

Figura 3: Complessità di Kruskal Union Find con una esecuzione per ogni quartetto di grafi con uguale numero di nodi.

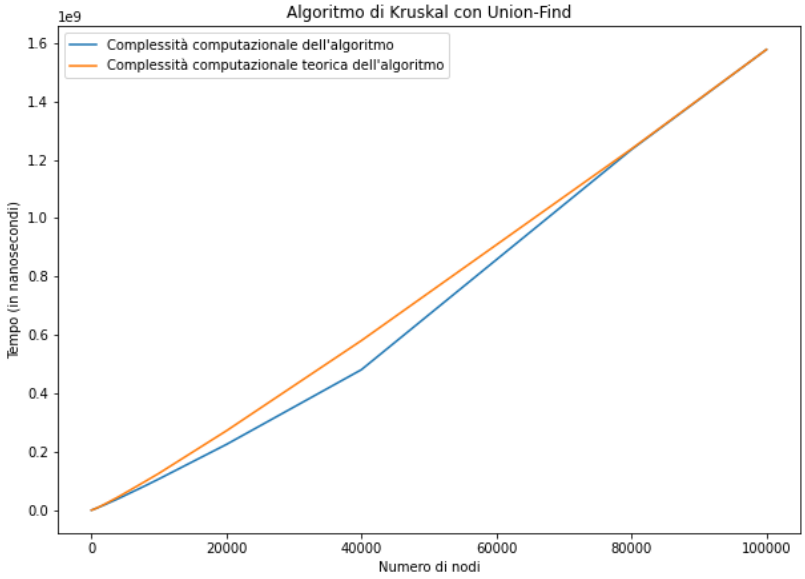

Figura 4: Complessità di Kruskal Union Find con k esecuzioni ripetute per ogni quartetto di grafi con uguale numero di nodi.

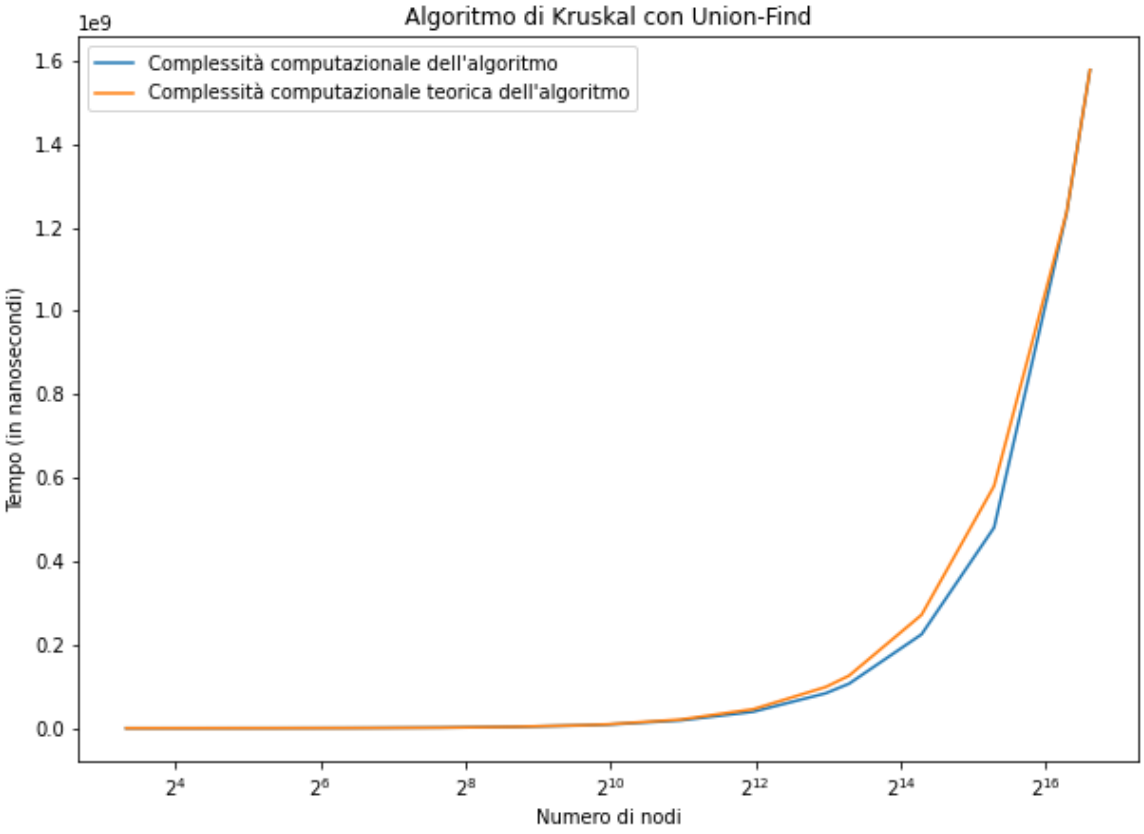

Figura 5: Complessità di Kruskal Union Find con k esecuzioni ripetute per ogni quartetto di grafi con uguale numero di nodi, in scala logaritmica.

Nei grafici appena illustrati (fig. 4, 5) è riportata la complessità computazionale attesa (in giallo) ed effettiva (in blu) per l'algoritmo di Kruskal Union Find; il grafico 5 è riportato in scala logaritmica per meglio apprezzare l'andamento asintotico dell'algoritmo. Come si può evincere dall'immagine, anche in questo caso la curva della complessità effettiva rimane leggermente al di sotto della curva teorica.

4.1.3 Risultati con Prim

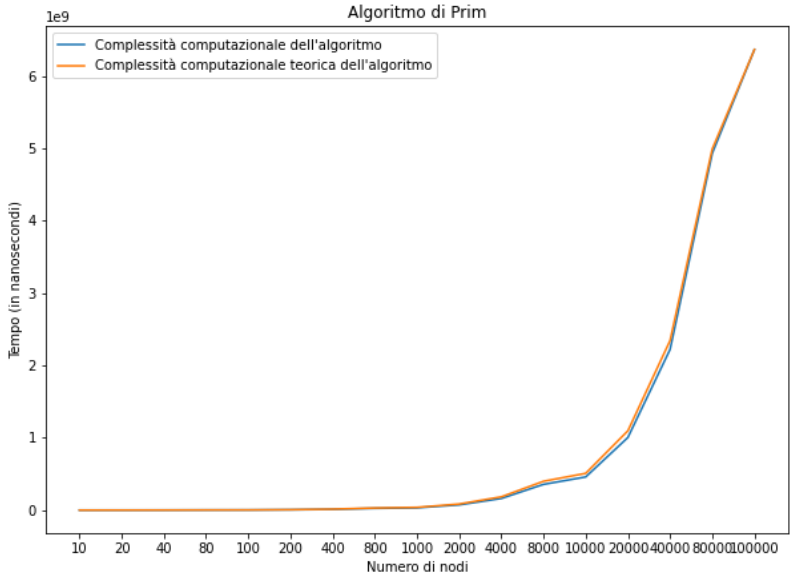

Figura 6: Complessità di Prim con una esecuzione per ogni quartetto di grafi con uguale numero di nodi.

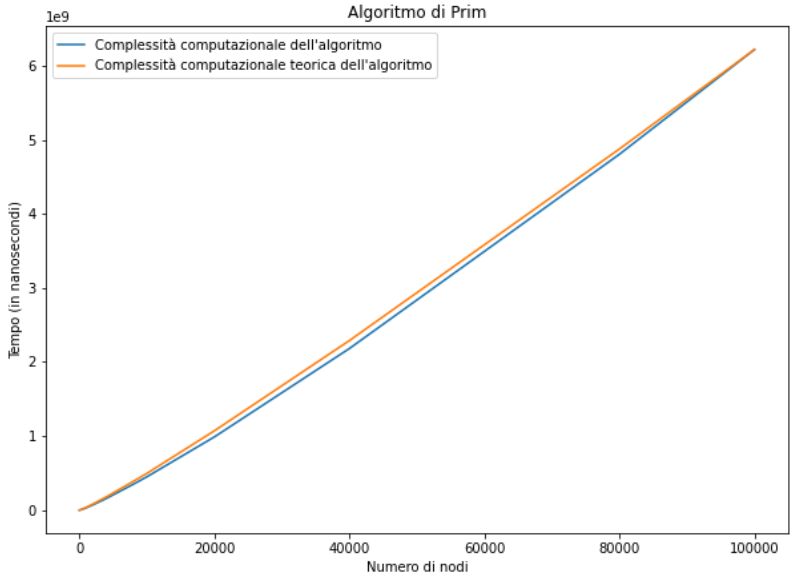

Figura 7: Complessità di Prim con k esecuzioni ripetute per ogni quartetto di grafi con uguale numero di nodi.

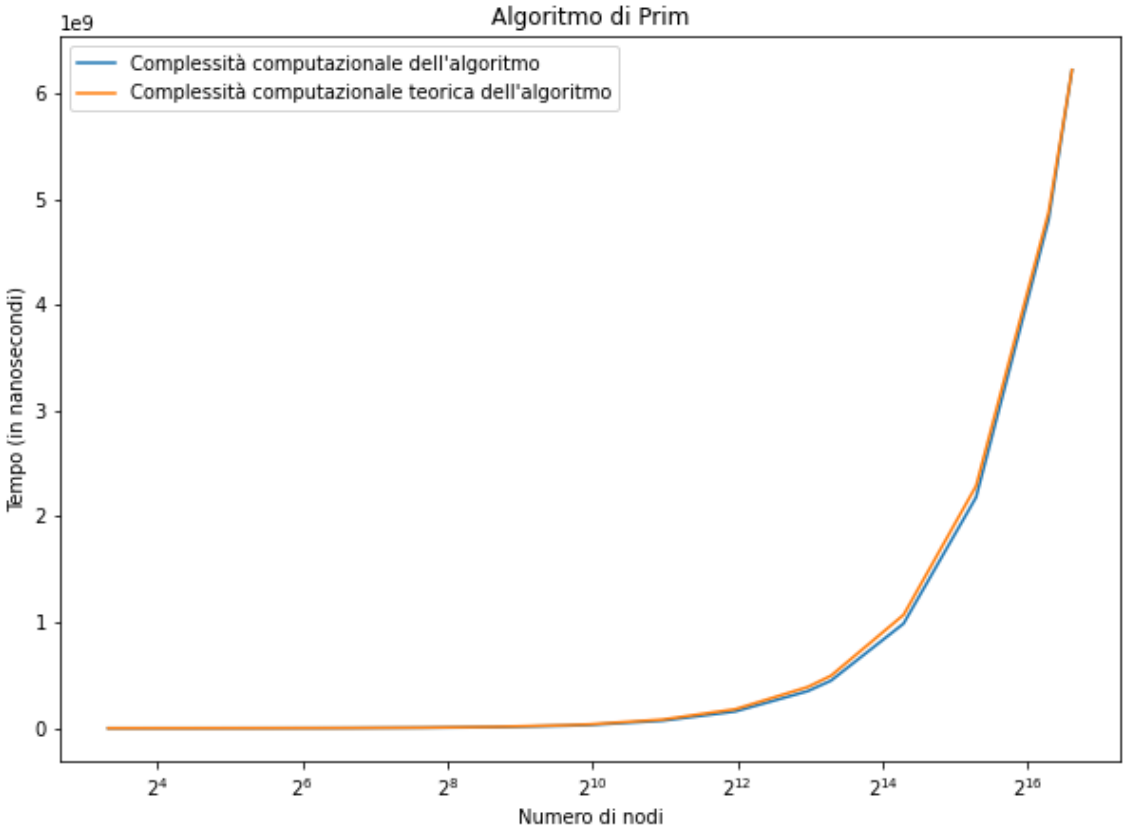

Figura 8: Complessità di Prim con k esecuzioni ripetute per ogni quartetto di grafi con uguale numero di nodi, in scala logaritmica.

Nei grafici appena illustrati (fig. 7, 8) è riportata la complessità computazionale attesa (in giallo) ed effettiva (in blu) per l'algoritmo di Prim; il grafico 8 è riportato in scala logaritmica per meglio apprezzare l'andamento asintotico dell'algoritmo. Come si può evincere dall'immagine, anche in questo caso la curva della complessità effettiva rimane leggermente al di sotto della curva teorica.

4.2 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?

Analizzando i tre algoritmi abbiamo appurato che nel caso di Kruskal Naive l'algoritmo richiede un tempo medio di esecuzione in un ordine decisamente superiore rispetto a Kruskal UF e Prim.

Infatti, il tempo medio calcolato nel caso del quartetto contenente il maggior numero di nodi (100000) i risultati sono stati i seguenti:

Num. di nodi	**Datasets**	**Kruskal naive [s]**	**Kruskal UF [s]**	**Prim [s]**
100000	65, 66, 67, 68	1117.8413379	1.5777018	6.2187226

Confrontando invece i risultati dei grafi con un minor numero di vertici abbiamo potuto constatare quanto segue:

Num. di nodi	**Datasets**	**Kruskal naive [s]**	**Kruskal UF [s]**	**Prim [s]**
10	1, 2, 3, 4	0.0000450	0.0000465	0.0000868
20	5, 6, 7, 8	0.0001154	0.0001138	0.0002398
40	9, 10, 11, 12	0.0002559	0.0002456	0.0005888

I tempi rilevati con Kruskal Naive risultano più vicini agli altri algoritmi dal momento che avvicinandoci a un minor numero di nodi non si può apprezzare una differenza di tempi di esecuzione media, visto che per valori di n che tendono a 0 le funzioni delle complessità computazionali degli algoritmi restituiscono dei valori molto simili.

4.2.1 Confronto Prim-Kruskal Union Find

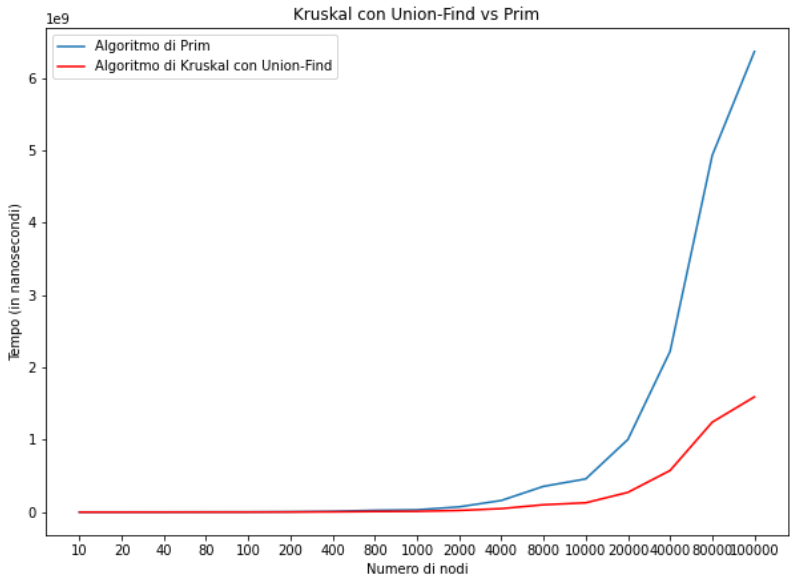

Figura 9: Confronto tra Prim e Kruskal UF con una esecuzione per ogni quartetto di grafi con uguale numero di nodi.

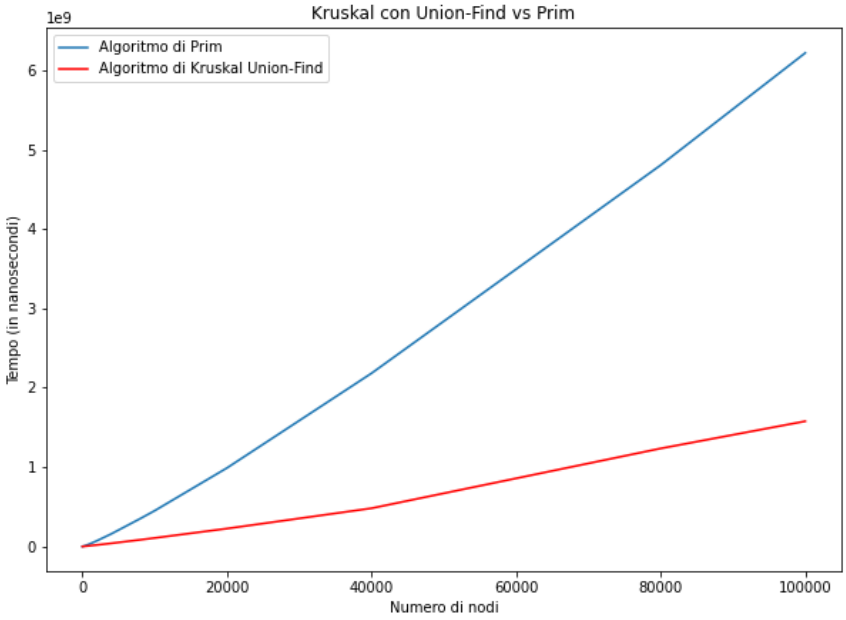

Figura 10: Confronto tra Prim e Kruskal UF con k esecuzioni ripetute per ogni quartetto di grafi con uguale numero di nodi.

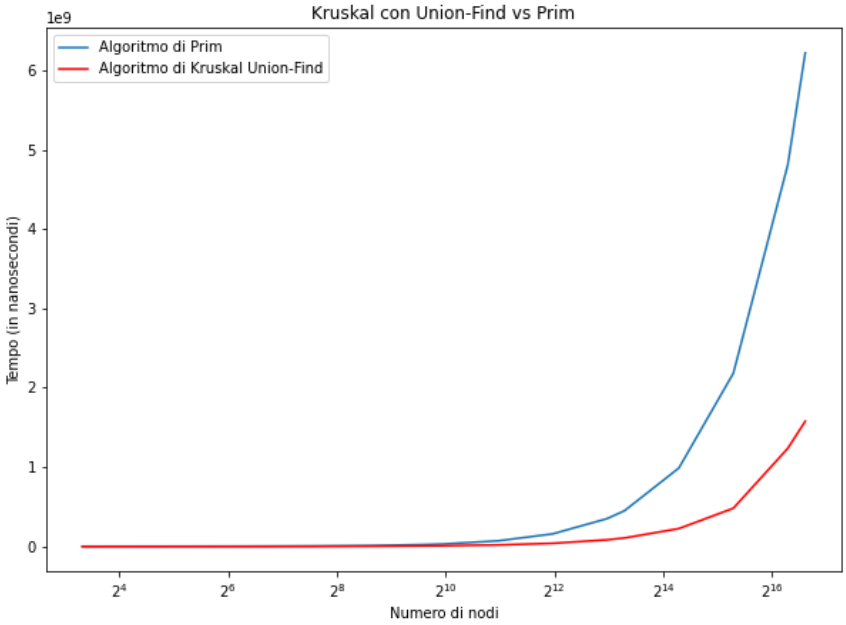

Figura 11: Confronto tra Prim e Kruskal UF con k esecuzioni ripetute per ogni quartetto di grafi con uguale numero di nodi, in scala logaritmica.

Nei grafici appena illustrati (fig. 9, 10, 11) sono riportati tre confronti tra l'algoritmo di Kruskal Union Find e l'algoritmo di Prim: i primi due in scala lineare e il terzo in scala logaritmica. In tutti e tre i casi è possibile vedere chiaramente che **l'algoritmo di Kruskal UF risulta essere più efficiente rispetto all'algoritmo di Prim**, dal momento che il tempo di esecuzione medio su un quartetto di grafi è moderatamente più basso per Kruskal UF. Da questo pertanto possiamo concludere che l'algoritmo di Kruskal UF tende a fare sempre meglio rispetto agli altri.

5 Conclusione

Alla fine del lavoro svolto possiamo dire che le ottimizzazioni che sono state implementate per la versione naive dell'algoritmo di Kruskal si sono rivelate utili per l'esecuzione dei primi grafi. Infatti,

1. Il controllo dei *self-loop* e della presenza o meno dei nodi dell'arco (u, v) hanno permesso di risparmiare diverse chiamate a DFS per il controllo della ciclicità del grafo G;
2. La versione modificata di DFS permette di evitare la visita di tutto il grafo quando non è necessario. È sufficiente controllare che esista un cammino da u a v nella componente connessa di u. Se tale cammino non esiste, allora il lato (u, v) farà parte del MST T finale.

Queste ottimizzazioni hanno permesso di ottenere delle performance migliori nel momento in cui l'algoritmo inizia ad analizzare il grafo ricevuto in input. È possibile notare che i tempi per Kruskal Naive sono paragonabili ai tempi degli altri due algoritmi per grafi con un piccolo numero di vertici (cfr. §4.3).

In secondo luogo, per quanto riguarda l'implementazione della struttura dati Heap utilizzata per l'algoritmo di Prim abbiamo potuto constatare che l'ottimizzazione integrata attraverso una mappa che permettesse di salvare l'indice di posizione corrente nella lista ausiliaria ha ridotto di gran lunga la durata media dell'algoritmo di Prim, tendendo a essere più simile al tempo medio ottenuto da Kruskal Union Find. Questa miglioria ha permesso di accedere quindi in tempo costante ai singoli elementi della lista rimuovendo la complessità lineare, che precedentemente avrebbe reso computazionalmente più lento di un valore polinomiale l'algoritmo di Prim.

Per quanto concerne le misurazioni effettuate abbiamo voluto rendere ciascuna misurazione il più preciso possibile così da ottenere risultati sui tempi medi di esecuzione più attendibili. Difatti, ripetendo l'esecuzione di ciascun grafo con tempo di esecuzione minore al secondo di un fattore k, che rappresenta il numero di ripetizioni da eseguire per rientrare entro il secondo sulla base del tempo per una esecuzione, siamo riusciti ad avere un risultato più comparabile che ci ha permesso di fare dei confronti più interessanti (vedasi i risultati della sezione §4).

Per quanto concerne il *workflow* adottato all'interno del gruppo, avendo lavorato con un sistema di versionamento siamo riusciti ad organizzarci per realizzare e sviluppare attraverso un approccio incrementale ciascun algoritmo. Inizialmente infatti abbiamo iniziato con lo studiare bene il funzionamento di ogni algoritmo e poi, gradualmente, lo abbiamo implementato partendo dalle strutture dati richieste. Una volta controllato che in tutti e 3 i casi gli algoritmi ottenevano risultati identici sul peso finale degli archi, abbiamo ottimizzato ciascuno di essi limando sulle parti di codice superflue ed applicando delle migliorie in base a quanto il linguaggio Python ha da offrire. Infine, abbiamo voluto anche rendere facilmente utilizzabile l'applicazione finale, così da poter provare diversi dataset in diverse modalità. Per la realizzazione dei grafi, tuttavia, abbiamo preferito raccogliere i risultati e

generare i grafici sfruttando Google Colab, la cui logica è stata riportata per completezza all'interno di una cartella del progetto (*graph-generation*).

In linea di massima siamo stati contenti dei risultati ottenuti sebbene l'uso del linguaggio Python, pur se relativamente semplice, ci è risultato abbastanza disordinato per la mancanza di *strong typing*, cosa che ci ha causato inizialmente diversi rallentamenti nello sviluppo degli algoritmi. Per concludere, quanto ci è risultato per gli algoritmi è in linea con quanto ci aspettavamo, difatti:

- Kruskal Naive risulta il più lento tra i tre algoritmi e il meno efficiente in termini di tempo medio di esecuzione per grafi con molti nodi;
- Kruskal Union Find risulta il più veloce e il più efficiente tra i tre algoritmi, specialmente con grafi di grandi dimensioni;
- Prim si può classificare in secondo posto rispetto a Kruskal Union Find, avendo un tempo di esecuzione sufficientemente buono anche per grafi di grandi dimensioni.

Appendice

A Appendice

A.1 Risultati output Kruskal Naive

Dataset	Nodes	Edges	Time [ns]	Time [s]	Weight	Repetitions
1	10	9	38309.5888407	0.0000383	29316	13603
2	10	11	44351.3977144	0.0000444	16940	16713
3	10	13	51044.9467734	0.0000510	-44448	18487
4	10	10	43787.7281781	0.0000438	25217	21286
5	20	24	98822.0378053	0.0000988	-32021	9787
6	20	24	105718.8573739	0.0001057	25130	9276
7	20	28	134031.3407179	0.0001340	-41693	7132
8	20	26	114722.4178873	0.0001147	-37205	8397
9	40	56	255071.3821475	0.0002551	-114203	3865
10	40	50	248330.5537729	0.0002483	-31929	3989
11	40	50	256882.2292204	0.0002569	-79570	3874
12	40	52	241759.0395296	0.0002418	-79741	3997
13	80	108	801811.0751020	0.0008018	-139926	1225
14	80	99	716404.7953724	0.0007164	-198094	1383
15	80	104	732120.7298298	0.0007321	-110571	1351
16	80	114	934089.6933333	0.0009341	-233320	1050
17	100	136	1141836.2448513	0.0011418	-141960	874
18	100	129	924543.3148148	0.0009245	-271743	1080
19	100	137	922108.6626054	0.0009221	-288906	1067
20	100	132	926144.7014925	0.0009261	-229506	1072
21	200	267	2905119.5714286	0.0029051	-510185	336
22	200	269	3079901.2408537	0.0030799	-515136	328
23	200	269	3062370.9631902	0.0030624	-444357	326
24	200	267	3449758.7577855	0.0034498	-393278	289
25	400	540	11152807.2247191	0.0111528	-1119906	89
26	400	518	9591474.8571429	0.0095915	-788168	105
27	400	538	10962710.3444444	0.0109627	-895704	90
28	400	526	10673518.0967742	0.0106735	-733645	93
29	800	1063	37964609.0000000	0.0379646	-1541291	26
30	800	1058	37701474.6153846	0.0377015	-1578294	26
31	800	1076	36661267.0740741	0.0366613	-1664316	27
32	800	1049	41528325.5833333	0.0415283	-1652119	24

Dataset	Nodes	Edges	Time [ns]	Time [s]	Weight	Repetitions
33	1000	1300	56174018.5294118	0.0561740	-2089013	17
34	1000	1313	63600892.1333333	0.0636009	-1934208	15
35	1000	1328	59794412.0625000	0.0597944	-2229428	16
36	1000	1344	61854901.8125000	0.0618549	-2356163	16
37	2000	2699	236135664.0000000	0.2361357	-4811598	4
38	2000	2654	236604744.5000000	0.2366047	-4739387	4
39	2000	2652	234245281.0000000	0.2342453	-4717250	4
40	2000	2677	246662528.5000000	0.2466625	-4537267	4
41	4000	5360	912594104.0000000	0.9125941	-8722212	1
42	4000	5315	968720773.0000000	0.9687208	-9314968	1
43	4000	5340	899673136.0000000	0.8996731	-9845767	1
44	4000	5368	1003618763.0000000	1.0036188	-8681447	1
45	8000	10705	3698823422.0000000	3.6988234	-17844628	1
46	8000	10670	3665222853.0000000	3.6652229	-18798446	1
47	8000	10662	3684639303.0000000	3.6846393	-18741474	1
48	8000	10757	3829282036.0000000	3.8292820	-18178610	1
49	10000	13301	5780728292.0000000	5.7807283	-22079522	1
50	10000	13340	5819827165.0000000	5.8198272	-22338561	1
51	10000	13287	5624067894.0000000	5.6240679	-22581384	1
52	10000	13311	5843340473.0000000	5.8433405	-22606313	1
53	20000	26667	23888895940.0000000	23.8888959	-45962292	1
54	20000	26826	23449849119.0000000	23.4498491	-45195405	1
55	20000	26673	23745057886.0000000	23.7450579	-47854708	1
56	20000	26670	23544887011.0000000	23.5448870	-46418161	1
57	40000	53415	118890358713.0000000	118.8903587	-92003321	1
58	40000	53446	120358945675.0000000	120.3589457	-94397064	1
59	40000	53242	117938669012.0000000	117.9386690	-88771991	1
60	40000	53319	118288795158.0000000	118.2887952	-93017025	1
61	80000	106914	643469573106.0000000	643.4695731	-186834082	1
62	80000	106633	658110627498.0000000	658.1106275	-185997521	1
63	80000	106586	667819692588.0000000	667.8196926	-182065015	1
64	80000	106554	656982775681.0000000	656.9827757	-180793224	1
65	100000	133395	1113112826154.0000000	1113.1128262	-230698391	1
66	100000	133214	1112607324308.0000000	1112.6073243	-230168572	1
67	100000	133524	1131364649966.0000000	1131.3646500	-231393935	1
68	100000	133463	1110298442370.0000000	1110.2984424	-231011693	1

A.2 Risultati output Kruskal Union Find

Dataset	**Nodes**	**Edges**	**Time [ns]**	**Time [s]**	**Weight**	**Repetitions**
1	10	9	39597.6445230	0.0000396	29316	13292
2	10	11	45731.7479564	0.0000457	16940	19084
3	10	13	52688.4001881	0.0000527	-44448	17012
4	10	10	42247.7829541	0.0000422	25217	21272
5	20	24	104909.0138404	0.0001049	-32021	8309
6	20	24	104242.8465955	0.0001042	25130	8709
7	20	28	119604.0416225	0.0001196	-41693	7544
8	20	26	112864.1022189	0.0001129	-37205	8022
9	40	56	254291.8269286	0.0002543	-114203	3513
10	40	50	233312.8574386	0.0002333	-31929	3865
11	40	50	231675.4907063	0.0002317	-79570	3766
12	40	52	240167.8244032	0.0002402	-79741	3770
13	80	108	522599.6063830	0.0005226	-139926	1692
14	80	99	484509.5977260	0.0004845	-198094	1847
15	80	104	507783.7900114	0.0005078	-110571	1762
16	80	114	550375.7953980	0.0005504	-233320	1608
17	100	136	673492.1640212	0.0006735	-141960	1323
18	100	129	641796.8609898	0.0006418	-271743	1374
19	100	137	675071.3916031	0.0006751	-288906	1310
20	100	132	653543.9889135	0.0006535	-229506	1353
21	200	267	1401315.0873786	0.0014013	-510185	618
22	200	269	1411328.5925926	0.0014113	-515136	621
23	200	269	1405656.7834395	0.0014057	-444357	628
24	200	267	1397145.5862620	0.0013971	-393278	626
25	400	540	3128440.7256318	0.0031284	-1119906	277
26	400	518	3026085.7681661	0.0030261	-788168	289
27	400	538	3150672.3115942	0.0031507	-895704	276
28	400	526	3070144.2801418	0.0030701	-733645	282
29	800	1063	6750177.4843750	0.0067502	-1541291	128
30	800	1058	6740257.3953488	0.0067403	-1578294	129
31	800	1076	6847874.1349206	0.0068479	-1664316	126
32	800	1049	6690976.3100775	0.0066910	-1652119	129
33	1000	1300	8471990.3431373	0.0084720	-2089013	102
34	1000	1313	8510821.7623762	0.0085108	-1934208	101
35	1000	1328	8584549.0000000	0.0085845	-2229428	100
36	1000	1344	8699039.0918367	0.0086990	-2356163	98
37	2000	2699	18597894.1111111	0.0185979	-4811598	45
38	2000	2654	18327104.0652174	0.0183271	-4739387	46
39	2000	2652	18359813.4130435	0.0183598	-4717250	46
40	2000	2677	18485540.7111111	0.0184855	-4537267	45
41	4000	5360	39116529.0000000	0.0391165	-8722212	21

Dataset	Nodes	Edges	Time [ns]	Time [s]	Weight	Repetitions
42	4000	5315	39006028.9523810	0.0390060	-9314968	21
43	4000	5340	39317801.2857143	0.0393178	-9845767	21
44	4000	5368	39481449.6666667	0.0394814	-8681447	21
45	8000	10705	82560147.1000000	0.0825601	-17844628	10
46	8000	10670	82508146.9000000	0.0825081	-18798446	10
47	8000	10662	82175961.9000000	0.0821760	-18741474	10
48	8000	10757	83415242.0000000	0.0834152	-18178610	10
49	10000	13301	104905625.2500000	0.1049056	-22079522	8
50	10000	13340	104711810.7500000	0.1047118	-22338561	8
51	10000	13287	104559018.6250000	0.1045590	-22581384	8
52	10000	13311	104644913.2500000	0.1046449	-22606313	8
53	20000	26667	221066963.6666667	0.2210670	-45962292	3
54	20000	26826	221880369.0000000	0.2218804	-45195405	3
55	20000	26673	222298615.3333333	0.2222986	-47854708	3
56	20000	26670	221084166.6666667	0.2210842	-46418161	3
57	40000	53415	477325755.0000000	0.4773258	-92003321	1
58	40000	53446	478210336.0000000	0.4782103	-94397064	1
59	40000	53242	473483033.0000000	0.4734830	-88771991	1
60	40000	53319	478497888.0000000	0.4784979	-93017025	1
61	80000	106914	1209893768.0000000	1.2098938	-186834082	1
62	80000	106633	1192000776.0000000	1.1920008	-185997521	1
63	80000	106586	1210478783.0000000	1.2104788	-182065015	1
64	80000	106554	1196255366.0000000	1.1962554	-180793224	1
65	100000	133395	1537921847.0000000	1.5379218	-230698391	1
66	100000	133214	1536065944.0000000	1.5360659	-230168572	1
67	100000	133524	1551826803.0000000	1.5518268	-231393935	1
68	100000	133463	1557143509.0000000	1.5571435	-231011693	1

A.3 Risultati output Prim

Dataset	Nodes	Edges	Time [ns]	Time [s]	Weight	Repetitions
1	10	9	73384.4760509	0.0000734	29316	8873
2	10	11	85553.0883619	0.0000856	16940	10921
3	10	13	97357.0755650	0.0000974	-44448	9912
4	10	10	86931.5251812	0.0000869	25217	10762
5	20	24	246858.1311475	0.0002469	-32021	3965
6	20	24	218168.3059150	0.0002182	25130	4328
7	20	28	253380.6182658	0.0002534	-41693	3898
8	20	26	227881.9872597	0.0002279	-37205	4317
9	40	56	606313.1715686	0.0006063	-114203	1632

Dataset	**Nodes**	**Edges**	**Time [ns]**	**Time [s]**	**Weight**	**Repetitions**
10	40	50	561691.5469550	0.0005617	-31929	1757
11	40	50	566737.3930006	0.0005667	-79570	1743
12	40	52	576771.5316676	0.0005768	-79741	1721
13	80	108	1440989.3575581	0.0014410	-139926	688
14	80	99	1414840.6054795	0.0014148	-198094	730
15	80	104	1369863.4372414	0.0013699	-110571	725
16	80	114	1521327.6841294	0.0015213	-233320	649
17	100	136	1916219.3442308	0.0019162	-141960	520
18	100	129	1859230.5279851	0.0018592	-271743	536
19	100	137	1958658.7246094	0.0019587	-288906	512
20	100	132	1903094.2461832	0.0019031	-229506	524
21	200	267	4419799.4844444	0.0044198	-510185	225
22	200	269	4485446.8654709	0.0044854	-515136	223
23	200	269	4513123.7954545	0.0045131	-444357	220
24	200	267	4533419.1357466	0.0045334	-393278	221
25	400	540	10407676.1684211	0.0104077	-1119906	95
26	400	518	10170872.3711340	0.0101709	-788168	97
27	400	538	10533477.8085106	0.0105335	-895704	94
28	400	526	10292546.2500000	0.0102925	-733645	96
29	800	1063	23454574.3571429	0.0234546	-1541291	42
30	800	1058	23645769.2857143	0.0236458	-1578294	42
31	800	1076	23929648.9512195	0.0239296	-1664316	41
32	800	1049	23516296.5714286	0.0235163	-1652119	42
33	1000	1300	30235913.0606061	0.0302359	-2089013	33
34	1000	1313	30572566.4062500	0.0305726	-1934208	32
35	1000	1328	30896512.2500000	0.0308965	-2229428	32
36	1000	1344	30906809.1562500	0.0309068	-2356163	32
37	2000	2699	70873597.6428571	0.0708736	-4811598	14
38	2000	2654	69789933.6428571	0.0697899	-4739387	14
39	2000	2652	69632619.6428571	0.0696326	-4717250	14
40	2000	2677	70234522.8571429	0.0702345	-4537267	14
41	4000	5360	157046280.5000000	0.1570463	-8722212	6
42	4000	5315	156028482.8333333	0.1560285	-9314968	6
43	4000	5340	157181312.0000000	0.1571813	-9845767	6
44	4000	5368	156269539.5000000	0.1562695	-8681447	6
45	8000	10705	347440138.5000000	0.3474401	-17844628	2
46	8000	10670	348194793.0000000	0.3481948	-18798446	2
47	8000	10662	344711363.5000000	0.3447114	-18741474	2
48	8000	10757	347015763.0000000	0.3470158	-18178610	2
49	10000	13301	442112535.5000000	0.4421125	-22079522	2
50	10000	13340	446760593.5000000	0.4467606	-22338561	2

Dataset	Nodes	Edges	Time [ns]	Time [s]	Weight	Repetitions
51	10000	13287	444743531.5000000	0.4447435	-22581384	2
52	10000	13311	442997466.0000000	0.4429975	-22606313	2
53	20000	26667	974610952.0000000	0.9746110	-45962292	1
54	20000	26826	973262147.0000000	0.9732621	-45195405	1
55	20000	26673	974384105.0000000	0.9743841	-47854708	1
56	20000	26670	973670869.0000000	0.9736709	-46418161	1
57	40000	53415	2136591753.0000000	2.1365918	-92003321	1
58	40000	53446	2133940840.0000000	2.1339408	-94397064	1
59	40000	53242	2125702872.0000000	2.1257029	-88771991	1
60	40000	53319	2132769220.0000000	2.1327692	-93017025	1
61	80000	106914	4700060910.0000000	4.7000609	-186834082	1
62	80000	106633	4695625792.0000000	4.6956258	-185997521	1
63	80000	106586	4694558420.0000000	4.6945584	-182065015	1
64	80000	106554	4705053462.0000000	4.7050535	-180793224	1
65	100000	133395	6060222656.0000000	6.0602227	-230698391	1
66	100000	133214	6042919511.0000000	6.0429195	-230168572	1
67	100000	133524	6050503941.0000000	6.0505039	-231393935	1
68	100000	133463	6087769864.0000000	6.0877699	-231011693	1

A.4 Risultati quartetto Kruskal Naive

Nodes	Datasets	Avg Edges	Time [s]	Total execs	Execs per alg.
10	(1 2 3 4)	10.75	0.0000450	73364	18341
20	(5 6 7 8)	25.5	0.0001154	33316	8329
40	(9 10 11 12)	52.0	0.0002559	15452	3863
80	(13 14 15 16)	106.25	0.0008078	4912	1228
100	(17 18 19 20)	133.5	0.0010000	3988	997
200	(21 22 23 24)	268.0	0.0032157	1256	314
400	(25 26 27 28)	530.5	0.0108574	368	92
800	(29 30 31 32)	1061.5	0.0395373	100	25
1000	(33 34 35 36)	1321.25	0.0613981	64	16
2000	(37 38 39 40)	2670.5	0.2425361	16	4
4000	(41 42 43 44)	5345.75	0.9682446	4	1
8000	(45 46 47 48)	10698.5	3.7993357	4	1
10000	(49 50 51 52)	13309.75	5.8582550	4	1
20000	(53 54 55 56)	26709.0	23.7567957	4	1
40000	(57 58 59 60)	53355.5	119.1684887	4	1
80000	(61 62 63 64)	106671.75	656.8066593	4	1
100000	(65 66 67 68)	133399.0	1117.8413379	4	1

A.5 Risultati quartetto Kruskal Union Find

Nodes	Datasets	Avg Edges	Time [s]	Total execs	Execs per alg.
10	(1 2 3 4)	10.75	0.0000465	67456	16864
20	(5 6 7 8)	25.5	0.0001138	32320	8080
40	(9 10 11 12)	52.0	0.0002456	14872	3718
80	(13 14 15 16)	106.25	0.0005280	6784	1696
100	(17 18 19 20)	133.5	0.0006818	5264	1316
200	(21 22 23 24)	268.0	0.0014412	2452	613
400	(25 26 27 28)	530.5	0.0031814	1100	275
800	(29 30 31 32)	1061.5	0.0068905	500	125
1000	(33 34 35 36)	1321.25	0.0087078	392	98
2000	(37 38 39 40)	2670.5	0.0187046	180	45
4000	(41 42 43 44)	5345.75	0.0394314	84	21
8000	(45 46 47 48)	10698.5	0.0837343	40	10
10000	(49 50 51 52)	13309.75	0.1067242	28	7
20000	(53 54 55 56)	26709.0	0.2250284	12	3
40000	(57 58 59 60)	53355.5	0.4810824	4	1
80000	(61 62 63 64)	106671.75	1.2348689	4	1
100000	(65 66 67 68)	133399.0	1.5777018	4	1

A.6 Risultati quartetto Prim

Nodes	Datasets	Avg Edges	Time [s]	Total execs	Execs per alg.
10	(1 2 3 4)	10.75	0.0000868	39496	9874
20	(5 6 7 8)	25.5	0.0002398	16332	4083
40	(9 10 11 12)	52.0	0.0005888	6696	1674
80	(13 14 15 16)	106.25	0.0014418	2736	684
100	(17 18 19 20)	133.5	0.0019220	2060	515
200	(21 22 23 24)	268.0	0.0045010	884	221
400	(25 26 27 28)	530.5	0.0103793	384	96
800	(29 30 31 32)	1061.5	0.0238707	164	41
1000	(33 34 35 36)	1321.25	0.0310556	128	32
2000	(37 38 39 40)	2670.5	0.0708493	56	14
4000	(41 42 43 44)	5345.75	0.1580204	24	6
8000	(45 46 47 48)	10698.5	0.3501350	8	2
10000	(49 50 51 52)	13309.75	0.4499823	8	2
20000	(53 54 55 56)	26709.0	0.9886166	4	1
40000	(57 58 59 60)	53355.5	2.1820559	4	1
80000	(61 62 63 64)	106671.75	4.8049394	4	1
100000	(65 66 67 68)	133399.0	6.2187226	4	1