Relazione Progetto Itinere Ingegneria degli Algoritmi

Ricerca Nodo Medio ***Progetto N°2***

Filippo Badalamenti (Team Leader)	0244480	***19/01/18***
Emanuele Alfano	0239226	
Marta Caggiano	0239470	

Versione Python: 3.6 su Linux, script di supporto scritti in bash, grafici con MatLab.

Funzionamento dell'algoritmo:

La Traccia 2 richiedeva, dato un grafo G non orientato, aciclico e non pesato, di trovare il nodo N che fosse medio per più coppie di nodi.
Per fare ciò bisogna visitare ogni nodo contenuto nel grafo almeno una volta, per contare quante volte esso è medio, e confrontarlo con gli altri per stabilire se è il maggiore.

Facendo una similitudine, durante l'analisi dei nodi, l'algoritmo procede sul grafo come un'onda in uno stagno, generata dal nodo (sasso) preso in esame.

Inoltre possiamo sfruttare gli alberi, essendo questi grafi non orientati aciclici, per esaminare il numero di percorsi di cui il nodo N preso in considerazione è medio, e che idealmente viene posto come radice.
In realtà il nostro algoritmo non usa mai esplicitamente gli alberi come oggetti, ma ne prende in prestito degli importanti concetti, che sono poi adattati ai grafi. Ad esempio quando esaminiamo il nodo N_0 (Figura 1), che diventa radice (Figura 2), indichiamo i nodi N_1, N_2 e N_3 come i suoi figli.
Ognuno dei figli di N_0 genera un ramo: un ramo non è altro che una lista composta da tutti i nodi che hanno lo stesso progenitore. Ad esempio, N_5 e N_4 fanno parte del ramo di N_1, visto che sono suoi figli; ma anche N_7, N_8 ed N_9 rientrano nel ramo di N_1, essendo figli di N_4, sebbene si trovino su livelli diversi.
Il livello è un altro concetto importante: è sempre una lista di nodi, ma al contrario del ramo, che scende in profondità nell'albero, il livello ci informa sulla distanza effettiva tra i nodi di un certo livello X e la radice, come una visita BFS. Ad esempio, la lista del livello 3 sarà composta da N_7, N_8, N_9 ed N_{10}, quella del livello 2 da N_4, N_5 ed N_6 (Figura 2).
Si noti che la schematizzazione ad albero in questo senso ci fornisce una maggiore facilità di lettura del livello, il quale ingloba le informazioni del ramo, e che quest'ultima è una struttura dinamica: infatti il numero di rami può solo diminuire (basti vedere il ramo di N_2, che scompare al secondo livello) e gli elementi che contiene si modificano ad ogni passo.

FIGURA 1

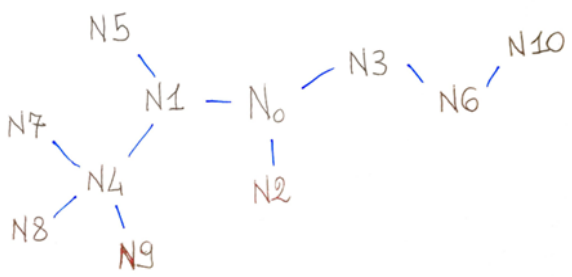
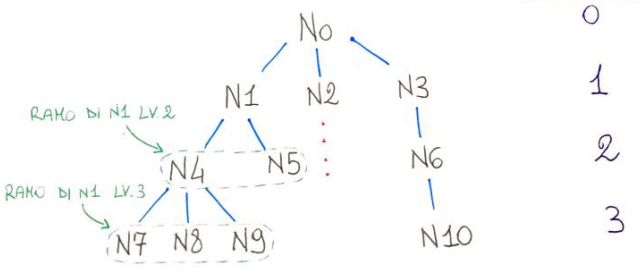


FIGURA 2



L'algoritmo che abbiamo ideato sfrutta questi concetti per calcolare il numero di volte in cui N_0 è medio: infatti basta utilizzare la lunghezza delle sotto-liste dei rami secondo la formula:

$$len\big(ramo(k)\big) \cdot (somma\ del\ numero\ di\ elementi\ dei\ rami\ successivi\ a\ k) \quad (1)$$

che va applicata ad ogni ramo di ogni livello. Ad esempio, nel caso della Figura 2, sarà:

- $(1 \cdot 2 + 1 \cdot 1) = 3$ *al livello 1*
- $(2 \cdot 1) = 2$ *al livello 2*
- $(3 \cdot 1) = 3$ *al livello 3*

sommando i risultati otteniamo che N_0 è medio 8 volte nel grafo.
È importante notare che, per continuare ad esistere percorsi validi per N_0, i figli devono crescere almeno su due rami diversi, poiché altrimenti N_0 non sarebbe più medio del percorso tra le coppie di nodi.
Dopo aver eseguito questo esame su ogni nodo del grafo, saremo sicuri di aver trovato quello medio per il maggior numero di volte.
Di seguito un output di esecuzione per il grafo in figura 1 (codice in ***demoGrafo.py***)

Il grafo in Fig1 ha la seguente matrice di adiacenza:
Adjacency Lists:
0:[1, 2, 3]
1:[4, 5, 0]
2:[0]
3:[0, 6]
4:[8, 7, 9, 1]
5:[1]
6:[3, 10]
7:[4]
8:[4]
9:[4]
10:[6]

Nel grafo in Fig1 il nodo medio di più percorsi è:
Nodo 1 medio per 9 volte.
I singoli nodi sono medi:
Nodo 0 medio per 8 volte.
Nodo 1 medio per 9 volte.
Nodo 2 medio per 0 volte.
Nodo 3 medio per 3 volte.
Nodo 4 medio per 6 volte.
Nodo 5 medio per 0 volte.
Nodo 6 medio per 1 volte.
Nodo 7 medio per 0 volte.
Nodo 8 medio per 0 volte.
Nodo 9 medio per 0 volte.
Nodo 10 medio per 0 volte.

L'implementazione del codice:

Il grafo viene generato dalla funzione *mkGraph*, che in base all'input sceglie una delle 6 modalità: "random", "stella", "lineare", "d-heap", "sfilacciato" e "asterisco". Si tratta di diverse forme significative, che sono utili per comprendere la versatilità e l'efficienza dell'algoritmo.
I fondamentali punti di forza del nostro codice sono le tre funzioni *hisSon*, *pathsCountLevel* e *medNode*.

***hisSon*:**
È una funzione che ci restituisce tutti i figli di un nodo, ricordando chi era suo padre; impedisce così di tornare ad esaminare nodi già visitati. Viene chiamata in medNode ed ha un tempo di esecuzione di $O(\delta v)$, dove δ è il grado del nodo V.

***pathsCountLevel*:**
È la funzione che applica la formula (1) delle sottoliste dei rami, per il calcolo dei cammini possibili. Viene chiamata in medNode e ha un tempo di esecuzione dipendente dal grado del nodo iniziale, ovvero dal numero dei rami: di conseguenza nel caso peggiore sarà O(n) con una costante moltiplicativa estremamente bassa, grazie all'esecuzione in tempo costante delle len(k).

medNode:

```
Algoritmo medOfGraph(grafo G, nodo N) --> N medio di percorsi
    level <-- figli di N inizializzati nella forma della struttura base (vedi descrizione)
    medOfPath <-- 0
    while (level ha ancora 2 rami) do
        medOfPath <-- medOfPath + coppie di percorsi al livello level
        newLevel <-- lista vuota
        for each (ramo in level) do
            newBranch <-- lista vuota
            for each (nodo in ramo) do
                    newBranch <-- estendi con la lista dei figli di nodo - suo padre
            if sonBranch ha figli then
                newLevel <-- estendi con sonBranch
        level= newLevel
    return medOfPath
```

È un esempio di programmazione dinamica, e funziona nel seguente modo:

- Inizialmente si crea una lista con la seguente struttura:

$$L_i = \left\{ \overbrace{\left[\left(\frac{n_1}{dad_{n1}}\right), \left(\frac{n_2}{dad_{n2}}\right), ecc \right]}^{R_k}, \overbrace{\left[\left(\frac{k_1}{dad_{k2}}\right), \left(\frac{k_2}{dad_{k2}}\right), ecc \ldots \right]}^{node_j}, ecc \ldots \right\}$$

  - In **blu** è rappresentata la variabile livello, la quale contiene rami e nodi;
  - In **verde** è evidenziato un ramo, che all'inizio dell'algoritmo conterrà solo le informazioni di un nodo e del padre;
  - In **arancio** si ha la coppia nodo-padre (nodo indice [0], padre indice [1]).

  Nel tempo questa struttura si potrà modificare ed evolvere come precedentemente descritto, ma la sua suddivisione interna rispetterà sempre questo schema.

- Si inizializza il contatore a zero e si impone che, finché almeno 2 rami di N_0 hanno figli, si calcolino i percorsi possibili tra di loro con *pathsCountLevel*.
- Il nuovo ramo sarà generato mediante *hisSon* applicato ad ogni nodo del ramo al livello precedente.

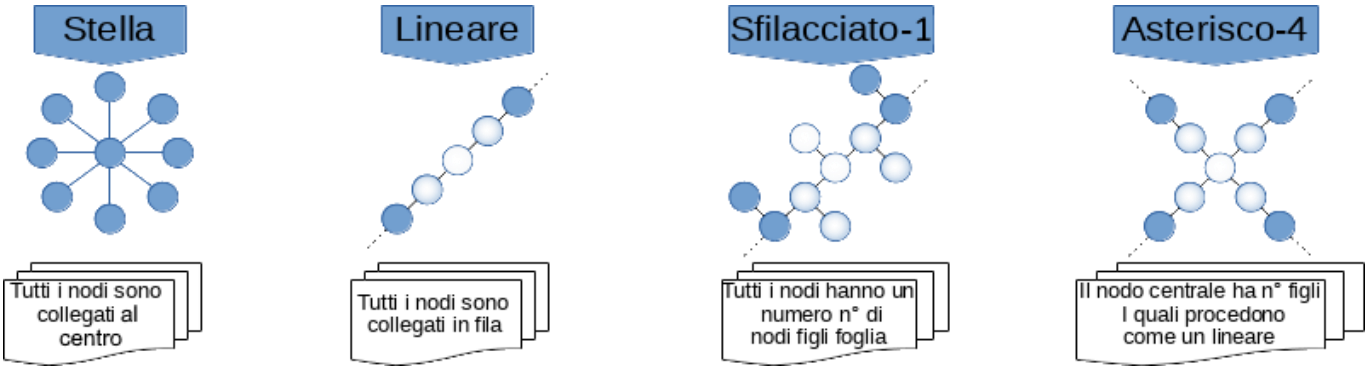
In altre parole, dopo aver applicato la formula (1) siamo "saltati" al livello successivo del grafo, e ogni ramo porta con sé solo le informazioni che ci sono strettamente necessarie: l'Id del nodo, e l'Id del nodo padre.
Si fa notare che queste informazioni hanno la forma di interi (*int*) contenuti in liste, non di oggetti, il che rende più efficiente l'utilizzo della memoria RAM, la quale al crescere degli elementi trae un notevole beneficio da questa semplificazione.

- Grazie all'if impediamo che level si riempia di liste vuote, generate dai rami del livello precedente composti da sole foglie; in questo modo si realizzerà la condizione del while.
- Una volta arrivato alla fine del grafo, *medNode* restituisce il *medOfPaths*, che rappresenta il numero di volte in cui N_0 è risultato medio.

Analisi tempi d'esecuzione:

L'analisi dei tempi di esecuzione risulta fortemente correlata alla topologia del grafo; l'algoritmo infatti esclude dal calcolo tutti i nodi "foglia", poiché sviluppano un albero con un solo ramo, mentre può potenzialmente impiegare n passi per calcolare quante volte un nodo più "centrale" nel grafo è medio.
Ciò crea una variazione significativa nei tempi, che oscillano tra $O(n)$ e $O(n^2)$: quante più foglie saranno presenti, tanti meno nodi analizzerà l'algoritmo, che tenderà ad $O(n)$.

Di seguito degli esempi di forme di grafi da noi codificati come forme significative.



Il caso migliore è dato dal grafo a stella, in cui (n-1) nodi sono collegati ad un unico nodo centrale N_0. Come costo base abbiamo il grado di N_0, che sarà $\delta v=(n-1)$: di conseguenza *hisSon* e *pathsCountLevel* costano rispettivamente δv e $2\cdot\delta v$ (una volta perché li visitiamo per stabilire la lunghezza dei rami, la seconda volta per moltiplicarli tra di loro). Inoltre, *medNode* viene chiamato solo una volta per il nodo centrale, mentre con un if sul grado degli altri nodi essi non vengono calcolati.

Il caso peggiore si verifica invece nel grafo di forma lineare; esso ha infatti, ad esclusione di soli due nodi "foglia", tutti i nodi con grado $\delta v = 2$; in questo caso, *hisSon* e *pathsCountLevel* non pesano molto nel tempo di esecuzione per ogni nodo, ma essendovene pochi terminali, verranno analizzati la quasi totalità degli stessi, in particolare secondo la formula:

$$\sum_{i=0}^{\frac{n}{2}-1}(n-2*i)=n+(n-2)+(n-4)+\cdots=O(n^2)$$

Dove i è l'indice che indica la distanza dal centro del grafo, fino ad arrivare ad un elemento che precede la foglia, ed n indica il numero di nodi analizzati.

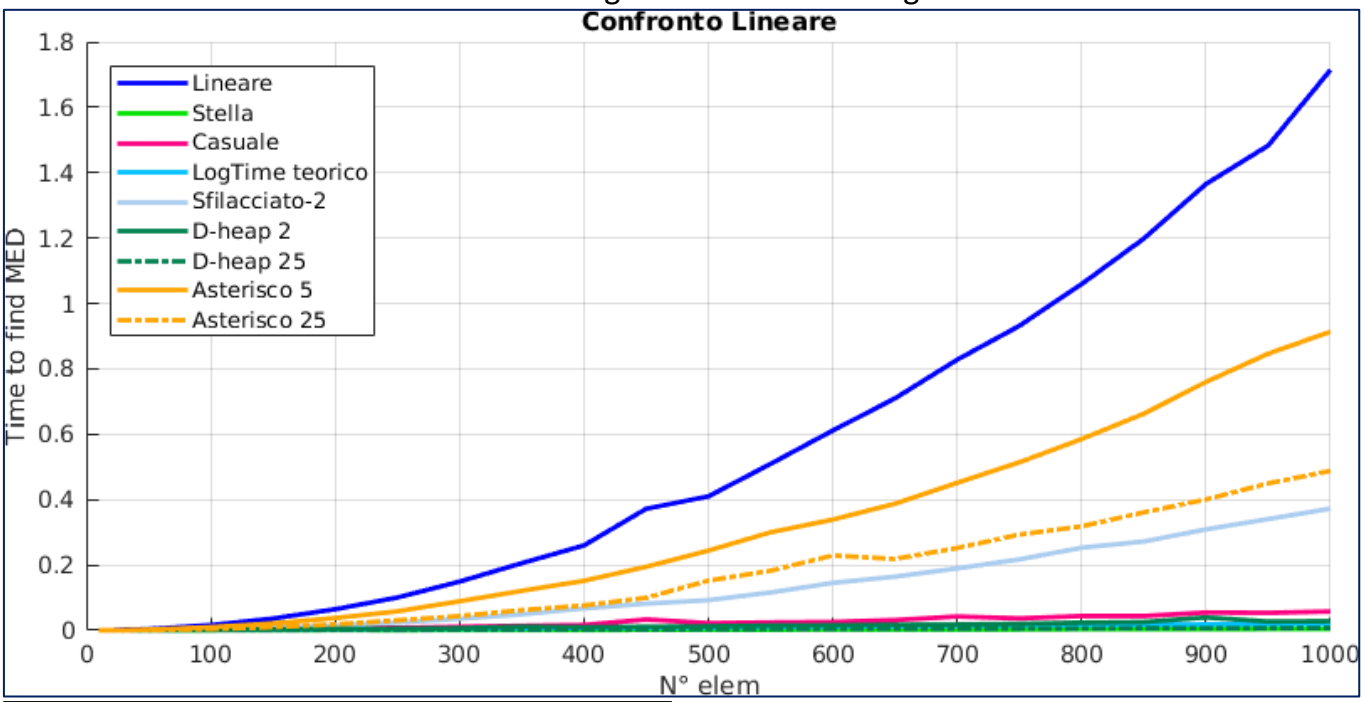
A causa della forte divergenza tra caso migliore e caso migliore, si rende pertanto palese come l'analisi teorica del caso randomico sia impossibile; possiamo tuttavia fare alcune osservazioni:

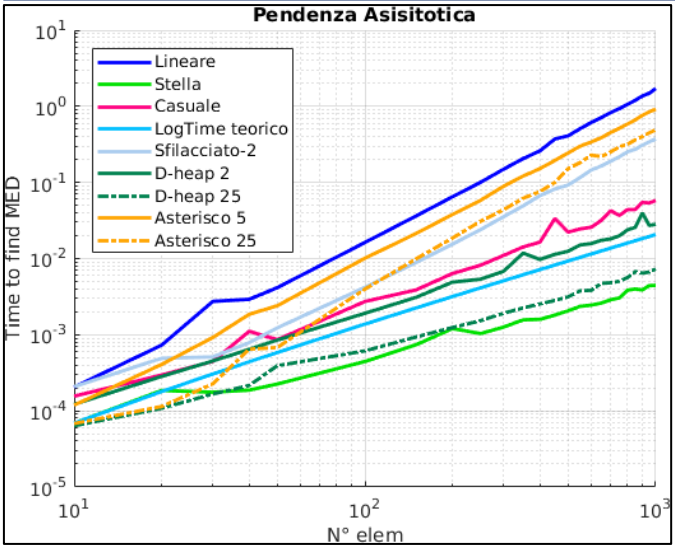
- Il tempo per lo svolgimento dell'algoritmo sarebbe $O(m*n)$, con il caso particolare $m=n-1$ archi ➔ $O(n^2)$ a causa della struttura semplice e connessa dei grafi *aciclici*.
- Al crescere del numero di nodi terminali i, il tempo di esecuzione diminuisce secondo la formula $O(n*(n-\alpha))$, fino al caso di $\alpha = n-1$ nodi "foglia" ➔ $O(n)$ (*stella*).
- È evidente che se il grafo ha una forma sufficientemente frastagliata, allora i tempi decrescono velocemente; ciò è confermato dall'analisi sperimentale di cui sotto, dove il grafo casuale, pur avendo una linea spezzata dovuta alla topologia diversa ogni volta che viene ricreato, ha un andamento molto più vicino alle altre topologie con nodi terminali distribuiti simmetricamente (vedasi *d-heap*).

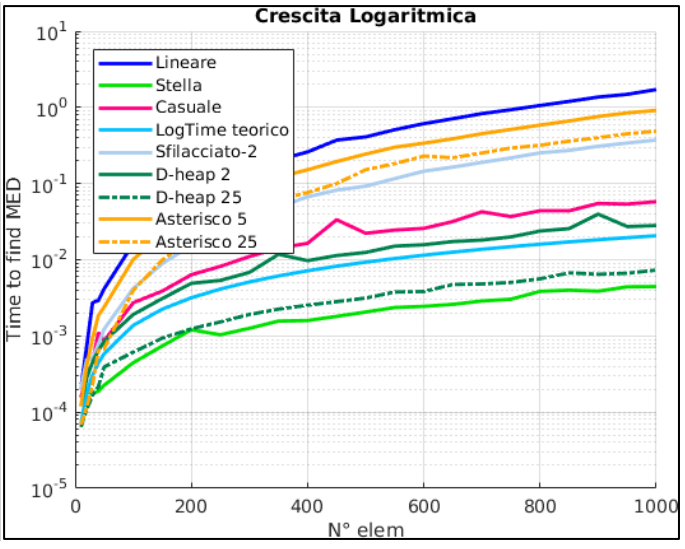
Analisi sperimentale:

Sotto sono riportati i tempi raccolti tramite lo script bash *serialRunner.sh*, il quale crea una suddivisione dei dati e il cProfile per ogni grafo. I dati sono raccolti su Linux, e i grafici risultanti organizzati per viste lineari e logaritmiche, al fine di evidenziare i diversi andamenti temporali.
Sul grafico è stata aggiunta anche la curva $n \cdot log(n)$, referenziata rispetto al primo punto del grafo a *stella*, al fine di dare un'idea dell'andamento dei tempi nei casi intermedi tra *stella* e *lineare*, e ottenere così un metro di paragone per classificarle, almeno nell'intervallo scelto.
Si precisa che gli spike nel grafico sono dovuti al memory management interno di python, e anche l'aggiunta di sleep tra un'esecuzione e l'altra mitiga solo in parte il problema.
Nell'intervallo tra 10 e 50 il campionamento dei dati è a salti di 10, che aumentano a salti da 50 nell'intervallo tra 50 e 1000.

Da una prima analisi si può osservare come nelle forme lineari o comunque molto estese, abbiamo un andamento spiccatamente parabolico (riconducibile a n^2); al contrario, tutte le forme che seguono un qualche tipo di evoluzione simmetrica, hanno un andamento della forma $n^2 - \alpha n$, con α che cresce come il numero di nodi foglia all'aumentare degli assi di simmetria.







Il grafico della **crescita logaritmica** evidenzia in maniera schiacciante che l'analisi teorica dei casi migliore e peggiore è corretta, e che tutte le forme intermedie sono comprese in questa forbice, sia per valori molto piccoli che per valori elevati. È importante osservare che il caso randomico si posiziona al centro di questo range, non molto sopra $n \cdot log(n)$, evidenziando come in grafi a complessità reale questo algoritmo risulti essere alquanto efficiente nel trovare la soluzione del problema in tempi accettabili.
Invece sul grafico della **pendenza asintotica**, essendo in vista doppiamente logaritmica, tutte le funzioni polinomiali saranno rette, con coefficienti angolari che cambieranno in base alla complessità intrinseca della topologia del grafo:

$$stella = 1\log(n) \qquad lineare = \log(n^2) = 2\log(n)$$

È interessante notare come la stella si mantenga in costante allontanamento da $n \cdot log(n)$, mentre i *d-heap* posseggano un andamento simile.
Inoltre dal confronto dei grafici **lineare** e **pendenza asintotica** facciamo notare la differente efficienza dell'algoritmo tra *d-heap* e *sfilacciato-2*: sebbene entrambi posseggano una "frontiera" di foglie alquanto vasta, nello *sfilacciato-2* il guadagno non è così evidente in termini temporali, poiché la crescita delle foglie è lineare ***e non*** esponenziale come nei d-heap (che posseggono una struttura più compatta e dunque più rapida da analizzare).
Si può notare inoltre che l'andamento dello *sfilacciato-2* è parabolico poiché, come è riscontabile nel grafico della **pendenza asintotica,** presenta lo stesso coefficiente angolare del lineare sebbene con una forte costante demoltiplicativa.

Analisi del cProfile:

Abbiamo utilizzato il cProfile built-in al fine di diminuire l'overhead. I dati di cui sotto vengono commentati i risultati è possibile trovarli nelle subDirectory di "formePerRelazione1000".
Le directory il cui nome comprende "son" sono significative solo per le forme d-heap (=fractal), sfilacciato e asterisco.

Lineare 1000 elementi:

Dall'analisi del numero di volte in cui viene chiamato *medNode*, ovvero 998, abbiamo verificato il calcolo teorico che suppone questo numero a n-2.
Inoltre il suo tempo cumulativo è effettivamente quello che incide di più sul totale, il che rende valida la supposizione di analizzare questa parte di codice per stabilirne la complessità totale.

Stella 1000 elementi:

In questo caso *medNode* viene chiamato una sola volta, e sottraendo al tempo totale quello necessario per la creazione del grafo, risulta ancora una volta che nell'esecuzione dell'algoritmo *medNode* è la funzione più influente al fine di calcolare la complessità del codice.

Confronto tra d-Heap da 2(→ 2-Heap) e sfilacciato-2 a 1000 elementi:

Nel caso del 2-heap *medNode* viene eseguito 500 volte, ovvero il numero dei nodi interni, mentre nello sfilacciato-2 333 volte, quindi un terzo del totale.
Verrebbe da chiedersi come mai il 2-heap sia più veloce, nonostante venga eseguito più volte, considerando anche che i rami che l'algoritmo prende in considerazione sono 2 per entrambi.
La spiegazione viene dal fatto che i nodi del 2-heap sono distribuiti in un insieme molto più compatto, avente $\text{diametro}(2log_2(n))$ nodi, mentre lo sfilacciato-2 possiede un $\text{diametro}(n/3)$
La tesi è supportata dal tempo cumulativo di *medNode* riportato nel cProfile:

- Per il 2-heap è pari a 0,026 sec ovvero mediamente 0.052 msec per nodo.
- Per lo sfilacciato-2 è 0,37 sec ovvero mediamente 1.11 msec per nodo.

Questo fa capire come non sia tanto il numero di volte che viene eseguito *medNode* a fare la differenza, bensì la durata dell'esplorazione dei nodi, che nel caso sfilacciato-2 è quasi come un lineare.