Progetto di Algoritmi e strutture dati
Relazione
a.a. 2015/16

Vlatko Markovski
n. matricola: 112122
markovski.vlatko@spes.uniud.it

Compilazione del codice:
Collocarsi nella cartella Markovski_112122 ed eseguire il comando:
javac Soluzione.java Nodo.java

Problema:
Dato un grafo orientato G=(V,E), diciamo ammette una radice se e solo se esiste un nodo r ∈ V a partire da quale può essere raggiunto (facendo partire una qualsiasi visita) ogni v ∈ V.
1. Determinare il numero minimo di archi da aggiungere a G in modo tale da ottenere G' = (V, E') che ammetta una radice r; vuol dire porsi la domanda : qual è il minimo numero di archi da aggiungere al grafo dato in input in modo da avere un nodo radice?
2. Produrre un albero T= (V, Et) dei cammini minimi in G' di radice r.

Soluzione:
1. Bisogna capire partendo da quale nodo si possono raggiungere più nodi possibili. Per ottenere il nodo di massimo grado di raggiungibilità *j* bisogna far partire la visita BFS da tutti I nodi e prendere quello con la raggiungibilità maggiore. Quel nodo sarà la radice dell'albero dei cammini minimi che va prodotto nel secondo punto. Una volta aver fatto questo, con la visita DFS si rimuovono gli eventuali cicli, facendola partire dal solo nodo *j*. I nodi che rimangono non raggiunti da tale nodo dopo aver effettuato le due visite vengono copiate su un'altra struttura dati, si ripete il procedimento e si prende il prossimo nodo k con raggiungibilità maggiore. Si aggiunge un nuovo arco dal primo nodo j verso k. Il procedimento si ripete fino a quando esistono nodi non esplorati.
2. Per produrre un albero T dei cammini minimi di un grafo orientato non pesato di radice r, basta far partire la visita BFS dalla radice che calcola le distanze. Per mantenere le proprietà dell'albero dei cammini minimi si rifa partire la procedura BFS ogni volta che si aggiunge un nuovo arco nell'insieme E'.

Classe Nodo.java:
Questa classe implementa il concetto di nodo del grafo. Un nodo ha il proprio nome, predecessore, colore, distanza, degree e lista di nodi adiacenti. I nomi, predecessori ed i colori vengono rappresentate con le stringhe, mentre la lista di adiacenza di ogni nodo è un ArrayList<Nodo>.

Classe Soluzione.java:
La classe principale che contiene gli algoritmi della lettura di input, la soluzione del problema e la scrittura dell'output.
Lettura di input:
Il metodo principale che legge l'input e salva i dati è **readTokens**. Usa le espressioni regolari per riconoscere i nodi che vengono passati dallo standard input come stringhe e li salva sulla struttura dati principale sd che è un ArrayList<Nodo>. Una volta che viene riconosciuto un nodo, prima si controlla se è già stato salvato sulla struttura dati principale. Se questo non è vero lo si aggiunge su sd e succesivamente lo si prende con il metodo **getNodo**. Poi va fatto lo stesso lavoro per il secondo nodo che va salvato sia sulla struttura dati principale che nella lista di adiacenza del primo nodo che è a sua volta un ArrayList<Nodo>. Nel caso si legga un nodo senza arco uscente lo si salva direttamente in sd. Per correttezza dell'input si assume che ci sia un arco (due nodi con la strigna "->" come delimitatore) oppure solo un nodo per riga. Dopo si passa a risolvere il problema.

Soluzione del problema:
Il metodo solve è l'algoritmo principale della soluzione. Contiene varie chiamate alle visite viste BFS e DFS a lezione. Tra tutti i metodi quello di complessità peggiore è getMaxDeg. In seguito spiegherò il perché. **getMaxDeg** praticamente chiama |V| volte la visita BFS_mod per capire qual è il nodo che può raggiungere il massimo numero di nodi. BFS_mod è la stessa visita in ampiezza vista a lezione con un ulteriore controllo dei colori dei nodi in modo da aggiornare il campo degree di Nodo. Dopo averla fatta partire da tutti i nodi, BFS_mod ci restituisce quel nodo con degree maggiore (nel caso ci sia più di un nodo con lo stesso degree si prende il primo trovato). Il nodo restituito è la radice dalla quale poi va fatta partire DFS per rimuovere gli eventuali cicli. La versione di DFS che uso è leggermente modificata. Non costruisce una "foresta di alberi" ovvero non va fatta partire da tutti i nodi ma dalla sola radice e nel caso si trovi un ciclo, ovvero si trovi un arco "back-edge", lo si rimuove. Invece la BFS è la stessa versione vista a lezione. Succesivamente nel ciclo while la guardia è il metodo **esisteBianco** che controlla se ci sono nodi non esplorati. Questo si avverà nel caso si passi in input un grafo sconnesso. Dunque tutti i nodi che sono rimasti bianchi vengono copiati su un'altra struttura dati dello stesso tipo ArrayList<Nodo> denominata sd2 e si procede in modo simile. Con il metodo **getMaxDeg2** che lavora sulla struttura dati appena creata si fa partire BFS_mod da tutti i nodi. Fra tutti i nodi di sd2, viene restituito il nodo con degree maggiore denominato max2 che va aggiunto nella lista di adiacenza della radice e nella ArrayList<Nodo> Nuovi che tiene traccia dei nuovi nodi aggiunti alla radice. Succesivamente vanno fatte partire DFS e BFS per rimuovere i cicli e aggiornare i valori. Il metodo esisteBianco controlla se ci sono altri componenti del grafo ancora non aggiunte all'albero che si costruisce.
Scrittura dell'output:
Uso il metodo stampa che scrive sullo standard output in formatto .dot l'albero dei cammini minimi che è stato costruito precedentemente.

Complessità:

Analizziamo il metodo solve. GetMaxDeg chiama |V| volte la visita BFS che è implementata usando le liste di adiacenza e costa O (|V|+|E|) quindi il costo è O (|V| *(|V| + |E|)). Poi ci sono due cicli for che scandiscono sd e dunque entrambi sono di costo $\Theta(|V|)$. Alla fine il costo totale è O (|V|+|E|). Dopo c'è la visita DFS che è modificata rispetto alla versione vista a lezine che non va chiamata da tutti ma da solo uno e dunque costa O (|V|+|E|). EsisteBianco scandisce di nuovo la struttura dati principale finchè non trova il primo nodo bianoc e quindi il costo totale è O (|V|). Quante volte va eseguita dipende dal numero di componenti connesse che ci saranno nel grafo. Nella peggiore ipotesi potrebbero esserci |V| nodi che tutti quanti formano una componente connesa per sé e dunque il costo totale arriva a $\Theta(|V|)^2$. Dentro il ciclo while c'è un altro ciclo for che scandisce sd e quindi il costo del for è $\Theta(|V|)$. La sd2 nuova che viene creata potrebbe contenere fino a |V|-1 nodi. GetMaxDeg2 va eseguita su sd2 e costa O((|V|-1) *(|V|-1 + |E|)). Dopo ci sono le chiamate alle visite DFS e BFS che costano entrame O(|V|+|E|). Quindi il costo complessivo della procedura solve è O(|V|) * O((|V|-1)*(|V|+|E|)).

Purtroppo non sono riuscito a misurare i tempi effettivi della mia soluzione.