definition

Relazione del Progetto di Laboratorio di ASD

Gabriele Venturato (125512)
venturato.gabriele@spes.uniud.it

01 dicembre 2016

Indice

1 Introduzione

1.1 Assunzioni

Per la realizzazione del progetto ho utilizzato il linguaggio C. I test di correttezza e per le misurazioni sperimentali sono stati effettuati su un laptop Lenovo E540 con CPU Intel Core i5–4210M (3,2 GHz), 12GB di RAM, sul sistema operativo Fedora 24 64-bit.

In aggiunta a quando scritto nel documento che descrive il progetto ho ritenuto opportuno fare le ulteriori seguenti assunzioni:

- ogni vertice contiene un'etichetta che corrisponde alla stringa alfanumerica che lo identifica nel formato dot. L'etichetta in questione può avere lunghezza massima di 127 caratteri. Considerando che ci sono 36 caratteri alfanumerici (26 lettere + 10 numeri) mi è sembrato un limite sufficientemente ampio 36^{127}
- i vertici del grafo vengono numerati per uso interno al programma con un id di tipo `int`, perciò il numero massimo di vertici è 2.147.483.647

1.2 Compilare ed Eseguire

Per compilare il programma principale (quello che risolve il problema) è sufficiente usare il seguente comando dalla root del progetto

```
$ gcc -o project project.c myprojectlib.c -I.
```

Se non dovesse funzionare, il comando

```
$ gcc -std=gnu99 -o project project.c myprojectlib.c -I.
```

risolve i problemi di compatibilità di versione. Il file eseguibile sarà poi `project` e si potrà lanciare con il comando nel formato seguente:

```
$ ./project < in.dot > out.dot
```

Qualora fosse necessario compilare il sorgente del programma per l'analisi dei tempi, eseguire dalla sottocartella *times_ analysis/* il comando:

```
$ gcc -o times times.c -lm ../myprojectlib.c
```

allo stesso modo di prima se dovessero presentarsi degli errori, provare con il comando con l'opzione `-std=gnu99`. L'eseguibile sarà poi `times` e dovrà essere lanciato nel formato:

```
$ ./times <graphcase>
```

dove `graphcase` è uno devi valori: worst, average, best. Eventuali dettagli aggiuntivi per la personalizzazione dei test si possono trovare nel sorgente, in particolare modificando i valori delle costanti globali.

2 Il Problema

L'obiettivo di questo progetto è quello di, dato un grafo $G = (V, E)$ in input nel formato dot, individuare prima di tutto un *vertice radice*, definito come un vertice v dal quale si raggiungono tutti gli altri vertici del grafo. Qualora non fosse possibile identificare tale vertice, si deve procedere con l'aggiunta del minimo numero di archi necessario a garantire l'esistenza della radice. Il grafo $G = (V, E')$ così costruito viene detto che *ammette radice* v.

In seconda fase bisogna costruire l'albero $T = (V, E_T)$ dei cammini minimi dalla radice precedentemente individuata, calcolando per ogni vertice la profondità in cui esso si trova nell'albero.

3 La Soluzione

3.1 Strutture dati

Prima di tutto è stato necessario pensare a delle strutture dati per poter gestire la computazione in maniera ottimale. Per la memorizzazione del grafo ho scelto di utilizzare le liste di adiacenza, e le ho implementate in C con delle struct:

```
typedef struct vertex_T {
  struct vertex_T *next;  // next vertex in the list
  struct edge_T *edges;   // pointer to the edges list
  struct edge_T *tedges;  // pointer to the transposed edges list

  char label[MAX_LABEL_LENGTH];
  ...
} vertex;

typedef struct edge_T {
  struct vertex_T *connectsTo;
  struct edge_T *next;
  char color[32];         // to manage the color in output dot file
  char style[32];         // to manage the style in output dot file
} edge;

typedef struct graph_T {
  struct vertex_T *vertices;
} graph;
```

La struct di tipo `graph_T` serve solamente a livello logico per conservare il puntatore alla lista di vertici. La struct di tipo `vertex_T` è quella che definisce ogni vertice e ne conserva le relative informazioni. Per ogni vertice infince esiste una lista di adiacenza, ovvero una lista di archi definita dalla struct `edge_T` che contiene solamente i puntatori ai vertici a cui il vertice proprietario è adiacente.

Ho utilizzato poi altre strutture dati secondare che si possono trovare nel codice, in particolare

- `sccset`, `scc` per la gestione delle componenti fortemente connesse
- `vlist` per la gestione di liste di vertici utili in diversi punti del codice
- `vqueue` per la gestione della coda di vertici nella visita BFS

ulteriori dettagli si possono trovare nel sorgente del programma.

3.2 Algoritmo

L'idea di base dell'algoritmo per la soluzione è molto semplice e si suddivide in quattro fasi distinte spiegate di seguito.

3.2.1 Acquisizione dell'input

Le funzioni per il parsing dell'input prendono lo streaming di dati da `stdin` e analizzano ogni riga, carattere per carattere, alla ricerca dei nomi dei vertici. Per ogni riga si possono trovare al più due vertici. Se ne viene trovato solo uno lo si aggiunge e si passa alla riga successiva. Se ne vengono trovati due può essere solamente dovuto al fatto che nella riga si esprime l'adiacenza tra due nodi. Per questo motivo vengono aggiunti entrambi, e in seguito viene aggiunto l'arco che li collega. L'arco viene aggiunto in testa alla lista per non peggiorare la complessità.

Il costo computazionele di questa procedura è quello che peggiora il costo di tutta la soluzione proposta, e il motivo principale è dovuto alla necessità di dover controllare che i vertici in input non siano già stati inseriti. In questo modo per ogni riga di input si scorre una lista di $O(V)$ elementi (si veda la sezione 3.4).

3.2.2 Componenti fortemente connesse (SCC)

Nella prima fase di processo dei dati ricevuti in input vengono individuate le *Componenti Fortemente Connesse (SCC)* del grafo, e vengono inserite nella struttura dati apposita.

Quello che si ottiene alla fine è un insieme di componenti fortemente connesse. Ogni elemento base della struttura dati che le gestisce contiene un puntatore al vertice radice rappresentante della componente. Per garantire efficienza e una buona complessità ogni vertice punta alla SCC a cui appartiene (compreso il rappresentante). In questo modo in tempo $O(1)$ si riesce a risalire a quale componente un vertice v appartiene. L'insieme di SCC restituire da questa procedura viene dato in input alla fase successiva dell'algoritmo.

3.2.3 Archi mancanti

Questo è il cuore dell'algoritmo in cui si individua la *radice del grafo* e si aggiungono gli eventuali archi mancanti. Il processo inizia con una chiamata a una sottoprocedura che si occupa di controllare la *raggiungibilità* delle varie componenti fortemente connesse.

Definizione 1 (di raggiungibilità) *Definiamo come* raggiunta *una componente fortemente connessa S di un grafo $G = (V, E)$ se e solo se $\exists v \notin S$ tale che per qualche vertice $u \in S$, $\exists (v, u)$.*