# Corso di Algoritmi Avanzati

# Laboratorio 3: Minimum Cut

10 luglio 2020


| | |
|---|---|
| **Busin Lorenzo** | 1237580 |
| **Tartaggia Nicolò** | 1237784 |
| **Voinea Ciprian** | 1237294 |

# Indice

# 1 Introduzione

Questo elaborato ha lo scopo di illustrare il lavoro svolto per il terzo homework del corso *Algoritmi Avanzati*.
L'homework ha come obiettivo quello di valutare le prestazioni dell'*Algoritmo di Karger* per il problema del *minimum cut* rispetto a quattro parametri:

1. Il tempo impiegato dalla procedura di *Full Contraction*;
2. Il tempo impiegato dall'algoritmo completo per ripetere la contrazione un numero sufficientemente alto di volte;
3. Il *discovery time*, ossia il momento in cui l'algoritmo trova per la prima volta il taglio di costo mimimo;
4. L'errore nella soluzione trovata rispetto al risultato ottimo.

Come per i precedenti homework, il linguaggio in cui sono stati implementati questi algoritmi è `Python`. Abbiamo deciso di utilizzare questo al contrario di altri linguaggi come `C++` o `Java` in quanto questa scelta ci ha permesso di utilizzare i *Jupyter Notebook* e di programmare utilizzando l'IDE *PyCharm* oppure con *Google Colab*.
Nella Sez. 3 sono presenti i risultati, in forma tabellare, che confrontano le performance richieste dalla consegna, calcolate sul dataset dato, contenente 40 grafi di esempio, di dimensione compresa tra 6 e 200 vertici e descritti in file `.txt`. Sono inoltre illustrati i grafici delle performance.
Il lavoro è stato suddiviso equamente tra i membri del gruppo collaborando all'implementazione dell'algoritmo ed effettuando una verifica finale del codice e dei risultati tramite *peer review* cercando di seguire una linea di sviluppo comune.

# 2 Algoritmo di Karger

```
KARGER(G, k):

    min = +∞
    for i = 1 to k:
        t = FULL_CONTRACTION(G)
        if t < min:
            t = min

    return min

FULL_CONTRACTION(G):

    for i = 1 to |V|−2:
        pick a random edge e = (u, v)
        merge u and v in a single node
        remove edges between u and v
        edit adjacency lists to point to the new node

    return |E| // the total number of edges remaining
```

L'algoritmo di *Karger* è un algoritmo randomizzato per la computazione del *minimum cut* di un multi-grafo connesso.
Un "*multi-grafo*" è un particolare tipo di grafo che ammette ripetizioni di lati al suo interno, ovvero questi possono apparire più di una volta (con *molteplicità* $\geq 1$).
L'idea su cui si basa l'algoritmo qui implementato, è quella di contrarre i vertici all'interno del multi-grafo fino a quando non ne restano solamente due, i quali rappresentano gli opposti del taglio.
Per "*contrarre*" si intende *unire* i due nodi estremi di un lato in maniera da averne solamente uno preservando il taglio minimo. Dopo averli uniti vanno aggiustate le vecchie liste di adiacenza, unite in una singola, in maniera che i vecchi puntatori ai nodi tolti siano stati correttamente sostituiti con riferimenti al nuovo nodo e che non ci siano *self-loops*.
Essendo un *algoritmo randomizzato*, ovvero utilizza un grado di casualità come parte della sua logica, la procedura di contrazione va eseguita $k$ volte, con $k$ calcolato in maniera da soddisfare la probabilità:

$$Pr(\text{"le k } FULL\_C. \text{ non accumulano la taglia del min cut"}) \leq (1 - \tfrac{2}{n^2})^k$$

Voglio ottenere qualcosa del tipo $\frac{1}{n^d}$ e quindi scelgo $k = d\frac{n^2}{2}\ln n$.
La logica che implementa la parte randomica in questo caso è la scelta del lato i cui nodi andranno contratti.
Possiamo quindi dire che l'algoritmo funziona con probabilità bassissima, che può però essere amplificata ripetendo questo processo molte volte, esattamente $k$.
Data l'alta complessità computazionale, abbiamo deciso di introdurre un tempo massimo di 60 secondi in modo da eseguire l'algoritmo su ciascuna istanza del dataset.

## 2.1 Strutture dati

A differenza degli altri homework, per migliorare le performance dell'algoritmo in questione, abbiamo deciso di utilizzare solamente la struttura dati `Graph`, implementata con i seguenti campi e metodi:

- `n_nodes`: numero di nodi del grafo;
- `n_edges`: numero di lati del grafo;
- `nodes`: dizionario con i nodi del grafo e le rispettive liste di adiacenza;
- `edges`: lista con i lati del grafo;
- `__init__()`: costruttore del grafo che imposta i precedenti campi a 0 o vuoti;
- `addEdge(v1, v2)`: metodo che dati due nodi aggiunge il lato che li unisce alla lista `edges` aggiornando le liste di adiacenza;
- `buildGraph(data)` metodo che riceve in input una matrice di adiacenza e, in base a questa, ne costruisce un grafo.

## 2.2 Funzioni

Le funzioni ausiliarie utilizzate per calcolare la soluzione dell'algoritmo di Karger sono le seguenti:

- `readInput(path)`: metodo che, dato in input il path del dataset, ne legge il contenuto e restituisce la matrice di adiacenza del grafo;
- `graphCopy(graph)`: metodo che riceve in input un oggetto di tipo `Graph` e ne restituisce una copia;
- `execute_alg(path)`: metodo che, dato in input il path del dataset, esegue la chiamata a `Karger(G)` calcolando i tempi richiesti nella consegna e confrontando i risultati ottenuti con quelli attesi.

## 2.3 Implementazione

La soluzione del problema è stata implementata nel seguente modo:

- a partire dalla lista di vertici nel file di input, viene costruito il grafo `graph`;
- viene calcolato il valore `k` tramite la formula `int((graph.n_nodes**2 / 2) * math.log(graph.n_nodes))`. Se `k` supera 10000 allora il suo valore lo fissiamo a tale soglia;
- per migliorare le performance della soluzione proposta, abbiamo deciso di creare in anticipo le copie dei grafi per ciascuna delle `k` volte che verrà eseguito `fullContraction(G)`. La limitazione del valore di $k$ a 10000 è dovuta all'elevato costo della creazione delle $k$ copie di `graph`;

- viene effettuata la chiamata a `Karger(G,k)`, nel quale per `k` volte viene chiamato `fullContraction(G)`. I risultati di ciascuna chiamata vengono confrontati e viene salvato quello minimo, inoltre vengono registrati i tempi che sono presentati in formato tabellare nella Sez. 3. Se il tempo di esecuzione supera quello massimo fissato, allora viene tornata la soluzione trovata fino a quel momento;
- per ciascuna chiamata a `fullContraction(G)`, questo metodo sceglie in maniera randomica un lato presente all'interno del grafo passato in input e unisce i vertici che ne rappresentano gli estremi in un singolo vertice. Per ciascuno degli altri vertici presenti all'interno del grafo si modificano le liste di adiacenza e aggiornano i riferimenti. Vengono inoltre diminuiti i contatori `n_nodes` e `n_edges` all'interno della struttura `Graph`. Questo metodo ritorna il numero rimanente di lati all'interno del grafo dopo aver eliminato $|V| - 2$ nodi;
- i risultati vengono messi in coda alla lista e questo procedimento viene ripetuto per i rimanenti file in input non ancora valutati.

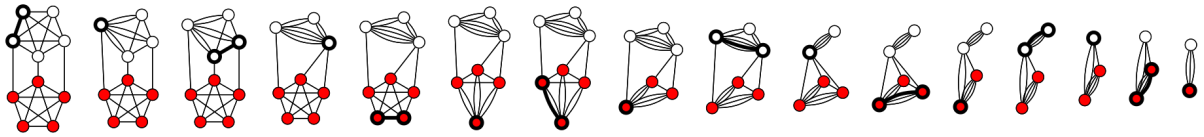

Figura 1: Esempio di esecuzione dell'algoritmo di Karger e Full Contraction

## 2.4 Complessità

La complessità della subroutine `fullContraction` è $O(n^2)$ e questo porta la complessità di `Karger`, e quindi di tutto l'algoritmo, ad essere $O(n^4 \log n)$ per ciascuna istanza in analisi. In particolare:

- nella subroutine di `FullContraction` la complessità è dovuta dal primo ciclo `for` che ha complessità $O(n)$ e ciascuna contrazione ha complessità $O(n)$. In totale, quindi, `FullContraction` ha complessità $O(n^2)$;
- la complessità di `Karger` è dovuta dalla complessità della precedente procedura `FullContraction`, ripetuta $k$ volte, con una complessità totale di $O(n^4 \log n)$.

# 3 Risultati

## 3.1 Tabella dei risultati

| Istanza | Soluzione | Tempo(s) | Tempo Full Contraction(s) | Discovery time(s) | Errore(%) |
|---|---|---|---|---|---|
| input_1_6.txt | 2 | 0.00231 | 0.00007 | 0.00024 | 0.00 |
| input_2_6.txt | 1 | 0.00174 | 0.00005 | 0.00065 | 0.00 |
| input_3_6.txt | 3 | 0.00180 | 0.00006 | 0.00018 | 0.00 |
| input_4_6.txt | 4 | 0.00217 | 0.00007 | 0.00014 | 0.00 |
| input_5_10.txt | 4 | 0.03442 | 0.00030 | 0.00072 | 0.00 |
| input_6_10.txt | 3 | 0.01447 | 0.00013 | 0.00095 | 0.00 |
| input_7_10.txt | 2 | 0.01540 | 0.00013 | 0.00062 | 0.00 |
| input_8_10.txt | 1 | 0.01490 | 0.00013 | 0.00013 | 0.00 |
| input_9_25.txt | 7 | 1.59042 | 0.00158 | 0.00887 | 0.00 |
| input_10_25.txt | 6 | 1.56295 | 0.00155 | 0.04115 | 0.00 |
| input_11_25.txt | 8 | 2.00112 | 0.00199 | 0.00893 | 0.00 |
| input_12_25.txt | 9 | 1.89574 | 0.00188 | 0.00201 | 0.00 |
| input_13_50.txt | 15 | 60.00233 | 0.01549 | 0.10327 | 0.00 |
| input_14_50.txt | 16 | 60.01616 | 0.01579 | 0.13614 | 0.00 |
| input_15_50.txt | 14 | 60.00795 | 0.01347 | 0.04197 | 0.00 |
| input_16_50.txt | 10 | 60.01186 | 0.01411 | 0.02720 | 0.00 |
| input_17_75.txt | 19 | 60.01799 | 0.04756 | 0.12836 | 0.00 |
| input_18_75.txt | 15 | 60.04845 | 0.05328 | 0.15721 | 0.00 |
| input_19_75.txt | 18 | 60.03456 | 0.05216 | 0.40870 | 0.00 |
| input_20_75.txt | 16 | 60.01651 | 0.05090 | 1.61908 | 0.00 |
| input_21_100.txt | 22 | 60.11853 | 0.12656 | 0.74940 | 0.00 |
| input_22_100.txt | 23 | 60.05595 | 0.12670 | 0.87588 | 0.00 |
| input_23_100.txt | 19 | 60.05745 | 0.13680 | 0.56977 | 0.00 |
| input_24_100.txt | 24 | 60.05831 | 0.13228 | 0.61221 | 0.00 |
| input_25_125.txt | 34 | 60.23190 | 0.30267 | 2.83151 | 0.00 |
| input_26_125.txt | 29 | 60.13841 | 0.27212 | 4.71717 | 0.00 |
| input_27_125.txt | 36 | 60.06676 | 0.30336 | 0.91342 | 0.00 |
| input_28_125.txt | 31 | 60.10401 | 0.25576 | 7.58857 | 0.00 |
| input_29_150.txt | 37 | 60.17212 | 0.60780 | 3.03944 | 0.00 |
| input_30_150.txt | 35 | 60.26632 | 0.52405 | 0.59696 | 0.00 |
| input_31_150.txt | 41 | 60.51738 | 0.54033 | 2.94316 | 0.00 |
| input_32_150.txt | 39 | 60.21118 | 0.56803 | 26.88796 | 0.00 |
| input_33_175.txt | 42 | 60.69174 | 0.90584 | 2.62535 | 0.00 |
| input_34_175.txt | 45 | 60.81718 | 0.90772 | 45.62800 | 0.00 |
| input_35_175.txt | 53 | 60.90280 | 1.14911 | 10.99047 | 0.00 |
| input_36_175.txt | 43 | 60.37577 | 0.86251 | 8.02117 | 0.00 |

| Istanza | Soluzione | Tempo(s) | Tempo Full Contraction(s) | Discovery time(s) | Errore(%) |
|---|---|---|---|---|---|
| input_37_200.txt | 54 | 60.65795 | 1.68494 | 28.33442 | 0.00 |
| input_38_200.txt | 52 | 61.38803 | 1.42763 | 18.03374 | 0.00 |
| input_39_200.txt | 51 | 60.17441 | 1.50436 | 7.57775 | 0.00 |
| input_40_200.txt | 61 | 61.56819 | 1.81083 | 12.76823 | 0.00 |

## 3.2 Grafico di confronto dei tempi di esecuzione

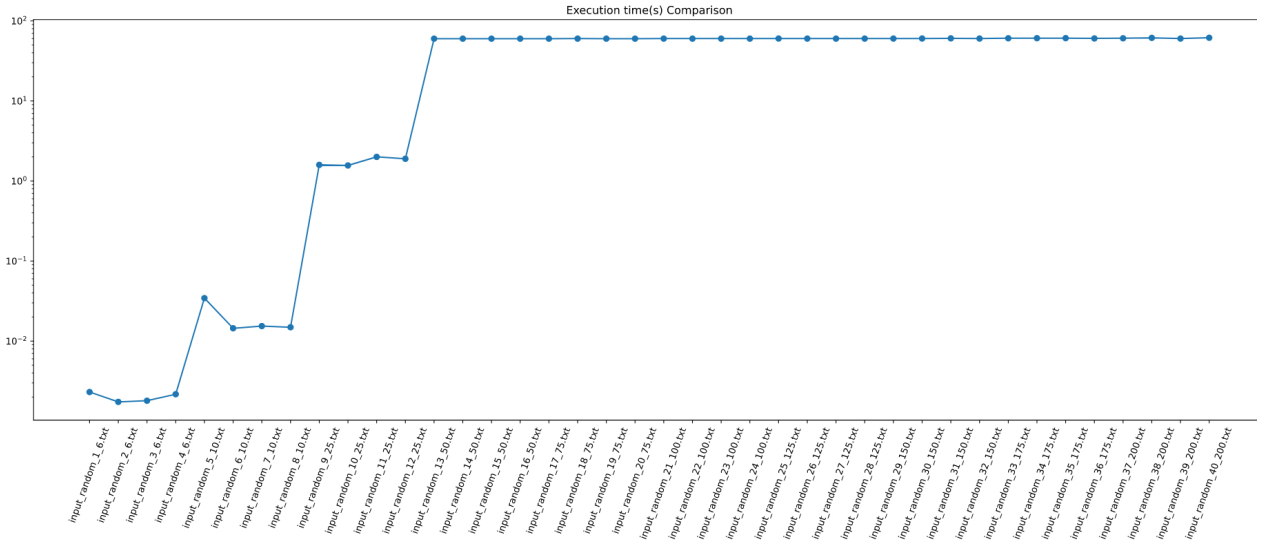


Figura 2: Confronto dei tempi di esecuzione

## 3.3 Grafico di confronto dei tempi di Full Contraction

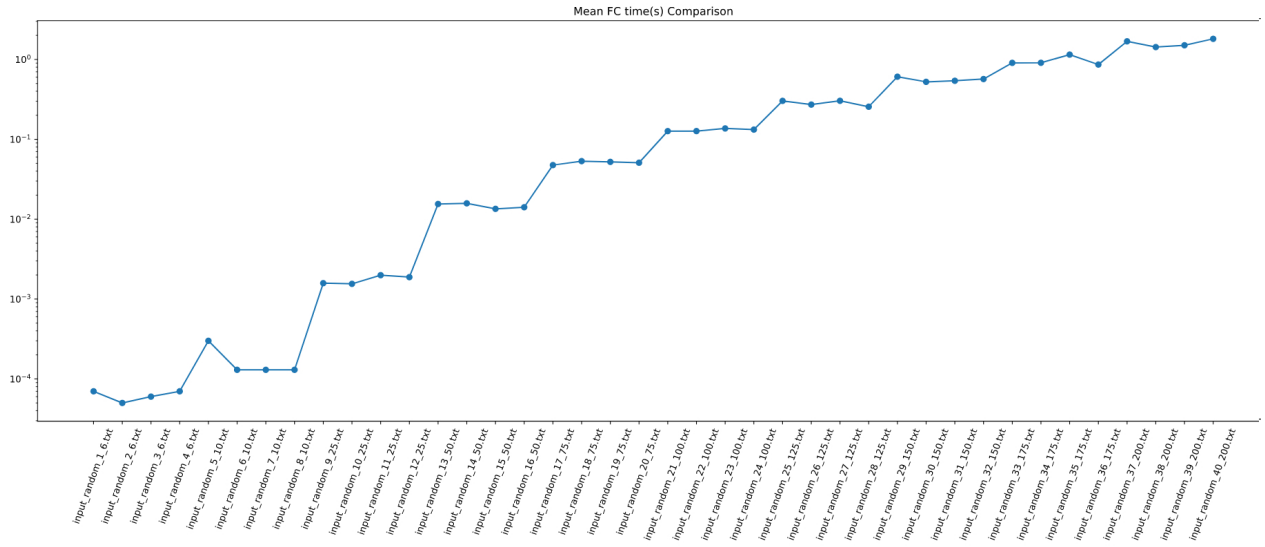


Figura 3: Confronto dei tempi di Full Contraption

## 3.4 Grafico di confronto dei Discovery Time

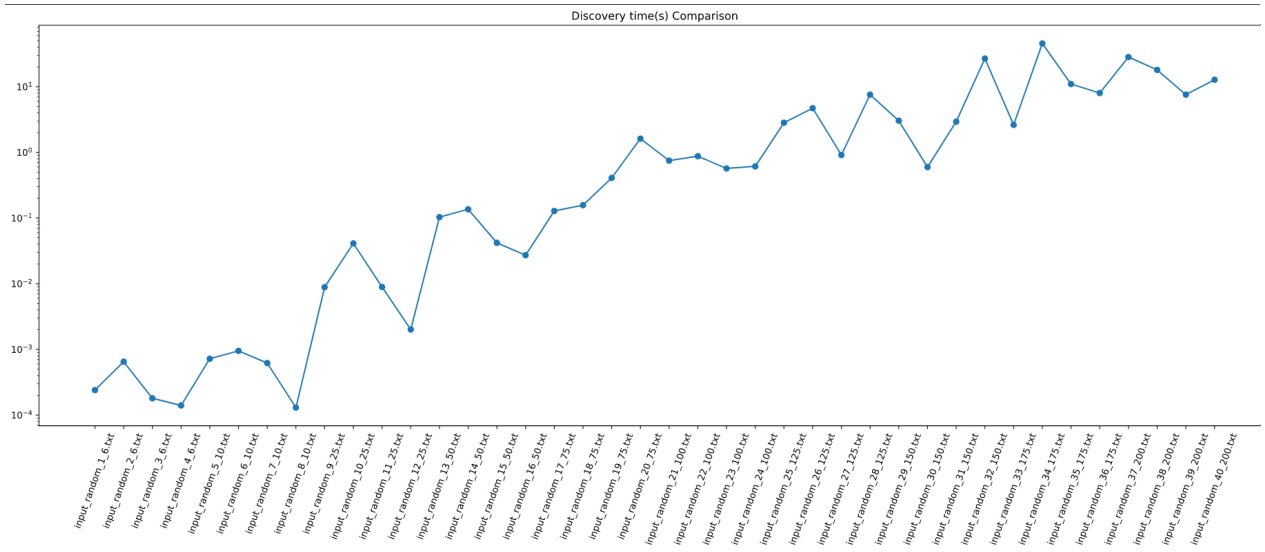


Figura 4: Confronto dei Discovery Time

## 3.5 Grafico di confronto dei Discovery Time rispetto ai tempi totali

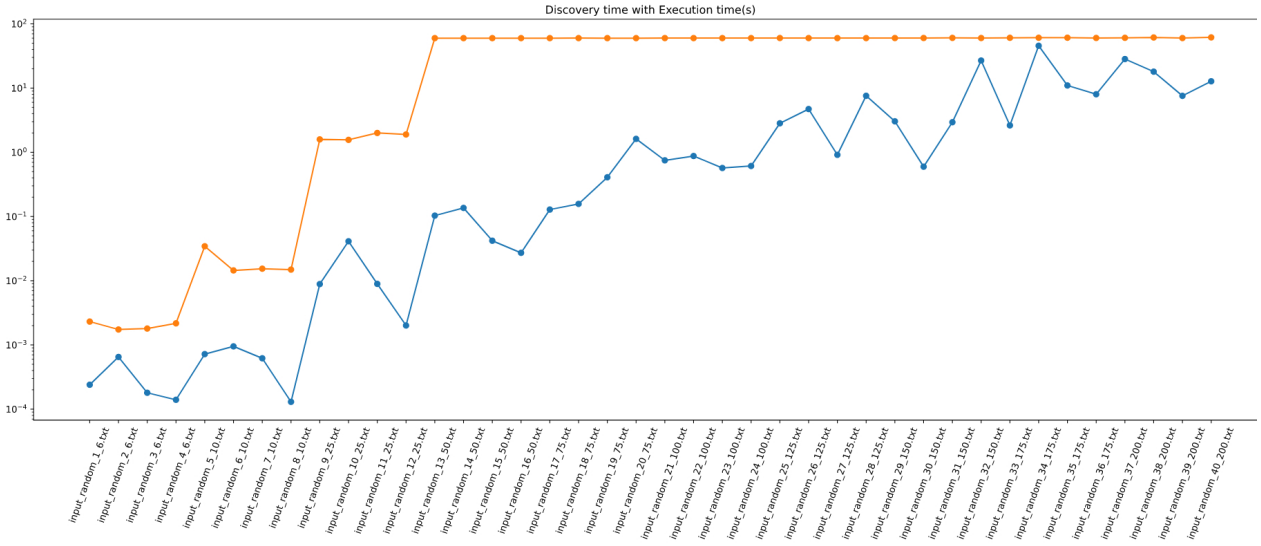


Figura 5: Confronto dei Discovery Time rispetto ai tempi totali di esecuzione

## 3.6 Grafico di confronto delle percentuali di errore

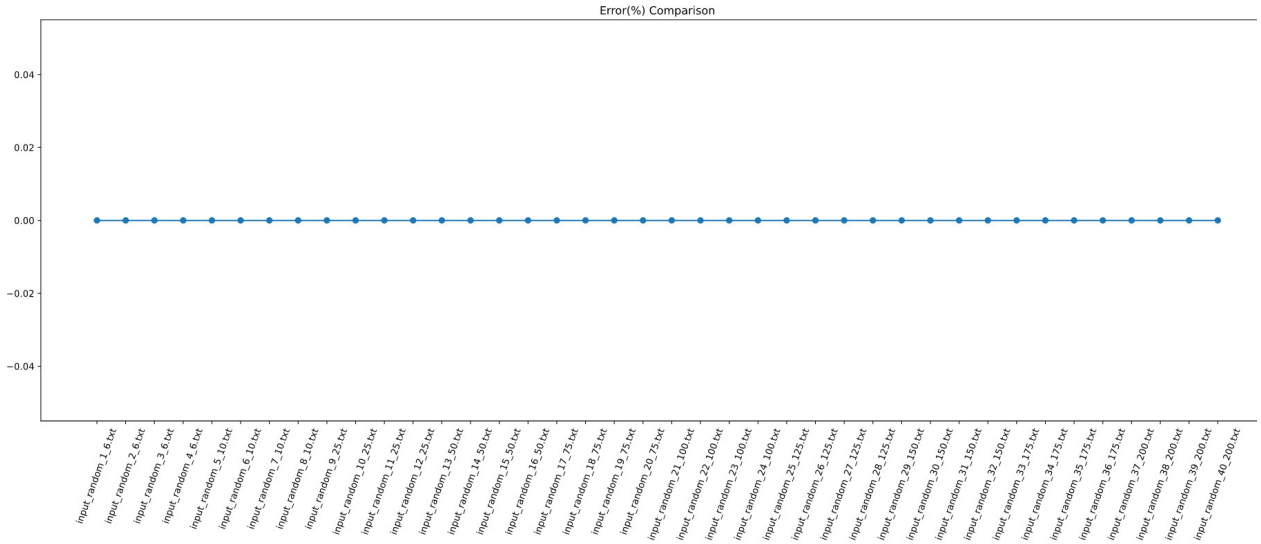


Figura 6: Confronto delle percentuali di errore

# 4 Conclusione

La complessità asintotica di `FullContraction` è $O(n^2)$ e quindi all'aumentare della dimensione dell'input il tempo di una singola `FullContraction` aumenta e, proporzionalmente, aumenta il tempo totale di esecuzione, cioè la ripetizione di $k$ volte della procedura. Ottenuti questi risultati si può affermare che rispettano la complessità attesa della procedura `FullContraction` e, di conseguenza, quella di `Karger`.
Il grafico in Fig. 3.3 indica i tempi medi dell'esecuzione della procedura `FullContraction` per ciascuna istanza in input.
Il grafico in Fig. 3.5 mostra il confronto tra il discovery time con il tempo di esecuzione complessivo per ognuno dei grafi nel dataset. Si può notare che all'aumentare della dimensione delle istanze di input e di conseguenza all'aumentare del tempo massimo di esecuzione dell'algoritmo il discovery time aumenta, però questo dato presenta un'elevata variabilità dovuta alla scelta casuale del lato effettuata dall'algoritmo.
Tutti i risultati ottenuti dall'algoritmo hanno una percentuale di errore di 0% in quanto la procedura riesce a trovare la soluzione esatta entro i tempi prefissati.