Luca Foschiani 109623 foschiani.luca@spes.uniud.it

Il problema da risolvere è il seguente:

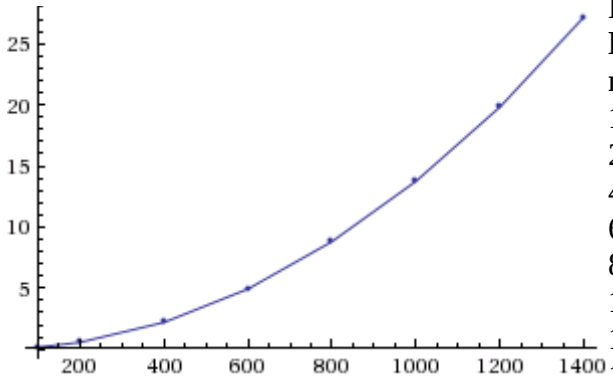
Viene data in input una collezione di grafi non orientati in formato .dot, l'algoritmo deve rendere aciclico ognuno di questi grafi senza sconnetterli (rendendo quindi ogni grafo un albero libero), a questo punto il programma deve individuare, per ogni albero, quale nodo va preso come radice in modo da rendere l'albero in output dell'altezza minima possibile. In output viene dato (sullo standard output) un altro file .dot contenente una collezione di alberi (con gli archi orientati) in cui le radici sono di altezza minima e sono etichettate con la loro altezza all'interno dell'albero.

Ho risolto il problema in questo modo:
-per ogni grafo, lancio una versione leggermente modificata dell'algoritmo DFS visto a lezione.
Il mio algoritmo utilizza un array aggiuntivo (roots) per segnare quali nodi sono già stati visitati (-1), in modo da lanciare la mia versione di DFS il minor numero di volte possibile (che è il numero di grafi presente nella collezione di grafi in input), e qual è il nodo dal quale DFS è partito (1).
Ho aggiunto l'array roots perché altrimenti la complessità sarebbe aumentata troppo volendo anche calcolare il vettore delle distanze in DFS; inoltre la mia versione di DFS è priva del ciclo che prende in esame tutti i nodi del grafo, sostituita dal ciclo for dal quale lancio DFS al quale passo il nodo dal quale iniziare la visita. Ho fatto questa scelta perché trovavo più semplice aggiungere il vettore delle distanze scrivendo DFS in questo modo.
Proseguendo con la visita in profondità, ogni volta che l'algoritmo arriva ad un arco il quale connetterebbe il nodo che sto esaminando ad un nodo già esaminato, elimina quell'arco. Questa soluzione garantisce che il grafo diventi aciclico senza sconnetterlo (ed eliminando il minimo numero di archi), ma non assicura che l'arco eliminato sia quello migliore dal punto di vista di rendere l'albero in output di altezza minima.
Durante la visita utilizzo anche un array delle distanze dal nodo dal quale DFS è partito (d). Questo array mi permette di trovare il nodo più lontano dal nodo iniziale, e di darlo in output tramite il vettore roots in modo da poterlo usare per la successiva chiamata a BFS. Calcolare il vettore delle distanze in questo modo è corretto (mentre per il DFS “classico” non lo sarebbe) perché eliminando i cicli, esiste un solo cammino per ogni coppia di nodi facenti parte del grafo.
-a questo punto BFS viene chiamato su tutti i nodi che in roots hanno valore 1 (che sono i nodi più lontani dalle sorgenti dalle quali DFS è partito). Uso una coda per effettuare la visita in ampiezza, un vettore parent per tenere in memoria i cammini e un vettore delle distanze d. L'ultimo nodo che esce dalla coda è il nodo più lontano dalla sorgente, a questo punto prendo la distanza di questo nodo dalla sorgente (il diametro), lo divido per due e questo valore serve a due cose: sarà l'altezza del mio albero in output (memorizzata in heights) e mi permette di risalire l'albero di visita per trovare il nodo che si trova a metà tra il nodo più lontano e la sorgente (il nodo che trovo è la radice dell'albero in output).
-ora con un'altra chiamata BFS (BFS_tree), prendendo come sorgente il nodo radice del precedente BFS, stampo l'albero in output tramite il vettore parent di BFS_tree.

Il massimo numero di nodi è MAX_NUMBER_OF_NODES, nelle prime righe di asd.c.
Vengono accettati nomi di nodi che siano composti da lettere (maiuscole e minuscole) e numeri.

Utilizzo gcc, quindi da terminale linux nella cartella dei sorgenti per compilare uso
$ gcc -o asd asd.c alg.c queue.c
poi, per eseguire il programma
$ ./asd < input.dot > output.dot

L'algoritmo da me implementato lancia DFS e BFS dall'interno di cicli for, ma di fatto questo serve solo a lanciarli un numero di volte pari al numero di grafi fra loro sconnessi presenti nell'input.
DFS_visit viene lanciato quindi una volta sola su ogni nodo del grafo, mentre i cicli while di BFS e BFS_tree operano anch'essi una volta sola su ogni nodo del grafo; le operazioni sulla coda hanno costo O(1).
La lettura dell'input e il popolamento della matrice hanno (insieme) complessità $O(|V|^2)$.
Nel caso dell'implementazione con matrice di adiacenza la complessità dell'algoritmo è $O(|V|^2)+O(|V|^2)+O(|V|^2)$ quindi $O(|V|^2)$ con V che è l'insieme dei nodi.
Ho usato l'algoritmo presente sulle dispense del corso per calcolare i tempi di esecuzione del mio programma. La funzione che genera il grafo (la matrice) in modo casuale aggiunge sempre esattamente $|V|^2/5$ archi in modo che il grafo sia piuttosto connesso, anche se non dovrebbero esserci grandi differenze variando il numero di archi visto che la complessità dipende dal numero di nodi.



Le misure che ho usato per disegnare il grafico sono le seguenti:

n° di nodi	tempo (ms)
100	0.158517
200	0.561200
400	2.193850
600	4.926650
800	8.804100
1000	13.736750
1200	19.807100
1400	27.164350

Le misure sperimentali sembrano in accordo con la complessità teorica dell'algoritmo visto che sul piano cartesiano formano una curva.