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 seguenti ulteriori 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
- 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 orientato* $G = (V, E)$ 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 infine 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 secondarie 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.

Ogni struttura dati tiene conto dell'allocazione di memoria e della sua deallocazine, per mantenerne un utilizzo minimo. Solamente la struttura dati del grafo non viene deallocata nel programma principale perchè viene utilizzata fino alla fine, però ho fatto dei miglioramenti in merito nel programma delle misurazioni sperimentali (si veda la sezione 4).

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 vertici. 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.

Algoritmo 1 Componenti Fortemente Connesse

```
1: procedure SccFinder(G)
2:     Dfs(G)                          ▷ calcola i il tempo di fine per ognu vertice u
3:     G^T ← TransposeGraph(G)
4:     s ← DfsScc(G^T)                 ▷ considerando i vertici in ordine di fine visita
5:     return s                                          ▷ ritorna l'insieme delle scc
```

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 $\Theta(1)$ si riesce a risalire a quale componente un vertice v appartiene. L'insieme di SCC restituite 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 (Raggiungibilità di una SCC)**.** 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 $u \in S$ esiste $(v, u) \in E$.

Quello che fa nella pratica la procedura `scc_reachability()` è andare ad impostare il booleano `isreached` nella struttura dati per la gesione delle componenti. Di default alla creazione viene impostato a $false$, viene quindi eseguita la procedura di verifica che in $\Theta(|V|)$ aggiorna a $true$ le componenti raggiunte.

Algoritmo 2 Raggiungibilità SCC

```
1: procedure SccReachability(G)
2:     for each v ∈ G.V do
3:         i ← scc[v]                                  ▷ recupero la SCC di v
4:         for each u ∈ Adj[v] do
5:             j ← scc[u]                              ▷ recupero la SCC di u
6:             if i ≠ j then          ▷ esiste l'arco (v, u), non è la stessa SCC
7:                 j.reached ← true                    ▷ la identifico come raggiunta
```

Una volta eseguita questa prima fase, vengono poi effettivamente aggiunti gli archi mancanti nel grafo (di colore rosso, come specificato nel pdf del progetto) in questo modo: la prima componente non raggiunta viene eletta a *root*, e viene collegata a tutte le altre non raggiunte attraverso i rappresentanti delle componenti.

Algoritmo 3 Aggiunta archi mancanti

```
1: procedure AddMissingEdges(G)
2:     SccNotReached ← SccReachability(G)          ▷ lista di comopnenti
3:     root ← pop(SccNotReached)                   ▷ prendo la prima scc come radice
4:     for each scc ∈ SccNotReached do
5:         v ← scc.vertex                          ▷ recupero il rappresentante della SCC
6:         AddEdge(root, v, G)                     ▷ aggiungo l'arco (root, v) a G
```

3.2.4 Albero dei cammini minimi

La procedura per la costruzione dell'albero dei cammini minimi è di fatto una *BFS visit* inizializzata dal vertice *root*, che per quanto già definito, necessariamente raggiungerà tutti i vertici del grafo. La BFS sfrutta la struttura dati di tipo *queue* che inserisce in coda e toglie dalla testa. Per mantenere una complessità di $\Theta(1)$ per le operazioni di `push` e `pop` ho sfruttato i puntatori forniti dal linguaggio mantenendone uno che punta all'inizio e un altro alla fine.

La costruzione dell'albero viene effettuata solamente cambiando lo stile degli archi da quello di default a *"dashed"* (come richiesto nel documento del progetto). Oltre a costruire l'albero, viene anche calcolata la profondità di ogni vertice. Questo esaurisce la soluzione al problema in ogni suo aspetto.

3.3 Correttezza

La correttezza di quasi tutte le procedure utilizzate è già affrontata dettagliatamente nel libro di testo, ed è stata vista anche a lezione durante l'anno. L'implementazione degli algoritmi nel mio programma si discosta dalla definizione del libro solo per dettagli che non influenzano quanto già dimostrato. In particolare:

- *DFS* nelle sue due varianti `DFS` e `DFS_SCC` hanno delle differenze puramente implementative
- *BFS* garantisce la costruzione dell'albero dei cammini minimi con la giusta profondità assegnata ai vertici per costruzione dell'algoritmo.

Resta da dimostrare quindi solo che la procedura `add_missing_edges()` ritorni effettivamente un grafo che *ammette una radice*, al più aggiungendo il *minimo* numero di archi.

Per dimostrarlo assumo noto il concetto di *componenti fortemente connesse* e verifico prima di tutto la correttezza di `scc_reachability()` che viene richiamata nella prima fase.

I passi eseguiti sono semplici (si veda l'Algoritmo 2): per ogni vertice v si scorre la sua lista di adiacenza, e per ogni $u \in Adj[v]$ si verifica (in tempo $\Theta(1)$) a quale SCC essi appartengono. Se $scc[v] \neq scc[u]$ significa che la componente di u è raggiunta dalla componente di v, e quindi si aggiorna lo stato di questa a *"raggiunta"*. Questo viene fatto per ogni vertice del grafo, e quindi è immediato che ogni componente raggiunta venga segnalata come tale.

A questo punto lo scopo è trovare un vertice dal quale raggiungo tutti gli altri. Si osservi che in una SCC ogni vertice raggiunge tutti gli altri. Le componenti *raggiunte* non vengono prese in considerazione perchè se sono già raggiungibili da un'altra componente, allora non sono le candidate ideali per garantire l'eventuale aggiunta del minimo numero di archi.

Considero allora solamente le componenti non raggiunte e osservo che esiste sempre almeno una SCC in un grafo che è il grafo stesso, e questa è banalmente non raggiunta da nessun'altra.

A questo punto: se è solo una, scelgo il suo rappresentate come radice del grafo e ho finito senza dover aggiungere archi. Se invece c'è più di una componente non raggiunta, ne scelgo *indifferentemente* una e uso il suo rappresentante come radice del grafo. Utilizzo poi questo vertice *root* per aggiungere tutti gli archi tra *root* e i rappresentanti delle componenti non raggiunte.

In questo modo ho esaurito tutti i casi e alla fine della procedura avrò un vertice *root* dal quale raggiungo tutti gli altri vertici del grafo.

Osservazione 1 (Il numero di archi aggiunti è minimo). É immediato dalla correttezza di `scc_reachability()` che se ci sono n componenti non raggiunte (compresa quella di cui è rappresentante la radice), il minimo numero di archi da aggiungere per raggiungerle tutte da *root* è esattamente $n - 1$. In caso contrario qualche componente rimarrebbe isolata.

Osservazione 2. In generale la soluzione al problema non è unica. Sia per l'arbitrarietà della scelta della componente il cui rappresentante sarà la radice del grafo, sia perchè ogni vertice in una SCC è mutualmente raggiungibile e ciò implica che ogni vertice della componente di cui *root* è il rappresentante può essere a sua volta radice del grafo.

3.4 Complessità

La soluzione proposta è sostanzialmente divisa in tre parti:

`SCC_finder()` è composta di due DFS con costo $\Theta(|V|+|E|)$, e una funzione `transpose_graph()` con lo stesso costo.

`add_missing_edges()` ha costo $\Theta(|V| + |E|)$ perchè nonostante operi in tempo $O(|V|)$, richiama al suo interno la procedura `scc_reachability()` la cui complessità ne determina quella effettiva.

`BFS()` la cui complessità in generale è $O(|V| + |E|)$, nel contesto di utilizzo della soluzione proposta visita necessariamente tutto il grafo, è quindi più precisamente $\Theta(|V| + |E|)$.

Ha quindi una complessità lineare sulla dimensione del grafo $\Theta(|V| + |E|)$, che nel caso peggiore di un grafo completo corrisponde a $\Theta(|V|^2)$.

Però, come anticipato nella sezione 3.2.1, c'è la procedura di acquisizione dell'input che aggrava la complessità totale perchè per ogni riga di input si deve controllare se un vertice è stato già insterito in una lista di lunghezza $O(|V|)$. Considerando che per descrivere il grafo si utilizzano $O(|V|+|E|)$ righe, il costo finale è $O(|V|(|V| + |E|)) = O(|V|^2 + |V||E|)$.

Questo vuol dire che anche nel caso migliore in cui il grafo $G = (V, E)$ di input ha $E = \emptyset$, la complessità totale è comunque *quadratica*. Nel caso peggiore invece, in cui il grafo è completo si ha $|E| = |V|^2$, perciò il costo è *cubico*.

4 Misurazioni sperimentali

Le misurazioni sono state eseguite con gli algoritmi descritti nel pdf degli appunti del laboratorio di ASD. In particolare si è fatto uso della procedura `misurazione()` con una tolleranza dell'errore imposta al 5%, campioni di 20 elementi, e un intervallo di confidenza del 95%. Le ripetizioni dell'algoritmo sono state effettuate su input da dimensione 0 fino a un massimo di 150 vertici.

La ragione di tale limite è stata imposta dai limiti hardware della macchina su cui ho eseguito i test.

Nella versione del programma in cui sono stati eseguiti i test sui tempi ho provveduto all'implementazione di una procedura che dealloca la memoria utilizzata da ogni grafo, nei limiti del linguaggio (e delle mie conoscenze relative ad esso). Nonostante questi accorgimenti, andando oltre i 150 vertici la memoria occupata era quasi di 10GB, e per evitare che iniziasse lo *swapping* in memoria secondaria ho preferito imporre questo limite. Le misure rilevate mi sono sembrate ad ogni modo sufficienti per fare le dovute considerazioni.

Per la generazione degli input ho distinto in casi:

- per il caso *ottimo* ho scelto un grafo senza archi perchè è il caso in cui l'esecuzione dell'algoritmo è più veloce
- per il caso *medio* ho utlizzato l'algoritmo per la generazione di numeri pseudo-casuali proposto a lezione
- per il caso *pessimo* invece, ho generato un grafo completo

A conferma dell'analisi analitica della complessità, si può vedere in Figura 1 che il caso ottimo è comunque quadratico, nonostante le procedure "cuore" della soluzione non lo siano. Tale andamento è giustificato dal modo in cui è stato effettuato il parsing del grafo in input.

In Figura 2 invece si può apprezzare la differenza nell'andamento dei tre casi: il caso pessimo è di ordine cubico, il caso ottimo viene riportato dal caso precedente in proporzione, e il caso medio si colloca esattamente tra i due.

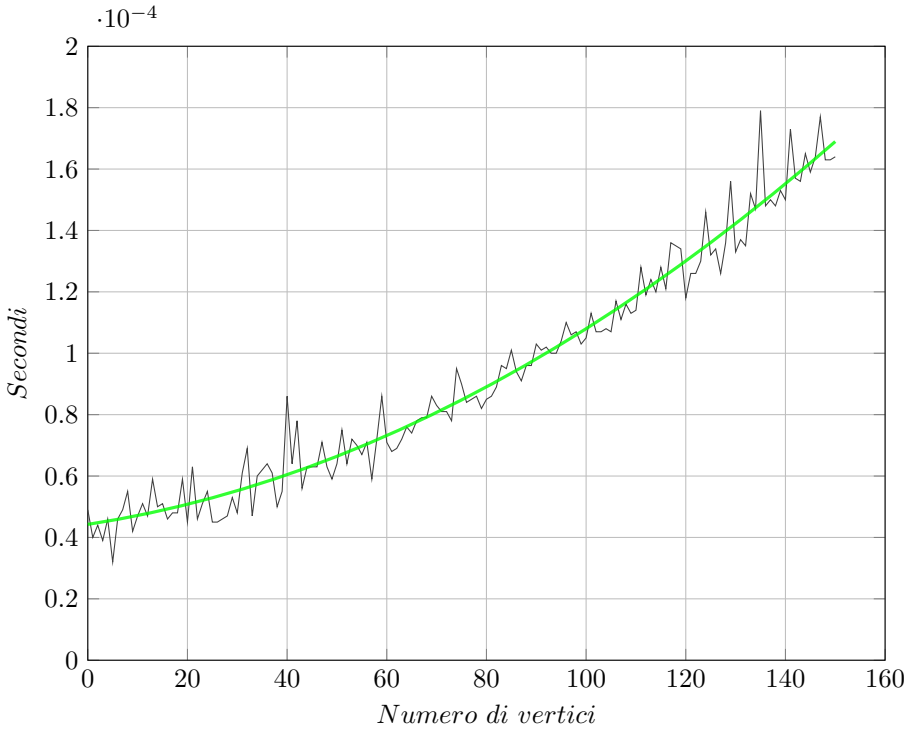

Figura 1: Misurazione nel caso ottimo

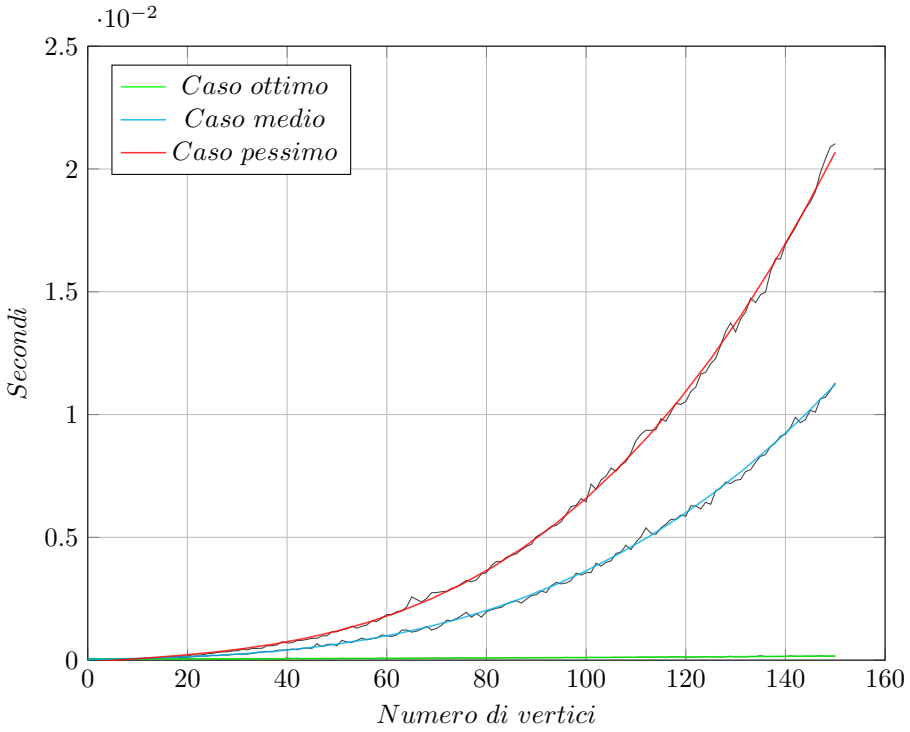

Figura 2: Confronto delle misurazioni nei tre casi

5 Conclusioni e osservazioni personali

Per lo sviluppo di questo progetto ho cercato di applicare al meglio le mie conoscenze relative al linguaggio C utilizzato, ma soprattutto le nozioni apprese durante il corso di Algoritmi e Strutture Dati. É stato utile e interessante applicare gli algoritmi studiati teoricamente, e verificarne sperimentalmente i risultati sia in termini di correttezza che di tempi effettivi di esecuzione.

Le principali difficoltà sono state all'inizio, nella strutturazione di una soluzione semplice ma che tenesse conto dell'efficienza. Ma anche nella scelta delle strutture dati più opportune e nella loro ottimizzazione, cercando di sfruttare le potenzialità del linguaggio, che offre una gestione diretta della memoria attraverso puntatori. Ulteriori osservazioni di minore importanza, in merito ad eventuali perfezionamenti, si possono trovare nel sorgente del programma tra i commenti dedicati ad ogni procedura.