Relazione Algoritmi e Strutture Dati

Matteo Scala

Luglio 2018

1 Introduzione

Avendo scelto di sviluppare l' elaborato numero 1, il mio programma deve calcolare i centri di un grafo fornito tramite file. Per fare ciò, ho deciso di suddividere il programma tendenzialemente in 4 parti. Apertura e lettura del file, definizione della struttura dati "grafo", calcolo delle distanze dei nodi e seguentemente dei centri, scrittura del risultato su file.

2 L' Input

Nella prima parte tutto il contenuto del file viene caricato su una stringa attraverso una funzione che prima calcola il numero di caratteri (e quindi la dimensione) del file, alloca sufficiente memoria per tutti i caratteri, poi con `fread` legge tutto il file e mette il contenuto nella stringa appena allocata. Alla fine ritorna al `main` il puntatore della stringa.

3 Creazione Grafo

Per la implementazione della struttura "grafo" ho deciso di adottare la classica struttura con liste di adiacenza; la sceltra tra questa e la rappresentazione con matrice è stata condizionata principalmente dal fatto che ritengo le liste di adiacenza più facili da utilizzare e perche in caso di grafi con elevato numero do nodi, poteva capitare che non si riuscisse ad allocare sufficente memoria contigua per la matrice. In paricolare, i nodi sono memorizzati con una lista concatenata (non ordinata) e ad ogni nodo è attaccata un altra lista concatenzata per rappresentare gli archi a cui è collegato.

```
struct GraphADS{
        struct GraphNode * graph;
        int nodes;
        int archs;
};
struct GraphNode{
        int tag;
        int eccentricita;
```

```
        struct AdjacencyListElem * al;
        struct GraphNode * next;
};
struct AdjacencyListElem{
        int nodeTag;
        struct AdjacencyListElem * next;
};
```

Le strutture sopra rappresentate servono, rispettivamente, per gestire meglio il grafo, rappresentare la lista dei nodi, rappresentare una lista di nodi adiacenti ad un nodo specifico. Considerando che le tre la lista concatenata per i nodi e quella concatenata per le liste di adiacenza ed occupano rispettivamente 16, 24 e 16 byte; si può facilmente calcolare che il grafo occuperà 16 + 24*numeroNodi + 16*numeroArchi*2 byte di memoria (es. tra le istanze già fornite, il grafo più piccolo occuerà circa 5.5KB). Per estrapolare le informazioni dal file, ho scelto un approccio in più fasi. Per prima cosa, divido la stringa (quella col contenuto del file) nelle righe che la compongono (divido sugli "\n"); questo per poi rendere piu facile la conversione delle singole righe in array di interi, che sono più facilemente gestibili nel resto del programma. A questo punto, dopo aver convertito le stringhe in array di interi, scorrendo tutto l' array che contiene le righe trasformate in array, genero il grafo. Il processo è piu o meno questo: salto la prima riga (dato che contiene informazioni più generali sul grafo), con un ciclo, controllo se il primo ed il secondo numero sono già inseriti nella lista dei nodi (il terzo numero lo ignoro, siccome nella consegna del progetto non si parla di grafo pesato), se con ci sono ovviamente li aggiungo, poi estendo le liste di adiacenza dei due nodi con il nodo a cui sono attaccati; questo per tutti i valori nel file. Il risultato è l' insieme dei nodi e delle liste di adiacenza che rappresentano il grafo.

4 Calcolo di Eccentricità e Centri

Una volta che ho a disposizione la struttura grafo, pero ogni nodo calcolo l' eccentricità. Anche quì, divide et impera. Prima di tutto implemento la classica *Breadth-First Search* con la quale estrapolo le distanze da un nodo da tutti gli altri. Ovviamente, per l' implementazione serve una coda che ho implementato con la struttura più semplice possibile:

```
struct SimpleQueue{
        int * elements;
        int elem_num;
};
```

Per quanto riguarda il BFS, ho creato una funzione che in input prende grafo e nodo da considerare sorgente, inizializza due vettori (uno per i nodi già visitati e uno per le distanze da quello sorgente), poi partendo dal nodo sorgente, mette in coda tutti i nodi adiacenti, ne calcola le distanze della sorgente e li tagga come visitati, questo per tutti i nodi del grafo (finché ce ne sono in coda). Con questa funzione calcolo l' eccentricià

di ogni nodo (che per definizione è la distanza dal nodo più lontano). Con un' altra funzione poi calcolo qual è l' eccetricità minore e quali sono i nodi con tale eccentricità (sempre per definizione, essi saranno i centri).

5 Output

In fine, una volta che ho collezionato tutti i centri con un array, non faccio altro che scriverli su file, semplicemente con le funzioni `fopen` per aprire il file e un `while` per scorrere i valori dell' array e stamparli su file separati da virgole, come richiesto.

6 Prestazioni

In generale ho cercato di scrivere il codice nella maniera più chiara e riutilizzabile possibile, senza privilegiare troppo le performance. Quindi, per concludere e per dare comunque un' idea dei risultati ottenuti mostro questa tabella sulle prestazioni.

Nome File	Num. Nodi	Num. Archi	Tempo Esecuzione	Memoria occupata
g100sparse.dat	100	1000	0.01	4648
g100dense.dat	100	7000	0.03	5400
g1000sparse.dat	1000	9000	3.22	13500
g1000dense.dat	1000	500000	80.82	78780

Tabella 1: *Test eseguiti con una CPU i7 8550u, 16GB di RAM DDR4, programma compilato con GCC ed eseguito su Linux Ubuntu. I tempi non considerano ne la lettura del file di input, ne la scrittura del file di output e sono espressi in secondi. L' occupazione di memoria si riferisce alla memoria viruale ed è espressa in kilobyte*