



**UNIVERSITÀ DEGLI STUDI DI ROMA  
TOR VERGATA  
FACOLTÀ DI INGEGNERIA**

CORSO DI LAUREA MAGISTRALE IN INGEGNERIA  
INFORMATICA

A.A. 2016/2017

**Algoritmi per il Web**

Evaluating a sublinear-time algorithm for the minimum  
spanning tree problem

**DOCENTE**

Evgeniia Perekhodko

**STUDENTI**

Gabriele Belli  
Daniele Capri



# Indice

<b>1</b>	<b>Introduzione</b>	<b>1</b>
1.1	Presentazione . . . . .	1
1.2	Obiettivi . . . . .	1
1.3	Stato dell'arte . . . . .	2
<b>2</b>	<b>BFS</b>	<b>3</b>
2.1	Implementazione . . . . .	4
<b>3</b>	<b>L'algoritmo</b>	<b>7</b>
3.1	Approssimazione del peso del MST . . . . .	7
3.1.1	Calcolo $d^*$ . . . . .	8
3.2	Approssimazione del numero delle componenti connesse . . . . .	9
<b>4</b>	<b>Test</b>	<b>12</b>
4.1	Intervallo di confidenza . . . . .	12
4.2	Progettazione dei test . . . . .	13
4.3	Dataset . . . . .	14
4.4	Risultati . . . . .	14
4.4.1	Test 1 . . . . .	15
4.4.2	Test 2 . . . . .	16
4.4.3	Test 3 . . . . .	18

4.4.4 Test 4 . . . . .	20
4.5 Conclusioni . . . . .	21
4.6 Sviluppi futuri . . . . .	22
<b>Bibliografia</b>	<b>23</b>

# Capitolo 1

## Introduzione

### 1.1 Presentazione

Il problema del minimo albero ricoprente, ricorre in numerosi ambienti. Alcuni esempi possono essere il design delle reti, il clustering, la generazione di ipotesi evolutive ecc.. In molti casi i grafi che si vuole studiare, tendono ad essere molto estesi (si pensi alle reti associate ai social network, allo studio di percorsi stradali legati al traffico in una nazione, traffico in rete o alle transazioni finanziarie) per questo motivo gli algoritmi lineari, che sono di norma considerati l'eccellenza, possono risultare non sufficientemente rapidi.

### 1.2 Obiettivi

In questo paper vogliamo presentare un'implementazione dell'algoritmo di complessità sublineare mostrato in [1] che tende ad approssimare il peso del minimo albero ricoprente di un grafo sotto determinate assunzioni:

- Gli archi dei grafi devono essere pesati con un peso che oscilla tra 1 ed  $W$
- I grafi devono essere connessi
- Gli archi dei grafi devono essere non diretti
- I grafi devono essere finiti
- I grafi non devono essere multi grafi

- L'implementazione dei grafi dovrebbe essere rappresentata con liste di adiacenza
- I grafi possono contenere dei self-loops

Una volta presentata l'implementazione dell'algoritmo [1] presenteremo i test eseguiti per mostrare caratteristiche sia funzionali che non funzionali. In particolare cercheremo di indagare sia sull'efficacia dell'approssimazione sia sull'effettiva sub-linearità dell'algoritmo.

## 1.3 Stato dell'arte

L'algoritmo presentato in [1] è un caposaldo nella letteratura. Nell'articolo [2] (proposto precedentemente a [1]) viene mostrato un algoritmo molto simile ad [1] applicato ad un problema analogo che sembra comunque andare nella stessa direzione. In ogni caso le prestazioni di [1] sono state dimostrate matematicamente, però in nessun articolo l'algoritmo è stato implementato e testato in maniera scientifica per sondarne l'efficacia e la sub-linearità.

# Capitolo 2

## BFS

Uno dei problemi fondamentali della teoria dei grafi è quello della visita di tutti i vertici di un grafo. Tra gli algoritmi di ricerca più utilizzati ci sono il BFS e il DFS. La visita in ampiezza o breadth-first-search (BFS) di un grafo dato un vertice sorgente  $s$  consiste nell' esplorazione sistematica di tutti i vertici raggiungibili da  $s$  in modo tale da esplorare tutti i vertici che hanno distanza  $k$  prima di iniziare a scoprire quelli che hanno distanza  $k+1$ . L'idea di base è quella di tenere traccia dello stato (già scoperto, appena scoperto, ancora da scoprire) di ogni vertice, “colorandolo” di un colore diverso. I colori possibili di un vertice possono essere:

- BIANCO: vertice ancora non scoperto.
- GRIGIO: vertice scoperto ed appartenente alla frontiera, ovvero adiacente al nodo.
- NERO: vertice per cui si è terminata la visita.

Per rappresentare lo stato dei vertici su evidenziato, si è deciso di utilizzare nel codice le seguenti strutture dati:

- LinkedList: ovvero una coda FIFO, nella quale vengono inseriti i vertici adiacenti al nodo che si sta prendendo in considerazione e dai quali successivamente si cercheranno i rispettivi vicini.
- HashSet: nella quale vengono inseriti i vertici visitati.

## 2.1 Implementazione

L'implementazione dell'algoritmo BFS che nell'algoritmo [1] è utilizzato per stimare il numero di componenti connessi è suddiviso in due step. La prima fase dell'algoritmo, ovviamente, è quella dell'inizializzazione.

```

1 public void initialize(int n){
2     this.rootNode=n;
3     numberOfVisitedVertices++;
4     degreeRootNode = graph.degree(n);
5     if(degreeRootNode>d_star){
6         dMajorStar = true;
7     }
8     q.add(n);
9     visited.add(n);
10}

```

Il nodo dalla quale si è deciso di far partire la visita in ampiezza viene aggiunto nella coda e viene marcato come visitato. Un ulteriore controllo che viene effettuato è quello sul grado di questo vertice, la quale importanza è legata all'algoritmo [1] e che vedremo in seguito. Se il grado del vertice è maggiore di un parametro passato nel momento della creazione dell'oggetto BFS la variabile dMajorStar è settata a true.

```

1 public void firstStep(){
2
3     int node;
4     node = q.remove();
5     Iterable<Edge> edge = graph.adj(node);
6     Iterator<Edge> it = edge.iterator();
7     while(it.hasNext()){
8         int sourceNode = it.next().other(node);
9         int degreeSourceNode = graph.degree(sourceNode);
10        if(degreeSourceNode>d_star){
11            dMajorStar = true;
12        }
13        if(!visited.contains(sourceNode)){
14            visited.add(sourceNode);
15            numberOfVisitedVertices++;
16            numberOfVisitedEdge++;
17        }
18    }
19}

```

```

17         q.addFirst(sourceNode);
18     }

```

Il primo step della BFS prevede la visita a partire dal nodo che in fase di inizializzazione è stato inserito nella coda. Questo vertice viene prima rimosso dalla FIFO e successivamente vengono ottenuti tutti gli archi adiacenti a questo. Per ogni vertice adiacente viene verificato che il grado non superi il valore di `d_star` passato come parametro e se questo vertice non è stato ancora visitato viene marcato appunto come "visitato" e aggiunto alla coda, successivamente vengono incrementati il contatore dei numeri di vertici visitati e il numero di archi visitati.

```

1  public int nextStep(){
2      int numberOfVisited = 2* numberOfVisitedEdge;
3
4      if (numberOfVisited == 0){
5          return 1;
6      }
7      while (!q.isEmpty() && numberOfVisitedEdge < numberOfVisited){
8          int node = q.getFirst();
9          Iterable<Edge> edge = graph.adj(node);
10         Iterator<Edge> it = edge.iterator();
11         while (it.hasNext() && numberOfVisitedEdge <
12             numberOfVisited){
13             int sourceNode = it.next().other(node);
14             if (!visited.contains(sourceNode))
15             {
16                 visited.add(sourceNode);
17                 numberOfVisitedVertices++;
18                 numberOfVisitedEdge++;
19                 q.add(sourceNode);
20             }
21         }
22         if (it.hasNext()==false){
23             q.remove();
24         }
25     }
26     if (q.isEmpty()){
27         return 1;
28     }

```

```
29         return 0;
30     }
```

Nel secondo step della BFS vengono visitati un numero di archi pari a quelli visitati nel step precedente. Se questo permette di completare la visita del grafo viene restituito l'intero 1 altrimenti viene restituito 0. Il comportamento del secondo step è molto simile al primo; viene controllato inoltre nel ciclo while se la coda FIFO si è svuotata o meno, questo perché solamente nel caso in cui questa sia vuota l'algoritmo si può considerare concluso. Un'altra differenza rispetto al primo step e imposta dal [1] sta anche nel fatto, come indicato in precedenza, che debbano essere fatti al massimo il numero di visite specificato nella variabile `numberOfVisited`. Ovviamente se il valore di tale variabile è uguale a 0, significa che il vertice visitato nello step precedente non ha vicini e pertanto la visita si può considerare come conclusa.

# Capitolo 3

## L'algoritmo

### 3.1 Approssimazione del peso del MST

L'algoritmo [1] per il corretto funzionamento, necessita solamente del grafo e del valore epsilon che rappresenta l'errore relativo.

```
1
2     ...
3
4     int dStar = GraphUtils.computateDStar(graphs.get(w-1), epsilon);
5     int w_star = (int) (4 * ((w) / epsilon));
6
7
8     for (int i = 0; i < w-1; i++) {
9
10         ApproxNumberConnectedComponent ancc =
11             newApproxNumberConnectedComponent(
12                 graphs.get(i), dStar, epsilon, w_star);
13         sum = ancc.approxNumberConnectedComponents();
14
15         total = total + sum;
16     }
17
18     double v = graphs.get(w-1).V() - w + total;
19
20     ...
21
```

Grazie all'errore relativo è possibile calcolare d\_star e w\_star che vengono passati al metodo che è il cuore dell'algoritmo stesso. La caratteristica fondamentale dell'algoritmo su evidenziato è il fatto che al metodo deve essere passato anche un grafo ottenuto dal grafo di partenza al quale devono essere rimossi gli archi che hanno peso superiore a i; dove i rappresenta una variabile che va da 1 a w-1 e dove w è il peso massimo degli archi che costituiscono il grafo. Il metodo ritorna un valore v che è l'obiettivo stesso dell'algoritmo, ovvero il calcolo, in tempo sub-lineare, del peso della MST.

Dal teorema 6 contenuto in [1], è evidenziato il fatto:

$$|v - t| < \varepsilon * t$$

Con probabilità di almeno  $3/4$ ; e dove t rappresenta il valore esatto del peso della MST ed  $\varepsilon$  l'errore relativo. Si è deciso inoltre in accordo a quanto scritto in [1] ed in modo da non appesantire l'algoritmo con computazione non necessaria al fine della valutazione di sub-linearità di preparare un'ArrayList nel quale sono contenuti a partire dallo stesso grafo, tutti i suoi sotto-grafi con peso via via crescente; quindi con il comando **graphs.get(3)** verrà ritornato il grafo, a partire dall'originale, a cui sono stati rimossi tutti gli archi con peso maggiore di 4. Ovviamente l'istruzione **graphs.get(w-1)** restituirà il grafo originale.

### 3.1.1 Calcolo d\*

Il calcolo del dStar segue un preciso algoritmo, vengono scelti esattamente n vertici, dove n è uguale al valore di C (un intero grande a piacere) fratto epsilon. Tra questi vertici si ottiene il grado maggiore, tale grado rappresenta il nostro dStar.

```

1 public static int computateDStar(EdgeWeightedGraph g, double epsilon) {
2     Random random = new Random();
3     int dStar = 0;
4     int r = (int) Math.ceil(Constants.BIG_INT_FOR_D_STAR / epsilon);
5     for (int i = 0; i < r; i++) {
6         int randomNum = random.nextInt(g.V());
7         if (g.degree(randomNum) > dStar) {
8             dStar = g.degree(randomNum);
9     }
10 }
```

## 3.2 Approssimazione del numero delle componenti connesse

Questo algoritmo rappresenta il cuore stesso di quanto dimostrato da Chazelle, vengono scelti  $r$  vertici del grafo e da ciascuno di questi viene fatta partire una visita BFS. Inizialmente il valore della moneta è impostato a true, quindi l'algoritmo procede con il lancio della moneta **coin.flip()**, successivamente se il valore ritornato è 1, il numero di vertici visitati è minore di  $w$  (valore passato come parametro ovvero `w_star`) e il valore del booleano calcolato nella BFS è false si procede con l'algoritmo, altrimenti si abortisce la BFS e si passa al vertice successivo che è contenuto nell'ArrayList `vertices`. Qualora si superassero le tre condizioni su evidenziate si esegue un nuovo step della BFS e se questo permettesse di terminare la BFS si incrementa del valore opportuno la variabile `betaSum`. Nel caso invece il nuovo step non permette di completare la visita della BFS viene rilanciata la moneta e ricontrollate le tre condizioni, se queste non vengono superate si abortisce anche in questo caso la BFS e si procede con il nuovo nodo. Terminato tale procedimento per tutti i vertici si restituisce, nel modo descritto, la variabile `c`.

Dal teorema 6 contenuto in [1], è evidenziato il fatto:

$$|c - d| < \varepsilon * d$$

Con probabilità di almeno  $3/4$ ; e dove  $d$  rappresenta il valore esatto delle componenti connesse del grafo ed  $\varepsilon$  l'errore relativo.

```

1  public double approxNumberConnectedComponents() {
2
3      Random random = new Random();
4      Coin coin = new Coin();
5
6      ArrayList<Integer> vertices = new ArrayList<Integer>();
7      boolean flipAgain = true;
8      int endBFS;
9
10     double r = (((1 / (epsilon * epsilon))) );
11
12     for (int i = 0; i < r; i++) {
13         int randomNum = random.nextInt(g.V());

```

```

14         vertices.add(randomNum);
15     }
16
17     for (int i = 0; i < r; i++) {
18
19         coin.setNumberFlip(0);
20         flipAgain = true;
21         BFS bfs = new BFS(g, dStar);
22         bfs.initialize(vertices.get(i));
23         bfs.firstStep();
24
25         while (flipAgain) {
26
27             int x = coin.flip();
28
29             if (x == 1 && bfs.getNumberOfVisitedVertices() < w &&
30                 !bfs.isdMajorStar()) {
31
32                 endBFS = bfs.nextStep();
33
34                 if (endBFS == 1) {
35                     flipAgain = false;
36                     if (bfs.getNumberOfVisitedEdge() == 0) {
37                         betaSum = betaSum + 2.0;
38                     } else {
39                         double tempBeta =
40                             ((bfs.getDegreeRootNode() * Math.pow(2.0,
41                                         coin.getNumberFlip())))
42                             / bfs.getNumberOfVisitedEdge());
43
44                         betaSum = betaSum + tempBeta;
45                     }
46
47                 } else {
48                     flipAgain = false;
49                 }
50             }
51         }
52         double c;

```

```
53     c = (g.V() / (2*r));  
54     c = (c * betaSum);  
55  
56     return c;  
57 }
```

# Capitolo 4

## Test

Una volta implementato l'algoritmo possiamo passare alla verifica e validazione del codice. Nella letteratura non abbiamo trovato alcun documento che approfondisse in maniera pratica i concetti espressi e dimostrati a livello matematico in [1]. Ci concentreremo principalmente su due ipotesi:

- L'algoritmo è sub-lineare
- L'algoritmo approssima un valore entro un certo livello di errore

### 4.1 Intervallo di confidenza

Tutti i risultati ottenuti dalle nostre simulazioni sono basati sul teorema del limite centrale e sulla sua relazione con la stima dell'intervallo di confidenza. Vogliamo restituire un intervallo che dia un metro di giudizio per capire quanto la stima sia precisa, invece di dare una stima puntuale; questo perché le simulazioni parallele tendono a produrre stime statistiche (come la media) che sono intrinsecamente incerte. Affinché i risultati siano quanto più possibile vicini alla realtà (quindi con un campione in input variegato) ma allo stesso tempo possano essere tra loro comparabili abbiamo scelto di generare dei grafi random invece di utilizzare data-set preesistenti. La generazione del grafo avviene con delle componenti che rimangono costanti (come il numero di nodi, il grado medio, il tipo di grafo ecc..) e componenti che risultano invece essere randomiche (la connessione tra 2 nodi, il grado di un determinato vertice, il coefficiente di clusterizzazione). Possiamo costruire un intervallo di confidenza con dimensioni scelte da noi (avendo un generatore di grafi e quindi un campione di cardinalità a nostro

piacere). Partendo dalla formula:

$$w = \frac{t_{\infty}^* s}{\sqrt{n - 1}}$$

Dove  $t_{\infty}^*$  è la distribuzione inversa di una normale standard con parametro  $1 - \alpha$ ,  $s$  la deviazione standard del parametro da monitorare ed  $n$  la cardinalità del campione.

Estrapolando  $n$  dalla formula si ottiene:

$$n = \left\lceil \left( \frac{t_{\infty}^* s}{w} \right)^2 \right\rceil + 1$$

Non siamo a conoscenza ne di  $w$  ne di  $s$  poiché sono valori ottenuti a run-time, quindi, per dare una stima possiamo usare come variabile il rapporto  $w/s$  che dà idea di quanto l'intervallo sia esteso rispetto alla deviazione standard.  $n$  può essere scelto quindi in base alla normale  $t_{\infty}^*$  ed al rapporto  $w/s$ . Per eseguire i nostri test abbiamo utilizzato i seguenti parametri  $\alpha = 0.1$  e  $w/s = 0.25$  che danno vita ad un campione di 44 grafi per ogni stima che viene effettuata. Nei grafi in cui viene messo a confronto l'andamento di due distribuzioni diverse verrà mostrato esclusivamente il centro dell'intervallo onde evitare confusione.

## 4.2 Progettazione dei test

Come già introdotto in precedenza cercheremo di testare (sotto le varie assunzioni elencate nell'introduzione) due ipotesi:

- L'algoritmo è sub-lineare
- L'algoritmo approssima un valore entro un certo livello di errore

Per valutare la prima metteremo a confronto l'algoritmo proposto in [1] con un algoritmo lineare (Kruskal o Prim); affinchè l'algoritmo sia sublineare si dovrà evincere che asintoticamente [1] tende ad avere tempi più bassi di un algoritmo lineare. Nel secondo caso eseguiremo i due algoritmi per testare se l'approssimazione effettuata da [1] rientra nell'intervallo dell'errore a livello asintotico, tenendo conto del teorema del limite centrale (almeno il 75% dei valori rientra nell'intervallo). Per verificare che queste ipotesi siano sempre valide controlleremo che questi aspetti siano validi in varie tipologie di grafo. Per sperimentare le precedenti due ipotesi daremo luogo a vari test:

**Test 1)** Andamento del tempo di risposta dell'algoritmo in base agli archi.

**Test 2)** Andamento dei risultati ottenuti dall'algoritmo in base agli archi.

**Test 3)** Andamento dei risultati ottenuti dall'algoritmo in base agli archi per varie tipologie di grafo.

Una volta dimostrate le ipotesi tramite i test sopra esposti mostreremo alcuni aspetti interessanti dell'algoritmo per capirne meglio il comportamento.

**Test 4)** Andamento dei risultati ottenuti dall'algoritmo in base al peso massimo degli archi.

## 4.3 Dataset

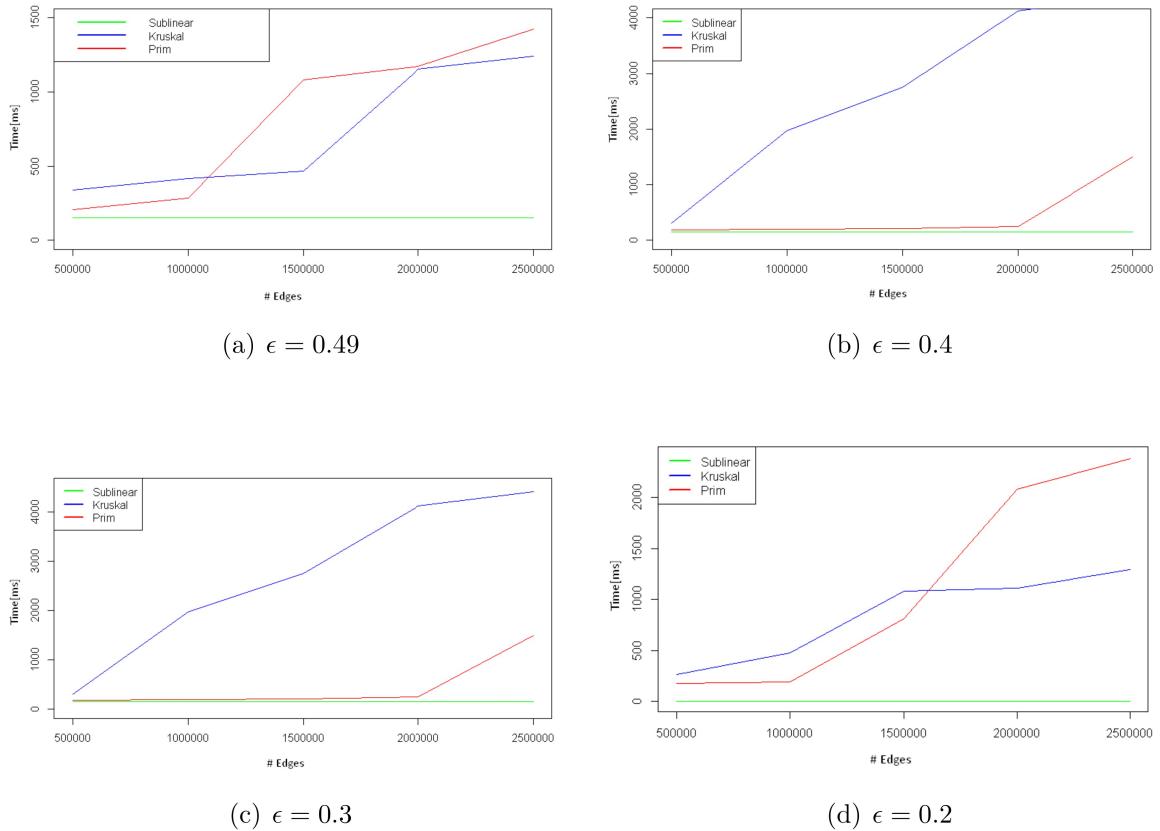
Il data-set è stato completamente creato tramite un generatore automatico che scrive i grafi su file, o può essere invocato per generare i grafi direttamente run-time. I tempi di esecuzione (per via della generazione dei grafi e per via dell'esecuzione parallela) sono risultati essere rilevanti in termine di ore di esecuzione. Il generatore nel corso dell'implementazione è diventato via via più complesso, permettendo di generare più tipologie di grafo, rendendo così i risultati dei test più approfonditi ed interessanti. Ogni esecuzione parallela è stata fornita di un grafo differente (in termini statistici) ma comparabile con gli altri grafi della stessa sessione. Per ogni sessione sono stati quindi creati 44 grafi dei quali è stato poi calcolato l'intervallo di confidenza e ne è stato preso il punto centrale. L'intero data-set consiste in un insieme di file (quindi non un data-set relazionale) dove in ogni file è racchiuso un grafo. Attualmente le dimensioni del data-set si aggirano intorno ai  $92GB$  ogni file ha una dimensione differente che varia da  $1MB$  ai  $167MB$  in base alla complessità del grafo che racchiude.

## 4.4 Risultati

In questo paragrafo mostreremo i risultati ottenuti.

### 4.4.1 Test 1

Questo test è necessario per vedere come si comporta l'algoritmo all'aumentare del carico. Affinché l'algoritmo sia sub-lineare è necessario che la curva generata dall'algoritmo approssimato tenda (all'infinito) a stare sotto alle curve generate dagli algoritmi lineari. Vediamo il comportamento dell'algoritmo in base ad i vari valori assegnati al parametro epsilon. Dalle figure si evince un evidente differenza di andamento tra i due algoritmi e l'algoritmo proposto in [1]. In particolare l'algoritmo sub-lineare sembra tendere ad  $O(1)$ ; questa supposizione è falsa in quanto il tempo di risposta dell'algoritmo da noi implementato tende a crescere proporzionalmente con la quantità di archi presenti nel grafo, nel grafico non è possibile notare questa crescita per via della presenza dei tempi degli altri due algoritmi che hanno cardinalità di gran lunga superiore.



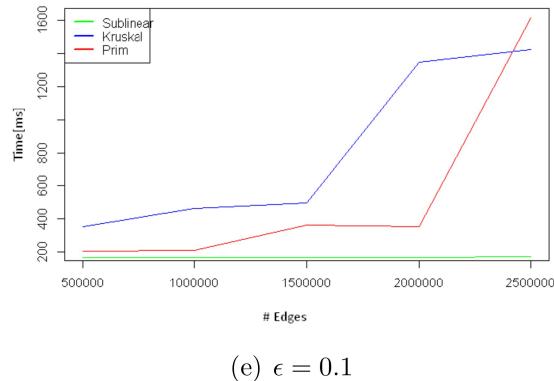


Figura 4.1: Andamento temporale dei tre algoritmi in base al numero di archi presenti nel grafo,  $N = 100000$  tempo espresso in secondi

#### 4.4.2 Test 2

In questa sezione tratteremo l'andamento funzionale del grafo, indagheremo in particolare l'assunzione:

$$(1 - \epsilon) \cdot w(T) \leq \hat{w}(T) \leq (1 + \epsilon) \cdot w(T)$$

Che per il teorema del limite centrale deve essere verificata almeno nel 75% dei casi. Come nel paragrafo Test 1 vedremo l'andamento dell'algoritmo al variare di  $\epsilon$ . In questo caso è possibile notare come al calare di  $\epsilon$  l'algoritmo tenda a sembrare più incerto, questo è dovuto ad una riduzione dei bound. La percentuale di riuscita dell'algoritmo ha sempre rasantato il 100% motivo per il quale non ci è sembrato rilevante rappresentarla su grafo. Un'altro aspetto da notare è che la dimensione del minimo albero ricoprente è inversamente proporzionale rispetto al numero di archi; questo comportamento è dato dal fatto che con una maggior quantità di archi (a parità di nodi) la probabilità che tra due nodi sia presente un cammino minimo più leggero è più alta.

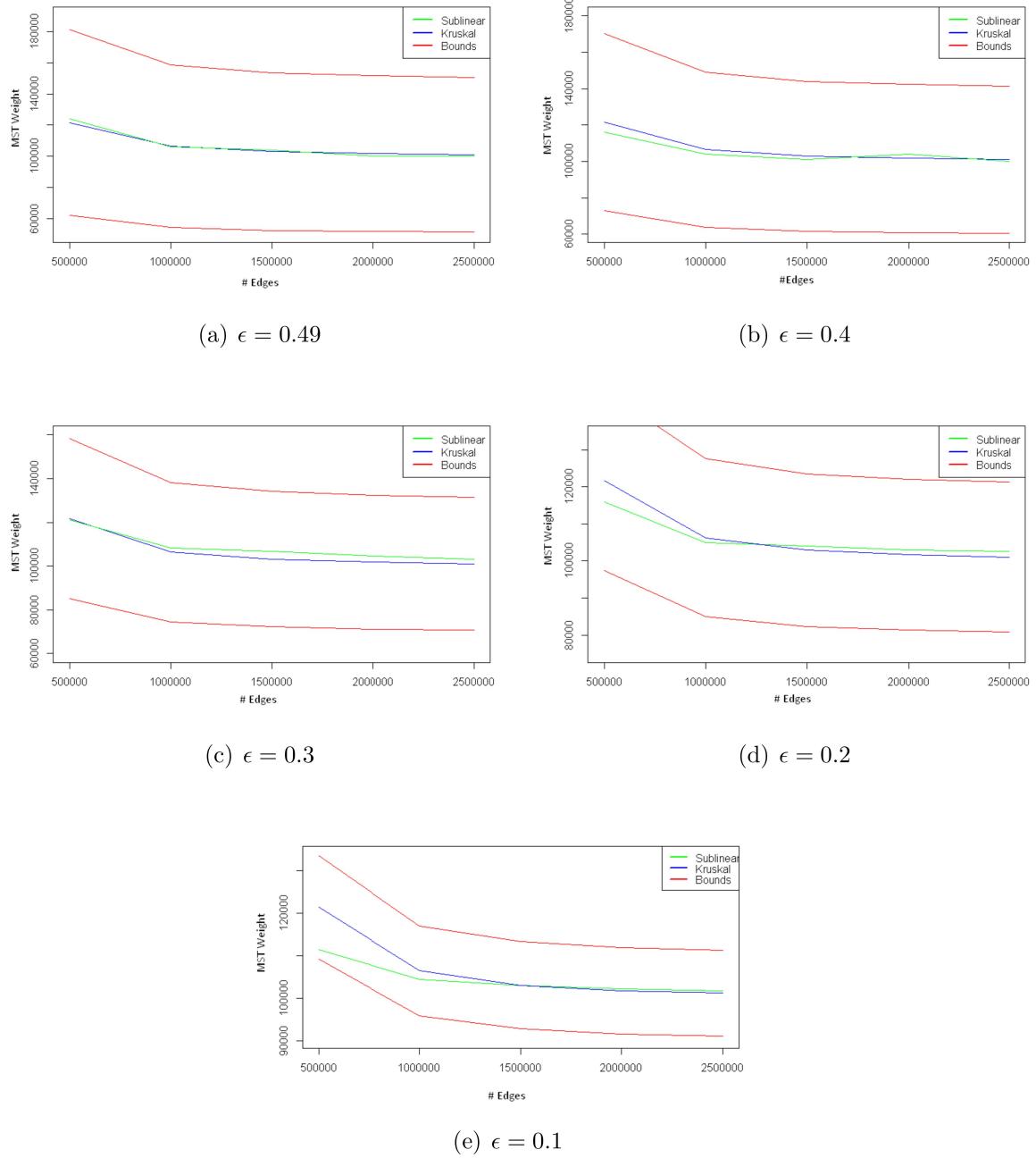
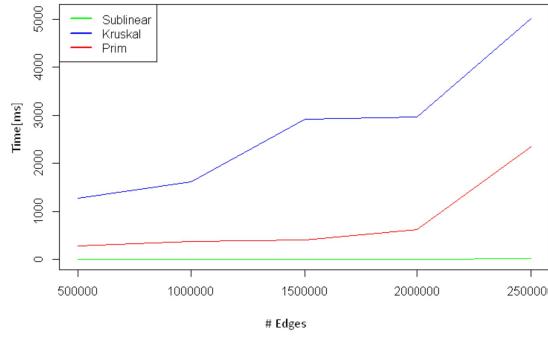


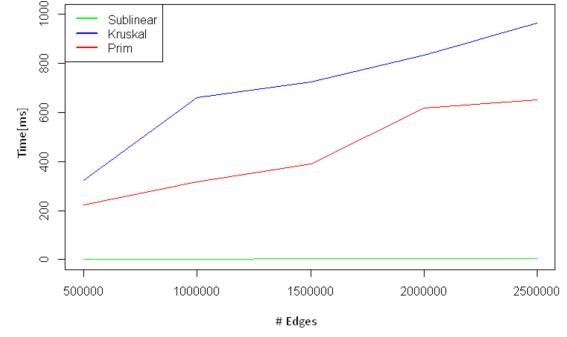
Figura 4.2: Andamento dei tre algoritmi in base al numero di archi presenti nel grafo,  $N = 100000$

### 4.4.3 Test 3

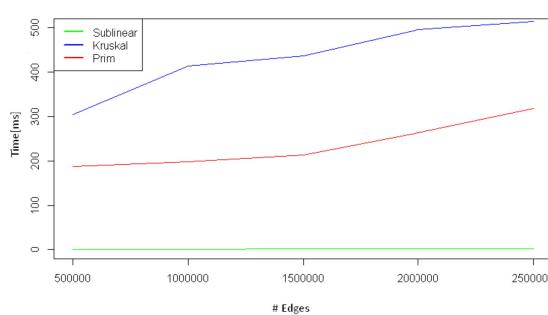
In questo test vogliamo vedere come si comporta l'algoritmo al variare della conformazione del grafo. Per questo abbiamo deciso di sottoporre all'algoritmo quattro tipologie di grafi differenti: Uniforme, Gaussiano, Scale-Free, Small-World. Per prima cosa è possibile notare che in tutte le tipologie di grafi l'andamento del tempo di risposta non differisce in maniera consistente. In tutte le tipologie di grafo kruskal sembra andare peggio, ma la cosa più importante dal nostro punto di vista è che l'algoritmo ha un andamento decisamente sublineare.



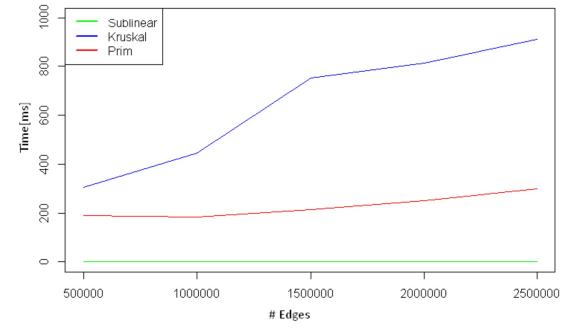
(a) Grafo Uniforme



(b) Grafo Gaussiano



(c) Grafo Scale-Free



(d) Grafo Small-World

Figura 4.3: Andamento del tempo dei tre algoritmi in base al numero di archi presenti nel grafo al variare della tipologia di grafo,  $N = 100000$ ,  $\epsilon = 0.3$

Vediamo ora come si comporta la stima del minimo albero ricoprente. Anche in questo caso il comportamento dell'algoritmo non dipende dalla tipologia di grafo. Di questi grafici l'unico che sembra rilevante ai nostri scopi è il grafo gaussiano; infatti, per una bassa quantità di archi, l'andamento sembra più smorzato rispetto all'andamento dell'algoritmo lineare. Nonostante l'algoritmo sia all'interno del bound.

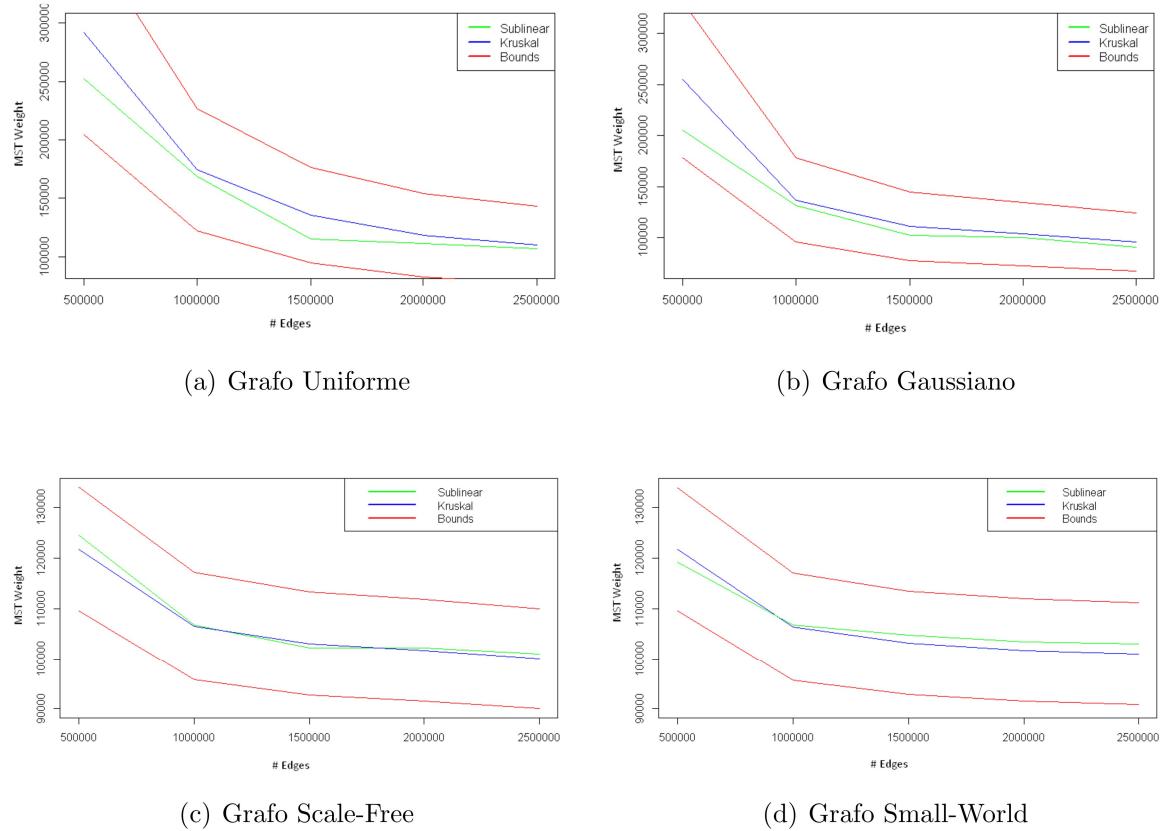


Figura 4.4: Andamento del tempo dei tre algoritmi in base al numero di archi presenti nel grafo,  $N = 100000$ ,  $\epsilon = 0.3$

#### 4.4.4 Test 4

Potrebbe risultare interessante anche analizzare l'andamento dell'algoritmo al variare del peso massimo. In tutti questi grafi l'andamento dell'algoritmo sub-lineare rispetta i bounds. Un aspetto che può risultare di interesse è che, come ci si aspetta, aumentando il peso massimo di ogni arco (e quindi anche il peso medio) risulta che il minimo albero ricoprente sia più pesante per un grafo con meno archi; in particolare nel grafo con  $W = 80$  si nota maggiormente l'andamento esponenziale negativo rispetto a  $W = 20$  che risulta essere più smorzato.

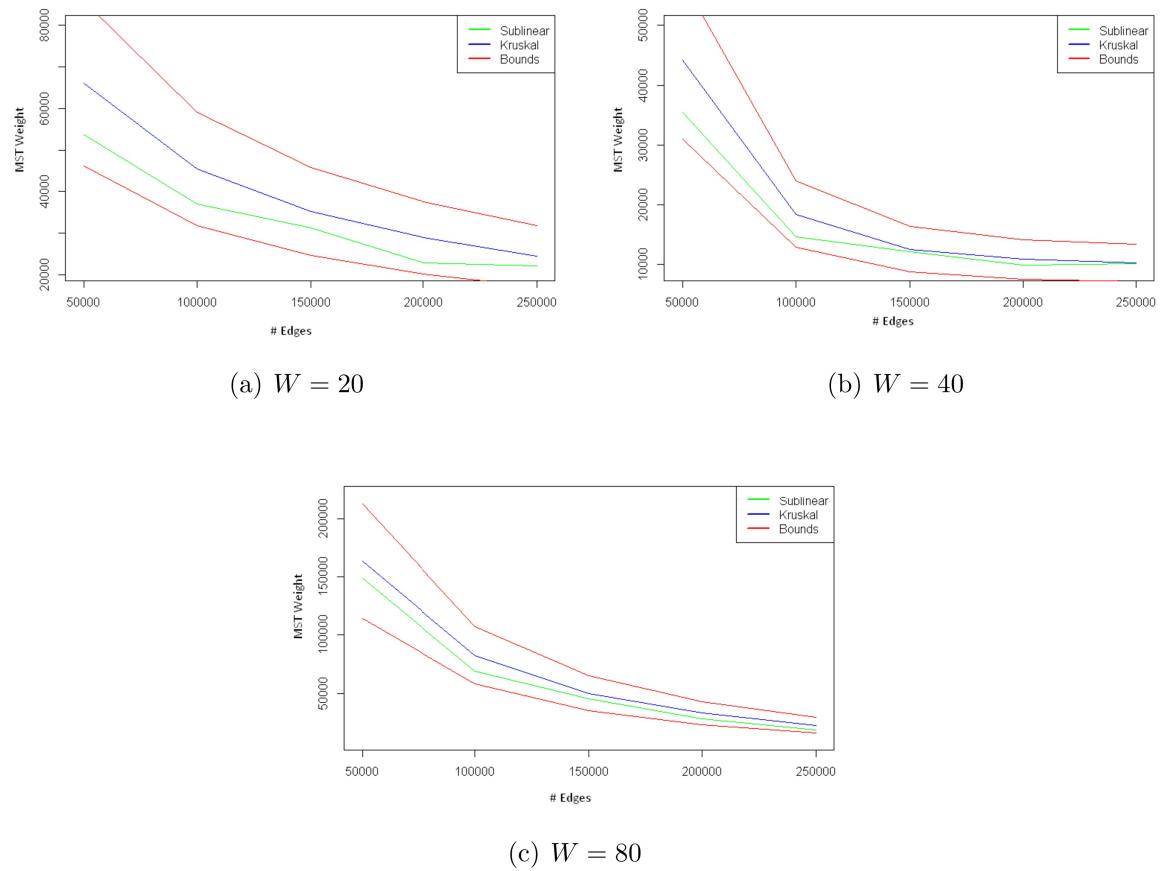


Figura 4.5: Andamento dei tre algoritmi in base al numero di archi presenti nel grafo al variare del peso massimo,  $N = 100000$ ,  $\epsilon = 0.3$

Come ultima parte del Test 3 analizziamo l'andamento del tempo di risposta dell'algoritmo al variare del peso degli archi. Con  $W = 80$  si nota che il picco di carico dei due algoritmi lineari viene raggiunto prima.

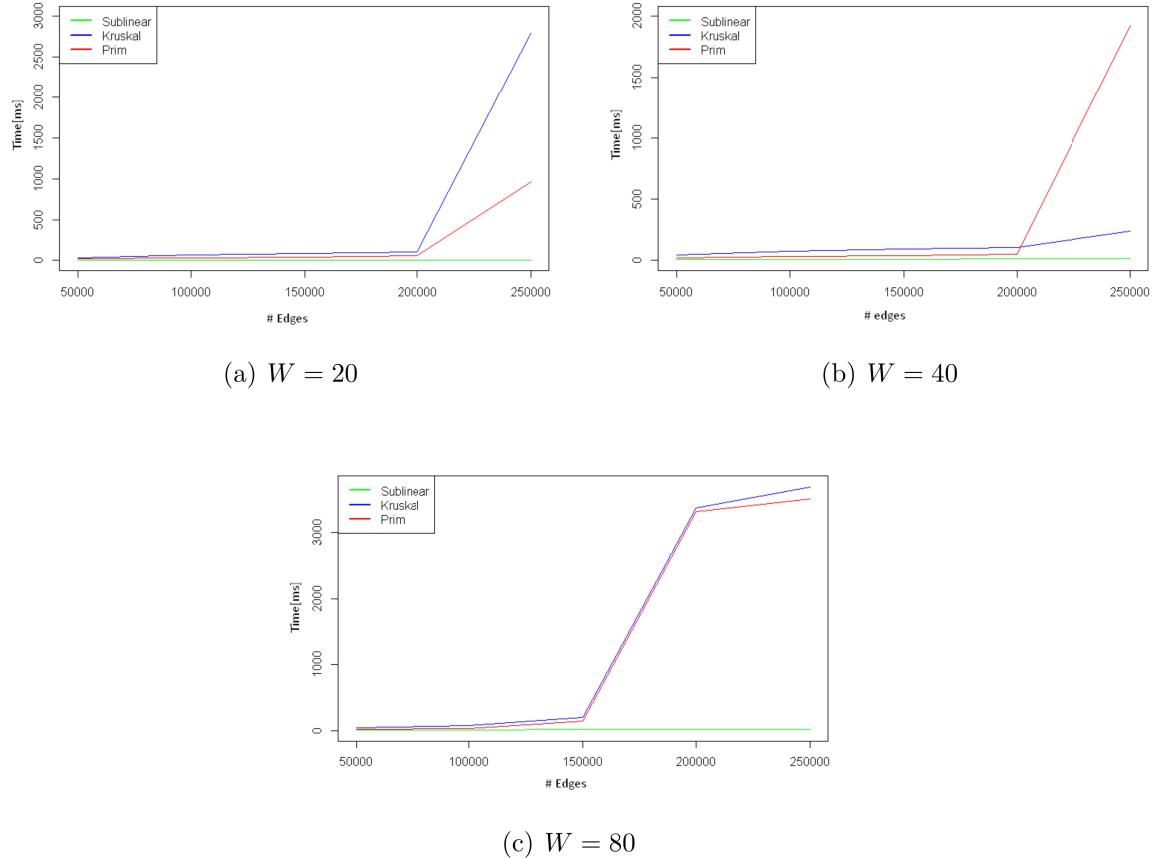


Figura 4.6: Andamento dei tre algoritmi in base al numero di archi presenti nel grafo al variare del peso massimo,  $N = 100000$ ,  $\epsilon = 0.3$

## 4.5 Conclusioni

Abbiamo creato un'implementazione dell'algoritmo sublineare in java. Con questa implementazione abbiamo testato in maniera scientifica le teorie presentate in [1] sotto vari ambienti e tipologie di grafo. Come dimostrato matematicamente in [1] l'algoritmo presentato consente di calcolare il peso del minimo albero ricoprente in tempo sub-lineare. Tramite  $\epsilon$  è possibile impostare il livello di accuratezza dell'algoritmo,

pagando però con un maggior tempo di esecuzione. Dai grafici non è possibile notarlo poiché i tempi degli altri due algoritmi risultano oscurare le variazioni dell'algoritmo sub-lineare. Presentiamo di seguito una tabella che mostra meglio le variazioni del tempo di risposta in base al numero di archi ed epsilon mantenendo gli altri fattori costanti:

<i>Archi</i>	$\epsilon = 0.49$	$\epsilon = 0.4$	$\epsilon = 0.3$	$\epsilon = 0.2$	$\epsilon = 0.1$
500000	0.3	0.75	1.0	3.0	14.75
1000000	0.5	0.75	1.0	3.25	16.0
1499999	0.75	0.75	1.5	3.75	16.25
2000000	0.75	1.25	1.5	4.25	16.5
2500000	1.0	1.5	2.33	5.0	22.66

Si nota al diminuire di  $\epsilon$  un aumento esponenziale dei tempi, in particolar modo per  $E = 500000$  archi il rapporto tra il tempo con  $\epsilon = 0.49$  e quello con  $\epsilon = 0.1$  si aggira intorno a 49.16 mentre per  $E = 250000$  archi dimezza è pari a 22.66. A dispetto di questa crescita esponenziale indirettamente proporzionale rispetto ad  $\epsilon$  l'algoritmo randomico consente un approssimazione molto più rapida del peso del minimo albero ricoprente rispetto ai già noti algoritmi di Prim e di Kruskal. Più precisamente il risultato ritornato dall'algoritmo è formato da 2 parti, una statica ed una dinamica, la parte dinamica è quella che ben approssima il peso del minimo albero ricoprente, ed è anche la parte più pesante in termini di computazione. L'algoritmo esegue una BFS in vari punti, e se questa termina è in grado di dare una stima di quella componente; tenta quindi di sondare il grafo in più parti, non trovando però informazioni per la maggior parte del tempo di esecuzione poiché il più delle volte la BFS non termina in tempo.

## 4.6 Sviluppi futuri

Uno degli sviluppi futuri potrebbe essere l'integrazione di thread all'interno dell'algoritmo che si presta bene ad essere parallelizzato per diminuire ulteriormente il tempo di esecuzione; questo comporta anche però la parallelizzazione dell'algoritmo di Prim e Kruskal così da avere uno stesso ambiente per confrontare i tre algoritmi.

# Bibliografia

- [1] Bernard Chazelle, Ronitt Rubinfeld, Luca Trevisan, "*Approximating The Minimum Spanning Tree Weight In Sublinear Time*", 2005.
- [2] Artur Czumaj, Christian Sohler "*Estimating the Weight of Metric Minimum Spanning Trees in Sublinear-Time*", 2003
- [3] Cay Horstmann, Gary Cornell "*Core Java Volume I*" PEARSON Prentice-Hall.
- [4] Cay Horstmann, Gary Cornell "*Core Java Volume II*" PEARSON Prentice-Hall.
- [5] Open Source Libraries for High Performance Scientific and Technical Computing in Java  
<https://dst.lbl.gov/ACSSoftware/colt/api/overview-summary.html>
- [6] O. Goldreich, S. Goldwasser, and D. Ron, "*Property testing and its connection to learning and approximation*", 1998
- [7] A. Frieze and R. Kannan, "*Quick approximation to matrices and applications*", 1999
- [8] N. Alon, S. Dar, M. Parnas, and D. Ron, "*Testing of clustering*", 2003
- [9] N. Mishra, D. Oblinger, and L. Pitt, "*Sublinear time approximate clustering, in Proceedings of the Twelfth Annual ACM-SIAM Symposium on Discrete Algorithms*", 2001