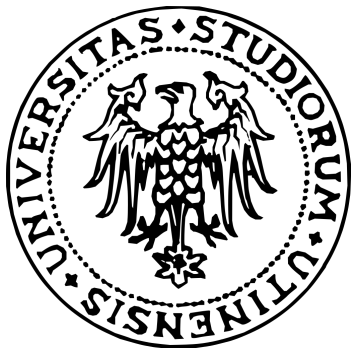


UNIVERSITÀ DI UDINE

Corso di Algoritmi e Strutture Dati

Risoluzione al problema dell'aggiunta del minor numero di archi e identificazione dell'albero dei cammini minimi di un grafo orientato

Docenti:

Carla Piazza

Alberto Policriti

Studente:

Marco Rosa

124835

rosa.marco@spes.uniud.it

Anno accademico 2015/2016

1 - Enunciato del problema

Dato in input un grafo orientato G = (V, E):

1) identificare una radice r. Il grafo G ammette radice se e solo se, a partire da essa, è possibile raggiungere tutti gli altri nodi appartenenti a G. Per farlo è necessario

2) determinare il numero minimo di archi da aggiungere a G ottenendo un grafo G' = (V, E') che ammetta una radice r. I nuovi archi |E-E'| devono essere colorati di rosso. Ottenuto quindi il nuovo grafo bisogna

3) tratteggiare gli opportuni archi di G' per evidenziare l'albero T = (V, E_t) di radice r che permette di raggiungere tutti gli altri nodi percorrendo il cammino minimo. Tutti gli altri archi restano invariati.

Il tutto deve essere eseguito partendo da un input in formato .dot e restituendo il grafo in output (rappresentazione sia di G' che dell'albero T) anch'esso in formato .dot.

2 - Come eseguire il programma

Estratto il contenuto del file .zip è possibile compilare i due programmi. Per quello risolutivo è necessario eseguire l'istruzione:

```
javac Project.java
```

Allo stesso modo per il programma per il calcolo dei tempi, eseguire l'istruzione:

```
javac TimingAnalysis.java
```

Così facendo tutte le classi utilizzate dai due programmi saranno automaticamente compilate.
Per eseguire il programma risolutivo e il calcolo dei tempi bisogna eseguire:

```
java Project <input.dot> output.dot
java TimingAnalysis
```

3 - Soluzione proposta

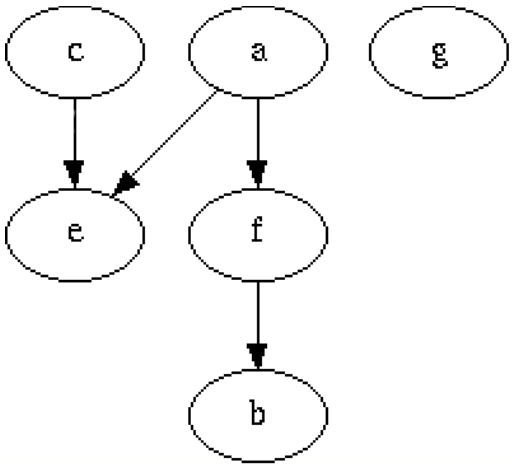
3.1 - Lettura dell'input

Il programma riceve dallo standard input il grafo da risolvere. Una volta lette tutte le righe di testo, queste vengono passate alla classe **`DotManipulation`**. Questa classe fornisce tutte le procedure necessarie per la corretta interpretazione del file e successiva creazione del grafo come struttura dati.

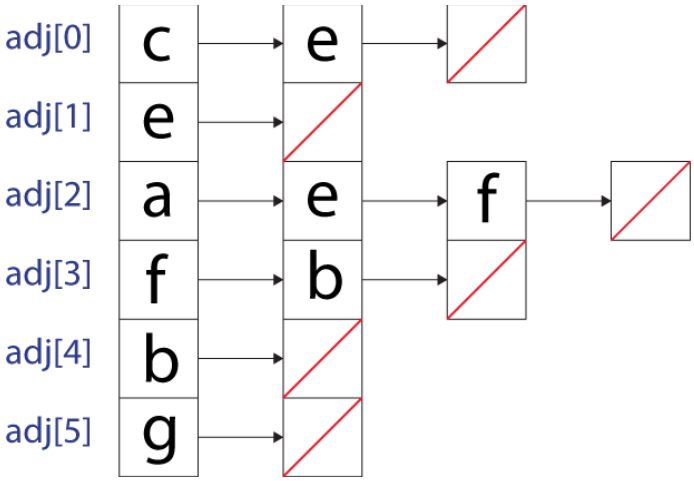
3.2 - Memorizzazione e gestione dei dati

Una volta letto il file, viene prodotto un oggetto di tipo **`DirectedGraph`** che permette di rappresentare un grafo orientato. La classe è caratterizzata da un array di nodi (chiamato *`adj`*), tanti quanti quelli presenti nel file .dot. Allo stesso modo viene anche generato il grafo trasposto creando le opportune liste di adiacenza risiedenti in *`adjT`*. Per farlo è stato sufficiente invertire l'arco tra i due nodi.

Un **`Node`** è un oggetto che contiene un dato in formato stringa (il nome del nodo, che è unico) e un riferimento ad un nodo successivo. Questo è necessario per la costruzione delle liste di adiacenza. Infatti ogni nodo appartenente all'array *`adj`* di **`DirectedGraph`** possiede un riferimento a tutti quei nodi che ha come figli diretti. Quindi dato in input un grafo come il seguente:



le liste di adiacenza che verrano a crearsi sono:



Questo è solo uno delle tante possibili strutture dell'array `adj` per il grafo proposto, in quanto l'ordine degli elementi al suo interno dipende dall'ordine delle istruzioni nel file .dot.

3.3 - Passaggi per la risoluzione del problema

1) Una volta ottenuto il grafo dal file .dot, viene invocato il metodo ***DFS(boolean transposed)*** della classe **DirectedGraph**. Tutti i nodi inizialmente vengono colorati di bianco, quindi viene effettuata una visita in profondità del grafo, impostando, per ogni nodo, il tempo di inizio e fine visita, modificandone il colore di volta in volta. Se durante la visita l'algoritmo incontra un nodo grigio, vuol dire che è presente un ciclo e non serve continuare a visitare in quella direzione. Viene quindi preso l'ultimo nodo in esame e gli viene assegnato un flag booleano `scc` (della classe **Node**, acronimo di Strongly Connected Component) che prende valore true, potendo così capire in analisi future che quel nodo chiude un ciclo;

2) Viene effettuata una seconda chiamata al metodo ***DFS(boolean transposed).*** Questa volta la visita in profondità verrà effettuata sul grafo trasposto. Prima di essere eseguita vengono ordinati in ordine decrescente tutti i nodi del grafo G in base al loro tempo di fine visita. Partendo quindi dal primo di questi nodi, è possibile trovare le componenti fortemente connesse e le loro rispettive radici;

3) A questo punto è possibile identificare il nodo radice tramite il metodo ***findRadix()*** della classe **DirectedGraph**. Questa procedura crea un array delle possibili radici `roots`, cioè tutti quei nodi che fungono da radice per le varie componenti fortemente connesse. Quindi si scorre la lista di nodi e per ognuno di loro si calcola a quale componente fortemente connessa appartiene, prendendone la radice ed eliminandola da `roots` (poichè l'array di radici è formato a partire dalla

visita DFS del grafo trasposto). A questo punto è possibile che le radici candidate siano più di una, quindi effettuo una seconda scrematura, individuando quale radice mi permette di raggiungere il maggior numero di nodi;

4) Trovata la radice r bisogna aggiungere i nuovi archi e per farlo utilizzo il metodo **`addMinimumEdges(int index)`** della classe **`DirectedGraph`**. Prima di procede vengono resettati tutti i tempi di inizio e fine visita di tutti i nodi e il loro colore viene riportato a bianco. Partendo quindi da r effettuo un'altra visita in profondità. In questo modo tutti i nodi rimasti bianchi non sono raggiungibili dalla radice e devo collegarli con un nuovo arco. Dovendo aggiungerne il minor numero possibile, l'algoritmo deve trovare i nodi adatti a cui far puntare la radice, quindi viene fatto un controllo per ogni nodo: se è bianco, se non è la radice, se non ha genitori oppure se appartiene ad una componente fortemente connessa. Se un nodo v soddisfa questi criteri, creo il nuovo arco `r -> v` e rendo neri tutti i nodi figli di v, poiché ora sono raggiungibili da r;

5) Il grafo G' = (V, E') è stato creato, ma manca ancora l'individuazione dell'albero T dei cammini minimi. Per trovarlo è necessario invocare il metodo ***`BFS_visit(int root)`*** della classe **`DirectedGraph`**. Utilizzando una visita in ampiezza sul grafo G' rendo nuovamente bianchi tutti i nodi, impostando il nodo radice r grigio e a distanza 0. Il nodo r viene aggiunto ad una coda e inizia la visita: prelevo il nodo u in testa alla coda e se non è nero ne prendo la lista di adiacenza. Quindi ciclo su tale lista tutti i nodi v_i, impostando la nuova distanza, il colore (grigio) e il padre che dovrà avere nell'albero T. Infatti per ogni v_i, se questo è bianco vuol dire che non è stato visitato prima e quindi l'arco `u -> v`$_i$ sarà sicuramente il percorso minimo da u a v_i. Ovviamente i nuovi archi aggiunti al grafo saranno tutti parte dell'albero T, poiché senza la loro aggiunta non sarebbe stato possibile raggiungere alcuni nodi. Effettuati questi aggiornamenti per ogni v_i, se questi non sono la radice e se non sono foglie, vengono aggiunti in coda alla lista e il nodo genitore viene colorato di nero. Si procede così fino a quando la lista non è vuota.

3.4 - Scrittura dell'output

Ora che il grafo G' è stato creato ed è stato evidenziato l'albero T, è possible convertire tutti i dati in .dot tramite il metodo ***`toString()`*** della classe **`Node`**, per ogni nodo appartenente all'array *`adj`* di G'. Dato un nodo u, se suo figlio v possiede il flag *`newEdge`* impostato a true, significa che `u -> v` è un nuovo arco aggiunto, quindi va colorato di rosso. Se il genitore di v nell'albero T è u, allora quell'arco va tratteggiato.

4 - Complessità computazionale

Per il calcolo considero le seguenti notazioni:

- **V** rappresenta i vertici e **|V|** il loro numero totale
- **E** rappresenta gli archi e **|E|** il loro numero totale
- **adj[v]** rappresenta la lista di adiacenza di un dato nodo v e **|adj[v]|** la sua lunghezza
- **M** rappresenta il numero di istruzioni di un file .dot

Node - findNode(Node v)

Dato un nodo v, il metodo cerca nella sua lista di adiacenza il nodo corrispondente a v. Per questo motivo, nel caso peggiore, o v si trova all'ultimo posto della lista oppure non è presente, quindi:

$$\theta(adj[v]) = \theta(|E|)$$

mentre nel caso generale:

$$O(adj[v]) = O(|E|)$$

Analogamente anche i metodi ***`toString()`*** e ***`addToTree(String p, Node v)`*** possiedono la stessa complessità.
Tutti gli altri metodi non possiedono cicli, ricorsioni o richiami ai metodi sopraelencati o esterni, quindi sono caratterizzati da una sequenza di operazioni di costo 1.

DirectedGraph - listIndexNode(Node nodeA)

Dovendo cercare il nodeA nella lista `adj`, il metodo dovrà al massimo effettuare |V| controlli, quindi nel caso peggiore in cui nodeA si trovi nell'ultima cella dell'array, la complessità vale:

$$\theta(|V|)$$

e nel caso generale:

$$O(|V|)$$

DirectedGraph - allWhite()

Il metodo rende bianchi tutti i nodi presenti nell'array `adj`, quindi la complessità vale:

$$\theta(|V|)$$

DirectedGraph - setAdjList(String[] nodes)

Si occupa di riempire l'array `adj` con i dati contenuti in nodes (che possiede lunghezza |V|), quindi la complessità vale:

$$\theta(|V|)$$

DirectedGraph - DFS_visit(Node u, Node[] list)

La visita in profondità richiama ricorsivamente se stessa in base alla lista di adiacenza del nodo u. Per questo motivo si ha la seguente complessità:

$$O(adj[u]) = O(|E|)$$

DirectedGraph - DFS()

Poiché il metodo richiama ***`DFS_visit()`*** |V| volte, la complessità vale:

$$O(|V| + |E|)$$

DirectedGraph - BFS_visit(int root)

L'algoritmo della visita in ampiezza risulta peggiorato nella sua complessità. Infatti il metodo non solo calcola le distanze dei nodi dalla radice, ma richiama anche il metodo ***`addToTree(String p, Node v)`*** della classe **`Node`**, permettendo così di calcolare anche l'albero dei cammini minimi.
La visita in ampiezza implementa una coda e poiché ogni nodo può entrarvi una sola volta, il ciclo while verrà eseguito al massimo |V| volte. Al suo interno un secondo ciclo while scorre l'intera lista di adiacenza del nodo in questione, invocando il metodo ***`addToTree(String p, Node v)`*** che anch'esso cicla |E| volte (nel caso peggiore), quindi:

$$O(|V|) + O(|E|^2) = O(|V| + |E|^2)$$

DirectedGraph - allBlack(int index)

Dato l'indice, il metodo prende il nodo corrispondente e la sua lista di adiacenza, e colora tutti i nodi di nero. Quindi:

$$\theta(|\,adj[u]\,|) \leq O(|\,E\,|)$$

DirectedGraph - findRadix()

La procedura individua il nodo radice scandendo tutti i nodi. Per ognuno di essi vengono analizzate le liste di adiacenza, quindi:

$$O(|\,V\,|) + O(|\,E\,|) = O(|\,V\,| + |\,E\,|)$$

tale risultato si ottiene anche per il risultato migliore, cioè quando il nodo candidato non ha genitori.

DirectedGraph - addMinimumEdges(int index)

Il metodo inizialmente esegue |V| volte un ciclo per resettare i tempi di inizio e fine visita di ogni nodo, quindi li rende tutti bianchi ed esegue una visita in profondità utilizzando il metodo ***`DFS_visit(Node u, Node[] list)`***. Infine un ciclo esegue |V| volte un controllo su ogni singolo nodo, rendendo neri tutti i suoi figli con il metodo ***`allBlack(int index)`***, quindi la complessità vale:

$$O(|\,V\,|) + O(|\,V\,|) + O(|\,E\,|) + O(|\,V\,| + |\,E\,|) = O(|\,V\,| + |\,E\,|)$$

DotManipulation - getAllNodes(String[] lines)

Ogni file .dot può essere composto da due tipologie diverse di istruzioni: la dichiarazione di un singolo nodo oppure l'arco tra due nodi. Potenzialmente ogni singola riga (escluse la prima che contiene il nome e l'ultima che chiude il grafo), può contenere due nodi mai incontrati nelle righe precedenti. Chiamando M il numero di righe, il numero massimo di nodi che è possibile ottenere è pari a 2M. Tutti i calcoli effettuati di seguito tengono conto di questa possibilità, considerata la peggiore. Essendo 2M il numero massimo di nodi, il metodo crea un array di stringhe lungo 2M, dove ogni cella è caratterizzata da una stringa vuota. Quindi vengono prese le singole istruzioni e ne vengono estrapolati i nodi che vengono salvati nell'array. Procedendo in questo modo è possibile che vi siano dei nodi ripetuti, quindi viene effettuato un controllo per eliminare i doppioni. Alla fine vengono contati i nodi rimanenti e viene creato un'array della lunghezza corretta. Il tutto ha un costo di:

$$\theta(2M)+\theta(M)+\theta(2M^2)+\theta(2M)+\theta(2M)=O(2M^2)$$

DotManipulation - getGraphFromFile(String filename)

Il metodo crea la lista di adiacenza del grafo G, quindi utilizza la procedura ***`getAllNodes(String[] lines)`***, quindi:

$$O(2M^2)$$

DotManipulation - getDotFromGraph(DirectedGraph G, int index)

Per prima cosa viene generato il nome del grafo, quindi vengono scritti tutti i |V| nodi presenti nell'array `adj` del grafo G. Quindi per ognuno di essi viene richiamato il metodo ***`toString()`*** della classe **`Node`**, che stampa le istruzioni in base alla lista di adiacenza di ogni nodo. La complessità vale:

$$\theta(|V|)+\sum_{i=0}^{|V|}\theta(|adj[v]|)=O(|V|+|E|)$$

DotManipulation - createDotFile(DirectedGraph G, int index)

Tutto il contenuto del file viene racchiuso in un'unica stringa ottenuta tramite il metodo ***`getDotFromGraph(DirectedGraph G, int index)`***, quindi la complessità vale:

$$O(|V|+|E|)$$

Project - main(String[] args)

La classe principale richiama tutte le procedure necessarie per la risoluzione del problema. Le istruzioni richiamate sono:

1) **Scanner**
2) **getGraphFromFile(String filename)**
3) **DFS(false)**
4) **DFS(true)**
5) **findRadix()**
6) **addMinimumEdges(int index)**
7) **BFS_visit(int root)**
8) **createDotFromFile(DirectedGraph G, int index)**

Si ha quindi la seguente complessità:

$$\theta(M)+O(2M^2)+O(|V|+|E|)+O(|V|+|E|)+O(|V|+|E|^2)+O(|V|+|E|)$$
$$=O(2M^2)+O(|V|+|E|)+O(|V|+|E|^2)$$
$$=O(2M^2+|V|+|E|^2)$$

poiché M è il numero di istruzioni di un file .dot, allora vale la disuguaglianza:

$$M \leq |V|+|E|$$

infatti se il grafo fosse composto unicamente da 4 nodi senza archi, vale l'uguaglianza. Se fossero presenti solo 2 nodi e 2 archi, allora M vale 2 che è minore di V+E che vale 4. Quindi il calcolo sopracitato diventa:

$$\leq O(2(|V|+|E|)^2+|V|+|E|^2)$$
$$=O(2|V|^2+4|V||E|+2|E|^2+|V|+|E|^2)$$
$$=O(|V|^2+|V||E|+|E|^2)$$

Poiché vale la seguente condizione:

$$|E| \leq |V|^2$$

ottengo:

$$\leq O(|V|^2 + |V|^3 + |V|^4) \rightarrow \boxed{O(|V|^4)}$$

5 - Analisi dei tempi

Definisco con N la dimensione di un grafo G = (V,E), in particolare N = |V|.
All'avvio del programma principale **`TimingAnalysis`** vengono generati M input casuali di diverse grandezze N prefissate: 50, 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000, 1500, 2000, 2500, 3000 (modificabili all'interno della classe).
Per la creazione dei grafi ho implementato la classe **`RandomGraphs`** che si appoggia allo pseudo random number generator visto a lezione.
La classe principale **`TimingAnalysis`** esegue M volte il calcolo dei tempi, calcolando:

- le ripetizioni
- il tempo medio
- il tempo medio netto
- l'intervallo di confidenza

Per la creazione di input ben distribuiti, l'algoritmo procede nel seguente modo:

1) il programma viene invocato e genera uno **`PseudoRandomGenerator`**;
2) viene creato un array `nodesValue` lungo N che contiene i dati di ogni singolo nodo;
3) viene creato un array `val` di valori casuali lungo 3N. I valori vengono generati dallo PRNG;
4) presi due indici casuali A e B (tramite PRNG), se A-B>0: se A-B è divisibile per 2, allora viene creato un arco tra `nodesValue[A]` e `nodesValue[B]`, altrimenti viceversa. Se invece A-B <= 0 viene aggiunto il solo nodo in `nodeValues[A]`.

La classe principale effettua tutti i calcoli necessari e alla fine della computazione restituisce un file in formato .log chiamato *TimesResults*. Questo contiene tutti gli esiti per tutte le M diverse grandezze N dei grafi.
Per modificare la tipologia di input fissi è sufficiente aprire il file *TimingAnalysis.java* e alla riga 32 modificare l'array di interi `nodesNumber`.
Il programma inoltre crea una directory *Analysis_folder* che conterrà a sua volta tre diverse directory e il file con i risultati. Le tre sotto-directory sono: *fixedInput* (grafi casuali fissi), *fixedOutput* (i grafi casuali fissi risolti) e *randomGraphResolution* (utile al programma per salvare tutti gli input e gli output).

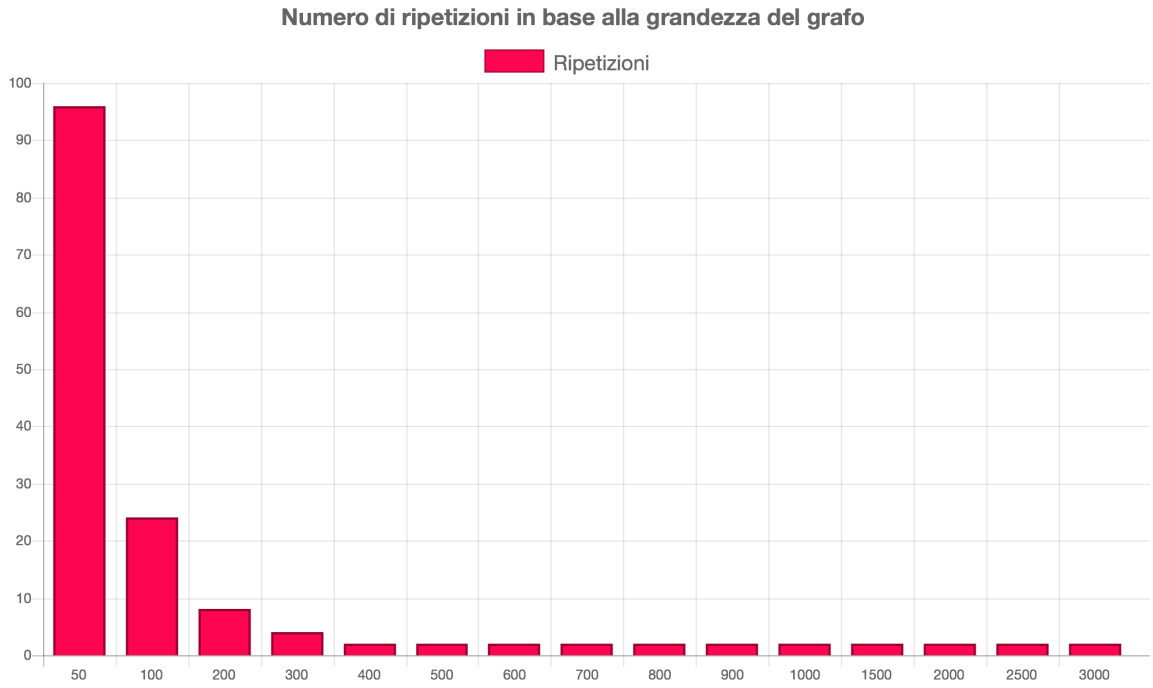
5.1 - Dati ottenuti

Il programma è stato lanciato su un MacBook Pro con 16GB di ram e un processore Intel Core i7 da 2,2 GHz. I seguenti dati sono stati ottenuti con un valore di errore massimo pari a K = 0.05:

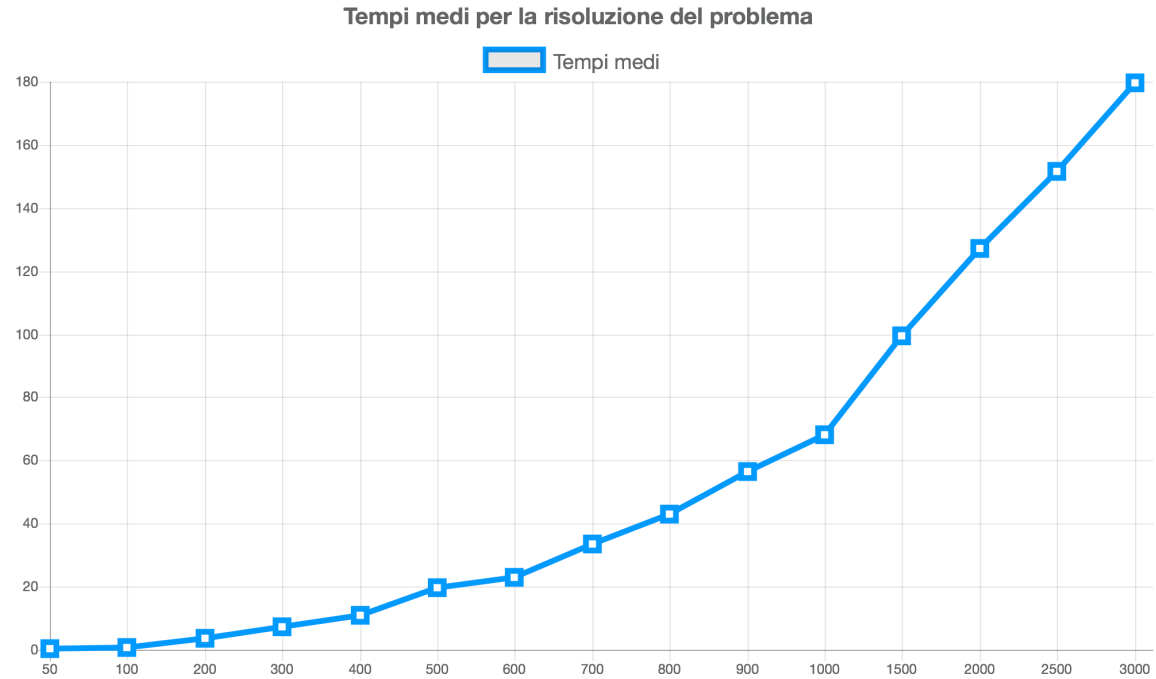
1 - Valori ottenuti con K = 0.05. I tempi sono espressi in millisecondi.

Dimensione (N)	Ripetizioni	Tempi medi	Tempi medi netti	Intervalli di confidenza
50	96	0.2395	0.2583	(0.3569619 ± 0.088378)
100	24	0.75	0.7142	(0.80386 ± 0.074086)
200	8	3.625	3.4166	(2.91559 ± 0.110680)
300	4	7.25	6.4166	(6.98392 ± 0.231113)
400	2	11.0	10.5	(12.6166 ± 0.371847)
500	2	19,5	18.6666	(16.3666 ± 0.463360)
600	2	23.0	24.25	(22.5 ms ± 0.805152)
700	2	33.5	35.875	(33.1 ms ± 1.276635)
800	2	43.0	43,25	(44.7 ms ± 1.755267)
900	2	56.5	57.25	(56.6 ms ± 1.156233)
1000	2	68.0	68.5	(68.6 ms ± 1.122517)
1500	2	99.5	97.5	(98.7 ms ± 1.952144)
2000	2	127.0	127.0	(125.0 ms ± 1.81762)
2500	2	151.5	154.5	(155.2 ms ± 1.83026)
3000	2	179.5	180.5	(179.1 ms ± 1.71765)

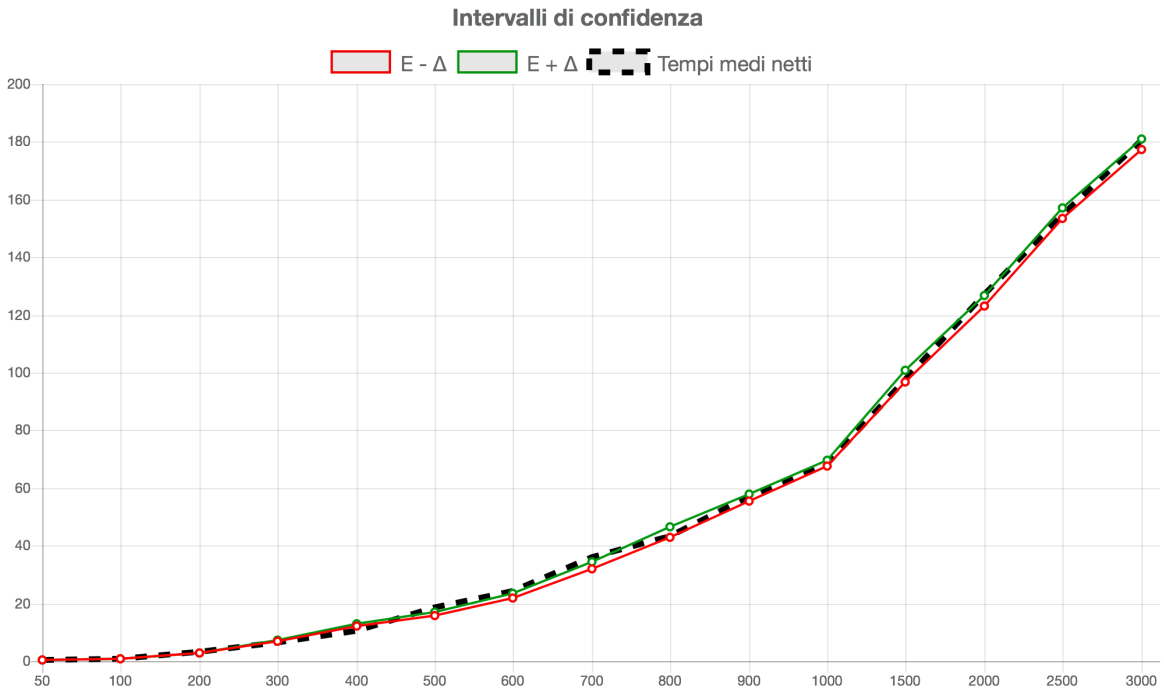
Di seguito il grafico che rappresenta il numero di ripetizioni. Sulle ascisse sono collocate le M diverse grandezze di input, mentre sulle ordinate il valore delle ripetizioni:
Si nota subito come il numero delle ripetizioni diminuisca al crescere di N fino a stabilizzarsi:



I tempi medi sono rappresentati nel seguente grafico, dove troviamo N sulle ascisse e il tempo in millisecondi sulle ordinate:



Inizialmente la curva tende a non alzarsi eccessivamente, in quanto sono necessari input più grandi per poter notare una considerevole differenza. Infatti verso destra, quando il numero di nodi di differenza cresce, si può notare una salita maggiore.
Infine il programma calcola gli intervalli di confidenza. Nel grafico seguente sono presenti tre gruppi di dati. Indicando con E il valore atteso, la linea rossa indica il valore E-Δ, quella verde E+Δ e quella nera i tempi medi netti:



e come si può notare i tempi medi netti restano, nella maggior parte dei casi, all'interno dell'intervallo di confidenza. A causa della generazione casuale di input è possibile che l'algoritmo risolutivo cada su grafi più complessi rispetto a quelli generati precedentemente, compromettendo così il calcolo. Vanno anche tenuti in considerazioni svariati fattori, uno tra questi l'I/O dei file, che vengono creati e sovrascritti più volte ed eventuali interrupt.

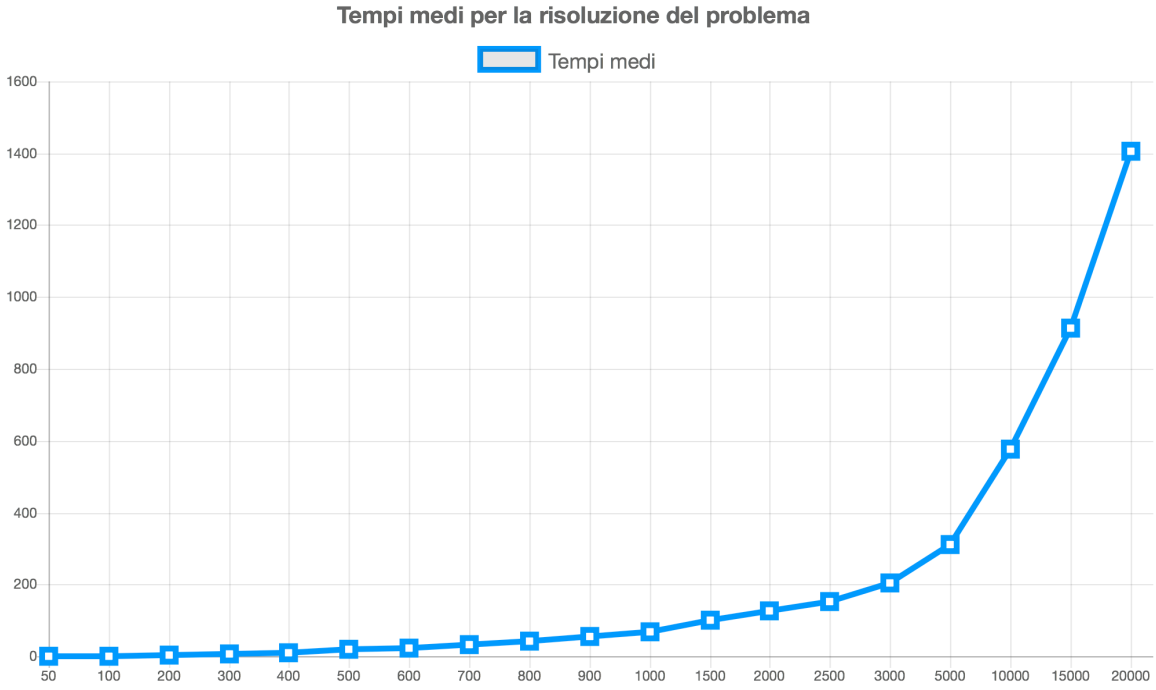
5.3 - Rapporto tra i dati ottenuti e complessità calcolata

Poiché la computazione risulta molto veloce, sarebbe necessario utilizzare input notevolmente grandi per vedere una crescita di $|V|^4$ nei dati raccolti.
Lo stesso test è stato effettuato aggiungendo 4 nuove dimensioni: 5000, 10000, 15000, 20000. I dati ottenuti sono i seguenti:

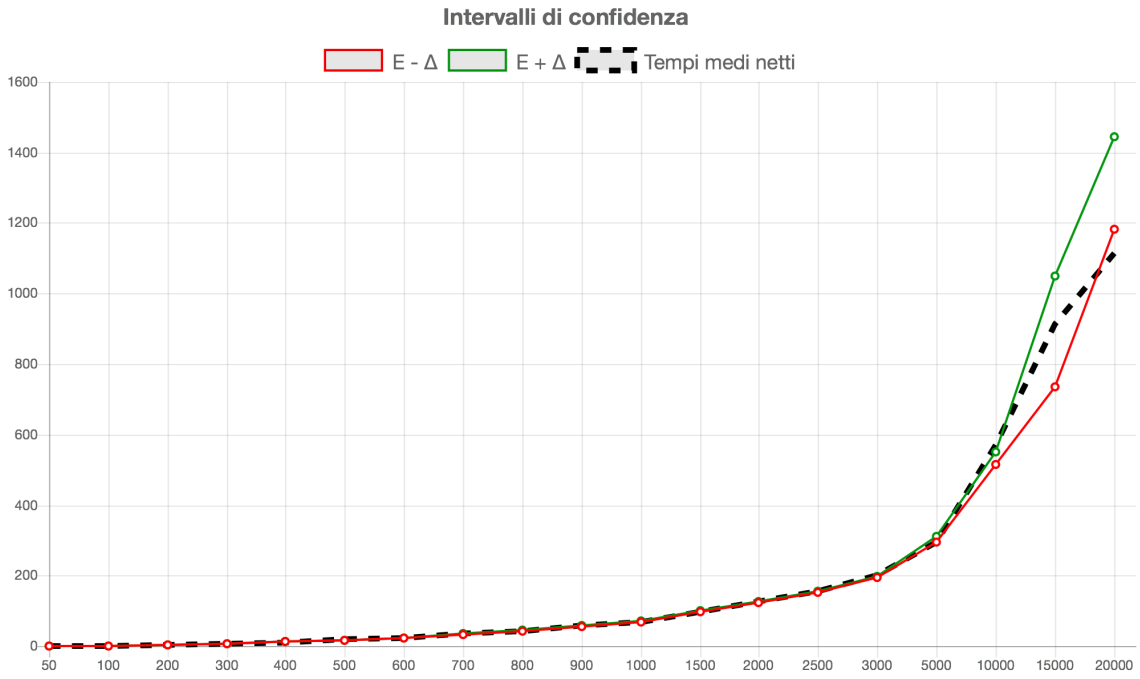
Valori ottenuti con K = 0.05 su grafi grandi. I tempi sono in millisecondi

Dimensione (N)	Ripetizioni	Tempi medi	Tempi medi netti	Intervalli di confidenza
5000	2	310.0	294.0	(302.6 ± 8.87600)
10000	2	575.0	570.0	(532.7 ± 16.8664)
15000	2	815.0	815.0	(891.3 ± 157.14423)
20000	2	1406.0	1155.5	(1314.1 ± 131.94029)

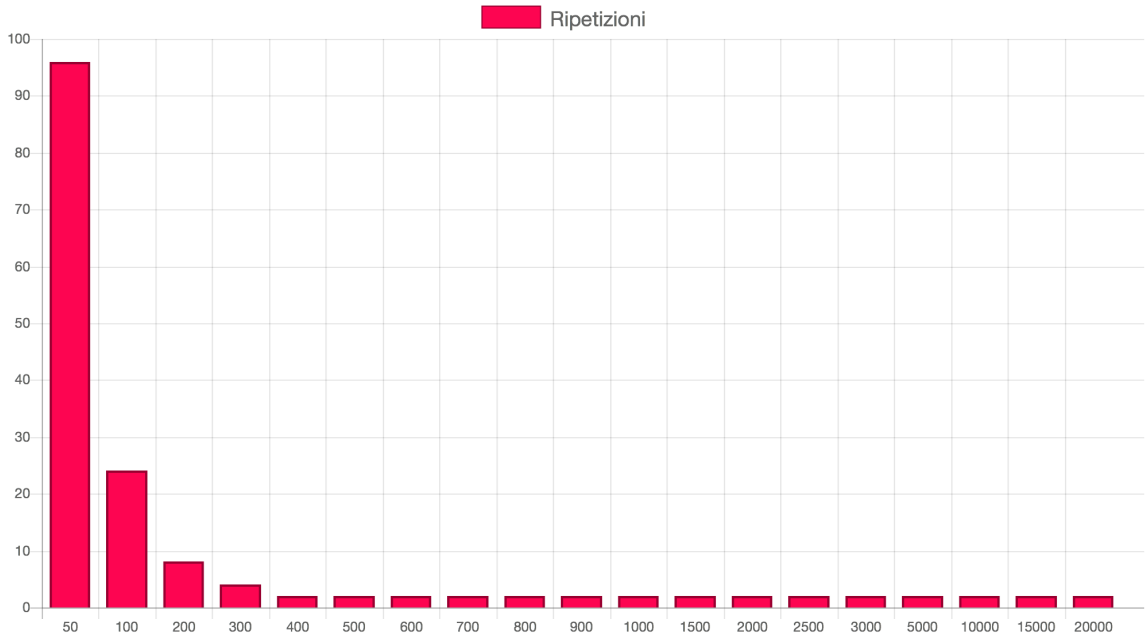
E visualizzando i dati nei grafici, si nota subito la differenza rispetto ai calcoli precedenti:



I tempi aumentano di molto poichè le differenze tra i vari input sono sempre più grandi. Infatti se prima vi era una differenza di 100 nodi, negli ultimi dati aggiunti vi è una differenza di 5000 nodi.
Come invece si può notare dal grafico seguente:



l'intervallo di confidenza (E-Δ, E+Δ) tende ad allargarsi, dando così meno precisione sulla previsione del tempo medio netto. Infine le ripetizioni non hanno subito variazioni rispetto ai calcoli precedenti, attribuendo il valore minimo di ripetizioni a tutti i grafi oltre una certa dimensione:



6 - Problemi riscontrati

L'implementazione del DFS è quello che mi ha dato i maggiori problemi. La difficoltà risiedeva nella corretta scansione di G e nello scrivere un unico metodo che non influisse sul calcolo del grafo G o del suo trasposto.

Il metodo per la ricerca della radice è stato totalmente rivisitato. Inizialmente non effettuavo il calcolo del DFS sul grafo trasposto, ma mi limitavo a scegliere come radice quel nodo che aveva il delta massimo rispetto al tempo di inizio e fine visita. Questo metodo funzionava su alcuni grafi più semplici, ma falliva in tutti gli altri casi. Utilizzando invece il DFS sul grafo trasposto ed individuando le componenti fortemente connesse credo di aver risolto il problema, infatti ho applicato la mia vecchia soluzione e quella nuova sugli stessi grafi e il risultato è andato a favore di quest'ultima.

Nel calcolo della granularità tramite l'algoritmo 4, per migliorarne la precisione ho utilizzato il metodo System.nanoTime() che fornisce il tempo trascorso in nano secondi da quando è stato avviato il programma. Nonostante la conversione in millisecondi, il tempo minimo per calcolare le ripetizioni risultava troppo piccolo e quindi indipendentemente dalla grandezza del grafo, questo veniva risolto in un tempo tale da portare il suo numero di ripetizioni a 2. Per ottenere quindi dei dati migliori, ho optato per il mantenimento del calcolo della granularità in millisecondi, quindi utilizzando System.currentTimeMillis(). I risultati migliori li ho ottenuti con granularità pari a 1ms.

Di rado accade che i valori degli intervalli di confidenza siano negativi. Questo probabilmente accade a causa delle generazione di grafi casuali che può portare inizialmente a grafi molto complessi e successivamente a grafi molto più semplici. Questi valori negativi sono dati dal calcolo del tempo medio netto. Per questo motivo, quando viene calcolato il tempo medio netto, se questo risulta negativo ripete il calcolo (può accadere che lo ripeta anche una decina di volte), mentre se nell'intervallo di confidenza $E-\Delta < 0$, allora questo viene posto pari a 0.