Relazione Elaborato 1

Nome: Diego Mazzieri, Milo Marchetti

Matricola: 0000792583, 0000793241

IDE: Visual Studio 2017, Qt Creator 4.3.1

Linguaggio: C++

Deadline: 01/08/2017

Analisi del problema

Il problema della **Betweenness Centrality** richiede di trovare per **ogni nodo** di un grafo un **indice** che rappresenta la sua centralità, ossia **quante volte** si **passa per esso** percorrendo i **cammini minimi** che uniscono ogni nodo ad ogni altro.

Le **complicazioni** nella risoluzione risiedono nel fatto che spesso è presente **più di un cammino minimo** per arrivare da un nodo ad un altro. In casi del genere, è fondamentale prendere in considerazione il fatto che essendo indifferente scegliere un cammino piuttosto che un altro, è necessario incrementare l'indice di ogni nodo proporzionalmente al numero dei percorsi che passano per esso.

Ulteriore dettaglio da tenere in considerazione è che i **grafi** per cui si viene richiesto di risolvere il problema sono sia **orientati** che **pesati**.

Librerie standard

Al fine di semplificare l'implementazione della soluzione soprattutto per le operazioni di input/output su file e per la rappresentazione delle strutture dati sono state utilizzate le seguenti librerie standard del linguaggio C++:

- **iostream**, input/output su terminale;
- **fstream**, input/output su file;
- **iomanip**, manipola la formattazione dell'input/output sia su terminale che su file
- **time.h**, acquisizione e manipolazione del tempo (utile per il benchmark)
- **limits.h**, macro con i minimi/massimi valori rappresentabili per ogni tipo di variabile
- **vector**, gestisce array di dimensione variabile

Header

Il file **betweenness-centrality.h** contiene, oltre alle definizioni delle funzioni che vengono utilizzate dal programma, una macro (**#define INF INT32_MAX**) che assegna come distanza tra due nodi non raggiungibili il massimo intero rappresentabile con 32 bit e due definizioni di tipo (**typedef int** AdjacencyMatrix, typedef vector<vector<vector<int>>> Predecessors**) utili a dare nomi significativi rispettivamente alla **matrice di adiacenza** associata ad un grafo ed alla **matrice dei precedessori** calcolata con l'algoritmo di Floyd-Warshall.

Strutture Dati

Per la rappresentazione del grafo è stata usata una **matrice di adiacenza** piuttosto che una lista di adiacenza per rendere più immediato l'utilizzo dell'**algoritmo di Floyd-Warshall**, che richiede ripetutamente l'accesso ai cammini tra due nodi qualsiasi per confrontare i pesi e eventualmente aggiornarli.

Per ovviare alle complicanze discusse nella sezione Analisi del problema, la **matrice dei predecessori** è stata implementata in modo non canonico: per ogni coppia di nodi (i, j) non necessitiamo di un singolo nodo predecessore, bensì di un numero di predecessori che dipendono da quante varianti con lo stesso peso abbiamo di un cammino minimo da i a j. Abbiamo quindi utilizzato una **matrice di vettori**, dichiarata **statica** perché usata anche dalla funzione betweennessCentrality, le cui dimensioni vengono incrementate o ridotte durante l'esecuzione di Floyd-Warshall ogni volta che un peso viene aggiornato con uno minore.

Al termine dell'esecuzione dell'algoritmo, la matrice dei pesi conterrà alla riga di indice i e alla colonna di indice j il peso del cammino minimo per arrivare a j partendo da i mentre la matrice dei predecessori conterrà alla **riga i** e alla **colonna j** un **vettore** con i **predecessori del nodo j** per tutti i **cammini minimi da i a j**.

Per memorizzare gli indici di Betweenness Centrality è stato usato un array allocato dinamicamente di float.

Funzioni e relativi algoritmi utilizzati

initGraph:

Funzione statica che **alloca** una **matrice di adiacenza vuota** e che si occupa di riempirla con valori standard: "**0**" se da un nodo dobbiamo raggiungere il nodo stesso (i=j) e **INF** in tutti gli altri casi; questo perché non abbiamo ancora analizzato nessun arco del grafo.

nodePredContains:

Funzione statica che **verifica** la **presenza** di un **nodo** all'interno di "predNodes", parametro passato alla funzione che rappresenta un puntatore ad un **vettore** di interi contenente i **predecessori**.

Floyd-Warshall:

Per trovare i **cammini minimi tra tutte le coppie** di nodi è stato usato l'algoritmo di Floyd-Warshall, che costruisce in tempo $O(|V|^3)$, dove $|V|$ è il numero di nodi del grafo, una matrice $|V| * |V|$ contenente alla posizione (i, j) il costo del cammino minimo dal nodo i al nodo j e parallelamente una matrice $|V| * |V|$ dei predecessori contente alla posizione (i, j) il predecessore del nodo j nel cammino minimo da i a j.

La matrice dei pesi è stata creata con initGraph per non alterare quella passata come parametro alla funzione ma ai fini del programma avremmo potuto anche modificare direttamente la matrice di adiacenza associata al grafo.

La funzione **restituisce** alla chiamante betweennessCentrality un **puntatore** alla **matrice dei predecessori**, in quanto è necessaria solo questa per calcolare gli indici di Betweenness Centrality dei nodi del grafo.

betweennesCentrality:

Funzione che si occupa di calcolare per ogni nodo il suo indice di Betweennes Centrality. Il calcolo, eseguito attraverso il richiamo della funzione shortestPathVisit, viene eseguito solo **in seguito all'esecuzione** dell'algoritmo di **Floyd-Warshall**, quando i cammini minimi tra tutte le coppie di nodi sono già definiti e non vengono aggiornati ad ogni iterazione. Questo permette di non dover continuamente decrementare gli indici relativi ai nodi quando un cammino, considerato minimo fino a quel momento, viene sostituito da uno più breve.

shortestPathVisit:

La funzione **ricorsiva** shortestPathVisit, chiamata per ogni coppia di indici (i, j) della matrice dei predecessori all'interno della funzione betweennessCentrality, **percorre tutti i cammini minimi** dal nodo i al nodo j operando gli spostamenti all'interno della riga i in modo **analogo** ad una **DFS** (Depth First Search): **dal nodo di arrivo j** torno **indietro** ricostruendo tutti i cammini minimi al contrario **fino** ad arrivare **a i**. Poi faccio **backtracking** da i finché non ritorno a j e non sono presenti altri cammini. Ogni volta che passo per un nodo incremento l'indice ad esso corrispondente nell'array **nodesPathsCount**, in cui, alla fine della ricorsione, saranno contenuti il **numero di cammini minimi da i a j passanti per ogni nodo**.

readGraph:

Funzione che **legge il file** di input e che, dopo aver allocato una **matrice di adiacenza** attraverso una chiamata ad initGraph, si occupa del **riempimento** di quest'ultima utilizzando i valori che acquisisce dalla lettura del file.

writeNodesBetCentrality:

Funzione che **scrive su file** i nodi e i relativi **indici di Betweenness Centrality** approssimati alla **sesta cifra decimale**.

delGraph:

Funzione che si occupa della **deallocazione** di una qualsiasi **matrice di adiacenza** passatagli come parametro.

Complessità

Il calcolo della complessità del programma è strettamente **dipendente** dalla **struttura del grafo** e quindi **non** è determinabile **a priori**, in quanto non si è a conoscenza della **quantità** di **cammini minimi** presenti e di quali archi del grafo appartengono ad almeno uno di essi. Possiamo comunque trattare il caso ottimo e il caso pessimo:

- Il **caso ottimo** è quello in cui vi è al massimo **un solo cammino minimo** che unisce **ogni coppia** di **nodi**. In questo caso, infatti, la **complessità spaziale**, data dalla matrice di vettori corrispondente ai predecessori, risulta $\boldsymbol{O}(|\boldsymbol{V}|^2)$, in quanto ogni vettore dei predecessori è unitario; mentre la **complessità temporale** è $\boldsymbol{O}(|\boldsymbol{V}|^3)$, data dall'esecuzione dell'algoritmo di Floyd-Warshall.

- Il **caso pessimo** è quello in cui, presi **due nodi qualsiasi**, tutti i **cammini minimi** che li uniscono sono **composti** da **tutti gli archi** del grafo. In questo caso, la **complessità spaziale** è data dalla matrice dei predecessori che, contenendo per ogni riga tanti elementi quanti sono gli archi, ha complessità $\boldsymbol{O}(|\boldsymbol{V}| * |\boldsymbol{E}|)$; mentre la **complessità temporale** è data dalla shortestPathVisit che, richiamata $|V|^2$ volte, impiega al massimo $|E|$ salti prima di chiudere la ricorsione, da cui risulta $\boldsymbol{O}(|\boldsymbol{V}|^2 * |\boldsymbol{E}|)$.

Benchmark

I test sono stati svolti con Visual Studio 2017 in modalità Release su un processore quad-core Intel Core i5-4670K che opera alla frequenza di 3.40Ghz. I tempi di esecuzione sono relativi alla sola funzione betweennessCentrality; sono quindi esclusi i tempi di lettura e scrittura su file del programma.