UNIVERSITÁ DI UDINE

INFORMATICA

ANNO ACCADEMICO 2015-2016

Relazione del progetto di laboratorio di

Algoritmi e Strutture Dati

Studente:
MICHELE COLLEVATI

Matricola:
116286

Mail:
collevati.michele@spes.uniud.it

Indice

Capitolo 1

Introduzione

La presente relazione tratta del progetto di Algoritmi e Strutture Dati assegnato dal Prof. Alberto Policriti nell'anno accademico 2015/2016, e fa riferimento alle specifiche fornite nel documento *"progetto_ASD_2015-16-1.2.pdf"* distribuito per lo svolgimento della prova ed allegato alla presente.
Il progetto consiste nel risolvere un problema su grafi e nell'implementare un programma progettato mediante una combinazione degli strumenti, algoritmi e strutture dati, illustrati durante le lezioni del corso di ASD.
Si vuole, con le presenti pagine, esporre il problema di natura computazionale affrontato, indicare le specifiche seguite, descrivere la soluzione adottata con relativo studio della complessità teorica e valutazione di correttezza degli algoritmi proposti, precisare le condizioni sotto le quali è stato implementato il programma e sono state effettuate le misurazioni dei tempi, ed infine illustrare tutte le osservazioni necessarie.

Capitolo 2

Condizioni di sviluppo e misurazione dei tempi

Il programma è stato implementato in linguaggio *Java* versione *1.7* e sviluppato in ambiente *Linux (Ubuntu 14.04, 64-bit)* utilizzando l'ambiente di sviluppo *Eclipse Mars 2 (4.5.2)*.
I test sperimentali relativi alla misurazione dei tempi sono stati effettuati su un *Computer Desktop* con processore *Intel Core i5-750 Quad Core* e con sistema operativo *Ubuntu 14.04 (64-bit)* su macchina virtuale.

Capitolo 3

Guida alla compilazione

I seguenti comandi per la compilazione si riferiscono al terminale di Linux. Per eseguire la compilazione del programma vanno effettuate le seguenti operazioni da terminale:

(1) Estrarre il contenuto dell'archivio e portarsi nella directory `"Collevati_116286/progettoSolver/"`;

(2) Eseguire il comando `"javac *.java"`;

(3) Per lanciare il programma reindirizzando l'input da file eseguire il comando `"java Main < /path/file"`;

Per la compilazione del programma per la misurazione dei tempi la procedura è analoga, con l'unica differenza che al punto (1) bisogna portarsi nella directory `"Collevati_116286/progettoTempi/"`. Quest'ultimo programma non richiede alcun file in input, ma genera su *standard output (stdout)* le informazioni relative ai test sperimentali, mentre genera su *standard error (stderr)* le informazioni relative all'attuale stato di computazione. Per redirigere su file solamente i dati significativi per la misurazione dei tempi, eseguire il seguente comando: `"java Main 1> /path/file"`, in tal modo lo *standard output* viene scritto su file, mentre lo *standard error* viene stampato su terminale.

Capitolo 4

Problema e Specifiche

4.1 Problema

Definizione 1. *Dato un grafo orientato G = (V,E) diciamo che* ammette una radice *se e solo se esiste un nodo $r \in V$ tale che ogni $v \in V$ sia raggiungibile da r in G.*

Problema 1. *Dato in input un grafo orientato G,*

(1) *si determini il numero minimo di archi da aggiungere a G in modo tale da ottenere G' = (V,E') che ammetta una radice r;*

(2) *si produca un albero $T = (V,E_T)$ dei cammini minimi in G' di radice r.*

Note:

- **non** è ammesso aggiungere nodi ma solo archi al grafo G;
- **non** è detto che G' sia diverso da G, potrebbe anche essere uguale, poichè non è stato necessario aggiungere alcun arco a G dato che ammette già una radice r;
- **non** è detto che il grafo G sia connesso, potrebbe non esserlo;
- alla fine si deve ottenere un nodo di G' (la radice r) da cui raggiungere tutti gli altri.

4.2 Specifiche

L'input del programma sarà dato in formato dot e l'output dovrà essere fornito nello stesso linguaggio.
L'output dovrà essere un grafo $G' = (V,E')$ che rappresenterà contemporaneamente sia G' che T secondo le seguenti direttive:

(1) G' deve ammettere una radice r etichettata con la parola *root* e con il numero naturale $|E'| - |E|$ (che deve essere il sia minimo possibile);

(2) gli archi in $E' \setminus E$ devono essere colorati di rosso;

(3) T deve essere un albero di radice r;

(4) ogni nodo di T deve essere etichettato con la sua distanza dalla radice;

(5) ogni arco di T deve essere tratteggiato.

Capitolo 5

Soluzione del problema

In questo capitolo verrà descritta la soluzione adottata per il problema presentando inizialmente l'idea generale e successivamente andando ad esporre ed analizzare lo pseudo-codice del programma, per poi studiarne la complessità teorica e valutarne la correttezza.
Per la rappresentazione del grafo verranno utilizzate le liste di adiacenza, pertanto l'analisi della complessità teorica sarà effettuata tenendo in considerazione tale scelta.
Si ricordi durante la discussione della soluzione che il grafo G in input è orientato, tale informazione verrà sottintesa nel seguito.

L'idea generale per la soluzione del punto (1) del problema è la seguente:

(1) Calcolare le *componenti fortemente connesse* (che nel seguito si indicheranno con l'acronimo inglese *SCC*) del grafo G in input;

(2) Trovare tutti gli archi che attraversano *SCC* diverse;

(3) Creare un nuovo grafo G' che ha come nodi i rappresentanti delle *SCC* del grafo G e come archi tutti quelli trovati al punto (2), aggiunti a G' in modo tale da collegare i rappresentanti delle rispettive *SCC*;

(4) Prendere come radice r di G un qualsiasi nodo di G' che non abbia archi entranti (ovvero si sceglie come radice r di G un rappresentante di una qualsiasi *SCC* che non sia raggiunta da nessun altra *SCC*; si noti che come rappresentante di una *SCC* si può prendere qualsiasi nodo appartenente a quella *SCC*);

(5) Aggiungere tanti archi a G quanti sono i nodi di G', diversi dalla radice r, che non hanno archi entranti (chiameremo questi nodi *E-Nodes*). I nuovi archi in G andranno dalla radice r ai rappresentanti delle *SCC* che corrispondono ai nodi *E-Nodes* di G'.

Per la soluzione del punto (2) del problema l'idea è la seguente:

(6) Eseguire una *BFS (Breadth-First Search)* in G partendo dalla radice r per produrre un albero T dei cammini minimi (non è detto che tale albero T sia l'unico possibile).

Dato il grafo G in input, il punto (1) della soluzione consiste nel calcolare le sue *SCC* tramite l'algoritmo di *Tarjan*, opportunamente modificato per svolgere contemporaneamente anche il punto (2) e il punto (3). Tale algoritmo è costituito da una procedura esterna, che chiameremo *"TarjanSCC"*, che si occupa di inizializzare il grafo G, inizializzare le strutture dati di supporto e lanciare per ogni nodo non ancora visitato la procedura ricorsiva interna, che chiameremo *"StrongConnect"*. Quest'ultima svolge la parte fondamentale dell'algoritmo eseguendo una sola *DFS (Depth-First Search)* del grafo G, trovando tutti i successori del nodo v attualmente visitato ed aggiungendo come nodi del nuovo grafo G', che sarà restituito in output dall'algoritmo, i rappresentanti

delle *SCC* calcolate.
Prima di proseguire è importante far notare che ad ogni nodo è associata una proprietà *color* che memorizza il colore assunto da esso durante l'esecuzione dell'algoritmo. Il colore può essere *WHITE* (nodo non visitato), *GREY* (nodo in visita) o *BLACK* (nodo visitato).
Per poter svolgere il punto (2) della soluzione è stato aggiunto un ulteriore controllo all'interno della procedura *"StrongConnect"*, il quale verifica, per ogni nodo w presente nella lista di adiacenza del nodo v attualmente visitato, se il suo colore è *BLACK* e in tal caso aggiunge l'arco *(v,w)* alla lista di archi chiamata *"egdes"*. Tale lista è necessaria per salvare tutti gli archi che attraversano *SCC* diverse, infatti dopo aver visitato tutti i nodi di *G*, *"edges"* conterrà gli archi che stiamo cercando. Si noti che nel caso in cui non ci siano archi tra *SCC* diverse, la lista sarà vuota. Successivamente gli archi contenuti in *"edges"* verranno aggiunti al grafo *G'* all'interno della procedura *"TarjanSCC"*, come ultima operazione eseguita prima di restituire *G'*.
Prima di passare alla spiegazione del prossimo punto, bisogna sapere che ad ogni nodo è assegnato un intero unico chiamato *"start"*, che indica il tempo di scoperta del nodo, ed anche un altro intero chiamato *"lowStart"*, che indica il tempo di inizio visita minimo tra quelli dei nodi raggiunti dal nodo. Per impostare questi due valori useremo un clock globale chiamato *"time"*, inizializzato al valore *0* ed incrementato di *1* ogni volta che viene visitato un nodo nuovo.
Per svolgere il punto (3) della soluzione è necessario, durante l'esecuzione della procedura *"StrongConnect"*, aggiungere a *G'* il nodo v attualmente visitato soltanto se le sue variabili *"start"* e *"lowStart"* sono uguali, perchè ciò significa che è stata trovata una nuova *SCC* e il nodo v è usato come il rappresentante. Tale operazione viene eseguita solo dopo aver visitato tutta la lista di adiacenza del nodo v, ed è l'ultima ad essere eseguita prima di terminare la *"StrongConnect"*.
Il grafo *G'* risultante dall'esecuzione dell'algoritmo di *Tarjan* esposto sopra è un *DAG (Directed Acyclic Graph)*, che ci offre la certezza di avere almeno una sorgente, cioè un nodo senza archi entranti, proprietà fondamentale per il corretto funzionamento dei successivi algoritmi. L'unica osservazione da fare sul grafo *G'* è che in alcuni casi potrebbe avere degli archi ripetuti. Questo si verifica quando nodi diversi in *G* appartenenti alla stessa *SCC* hanno un arco uscente verso un'altra uguale *SCC*, ma ciò non rappresenta un problema al fine del raggiungimento dell'obiettivo finale.
Il punto (4) e (5) vengono realizzati grazie a due algoritmi che chiameremo *"getRootAndMinEdges"* e *"addRootAndMinEdges"*. Il primo si occupa di analizzare tutti i nodi del grafo *G'*, che altro non sono che i rappresentanti delle *SCC* del grafo *G*, e scegliere come radice r per *G* il primo nodo senza archi entranti che viene incontrato (si potrebbe scegliere qualsiasi nodo di *G'* senza archi entranti, ma per semplicità si prende il primo incontrato). Una volta determinata la radice r di *G*, per ogni successivo nodo v senza archi entranti incontrato durante la scansione del grafo *G'*, viene aggiunto l'arco *(r,v)* alla lista di archi chiamata *"minimumEdges"*. Tale lista è necessaria per salvare tutti gli archi che costituiranno il minimo numero di archi da aggiungere a *G* in modo tale che ammetta r come radice. Si noti che nel caso in cui tutti i nodi di *G'*, eccetto la radice r, abbiano almeno un arco entrante, la lista sarà vuota.
L'algoritmo *"getRootAndMinEdges"* restituisce in output la coppia costituita

dalla radice r e dalla lista di archi *"minimumEdges"*. Tale coppia sarà l'input per *"addRootAndMinEdges"* che si occupa di impostare la radice r di G con quella trovata dal precedente algoritmo e di aggiungere tutti gli archi presenti nella lista al grafo G.
Con questi algoritmi si risolve il punto (1) del problema, cioè ottenere un nodo di G (la radice r) da cui raggiungo tutti gli altri.
Per risolvere il punto (2), basta eseguire una *BFS* in G partendo dalla radice r e marcando con *DASHED* tutti gli archi che si percorrono per raggiungere tutti i nodi. L'albero T dei cammini minimi in G di radice r sarà costituito da tutti i nodi e da tutti gli archi precedentemente marcati con *DASHED*.
Le strutture dati di supporto utilizzate nei precedenti algoritmi sono: la lista di archi, necessaria per *"TarjanSCC"*, *"getRootAndMinEdges"* e *"addRootAndMinEdges"*; lo stack, necessario per *"TarjanSCC"*; la coda FIFO, necessaria per la *BFS*.
Nelle pagine seguenti lo pseudo-codice degli algoritmi appena descritti, ad eccezione della *BFS* poichè uguale a quella vista a lezione.

Algorithm 1 Calcola le SCC del grafo G e crea il nuovo grafo G'

```
1: procedure TarjanSCC(Graph G = (V, E))
2:     for each v ∈ V do
3:         v.color ← WHITE
4:         v.start ← −1
5:         v.lowStart ← −1
6:     end for
7:     time ← 0
8:     G' ← new Graph                                  ▷ Graph G' = (V', E')
9:     S ← new Stack
10:    edges ← new List
11:    for each v ∈ V do
12:        if v.color = WHITE then
13:            time ← StrongConnect(G, G', v, time, S, edges)
14:        end if
15:    end for
16:    while edges ≠ ∅ do
17:        edge ← edges.head
18:        E'.addEdge(edge)
19:        edges ← edges.tail
20:    end while
21:    return G'
22: end procedure
23:
24: procedure StrongConnect(Graph G = (V, E), Graph G' = (V', E'),
    Vertex v, var time, Stack S, List edges)
25:    v.color ← GREY
26:    v.start ← time
27:    v.lowStart ← time
28:    time ← time + 1
29:    S.push(v)
30:    for each (v, w) ∈ E do
31:        if w.color = WHITE then
32:            time ← StrongConnect(G, G', w, time, S, edges)
33:            v.lowStart ← min(v.lowStart, w.lowStart)
34:        else if w.color = GREY then
35:            v.lowStart ← min(v.lowStart, w.start)
36:        end if
37:        if w.color = BLACK then
38:            add (v,w) in edges
39:        end if
40:    end for
41:    if v.lowStart = v.start then
42:        V'.addVertex(v)
43:        do
44:            w ← S.pop
45:            w.lowStart ← v.lowStart
46:            w.color ← BLACK
47:        while w ≠ v
48:    end if
49:    return time
50: end procedure
```

Algorithm 2 Ricerca radice r e minimo numero di archi da aggiungere a G

```
1: procedure getRootAndMinEdges(Graph G' = (V', E'))
2:     minimumEdges ← new List
3:     foundRoot ← FALSE
4:     root ← NIL
5:     for each v ∈ V' do        ▷ v.inAdj = lista adiacenza archi entranti in v
6:         if !foundRoot ∧ v.inAdj = ∅ then
7:             foundRoot ← TRUE
8:             root ← v
9:         else if v.inAdj = ∅ then
10:            add (r,v) in minimumEdges
11:        end if
12:    end for
13:    return (root, minimumEdges)
14: end procedure
```

Algorithm 3 Imposta radice r e aggiunge archi al grafo G

```
1: procedure addRootAndMinEdges(Graph G = (V, E), Vertex root,
   List minimumEdges)
2:     G.root ← root
3:     while minimumEdges ≠ ∅ do
4:         edge ← minimumEdges.head
5:         E.addEdge(edge)
6:         minimumEdges ← minimumEdges.tail
7:     end while
8: end procedure
```

5.1 Complessità

Si studierà ora la complessità teorica dell'algoritmo proposto per risolvere il problema.
Iniziamo considerando l'algoritmo di *Tarjan* che, come già accennato, è lievemente modificato rispetto a quello presente in letteratura. Le principali aggiunte che sono state introdotte sono due: la prima riguarda un controllo all'interno della procedura *"StrongConnect"*, il quale verifica, per ogni nodo w presente nella lista di adiacenza del nodo v attualmente visitato, se il suo colore è *BLACK* e in tal caso aggiunge l'arco *(v,w)* alla lista di archi *"egdes"*. Tale operazione ha costo costante *O(1)* e pertanto non influisce sulla complessità asintotica dell'algoritmo.
La seconda aggiunta si trova all'interno della procedura *"TarjanSCC"*, come ultima operazione da eseguire prima di restituire *G'*, e riguarda la scansione della lista *"edges"* per aggiungere tutti gli archi trovati al nuovo grafo *G'*. Tale operazione ha un costo pari a $O(|E|)$, poichè nel caso peggiore la lista può contenere al più tutti gli archi di *G*. Quindi, dato che prima di quest'ultima operazione la procedura ha la stessa complessità dell'algoritmo di *Tarjan*, che è lineare nel numero di archi e nodi in *G*, cioè $O(|V| + |E|)$, l'aggiunta di un costo pari a $O(|E|)$ non cambia la complessità asintotica dell'algoritmo, che in definitiva è di $O(|V| + |E|)$.

Successivamente viene eseguito l'algoritmo *"getRootAndMinEdges"*, che si occupa di scandire tutti i nodi del grafo *G'* alla ricerca della radice r e del minimo numero di archi da aggiungere a G in modo tale che ammetta r come radice. Dato che nel caso peggiore ogni nodo di G può essere una *SCC*, e quindi *G'* avrebbe lo stesso numero di nodi di G, la complessità asintotica di questo algoritmo è $O(|V|)$ con $V \in G$.
L'algoritmo seguente è *"addRootAndMinEdges"*, che si occupa di impostare la radice r e di aggiungere effettivamente al grafo G tutti gli archi trovati dalla procedura *"getRootAndMinEdges"*. La parte più costosa dell'algoritmo riguarda la scansione della lista *"minimumEdges"* per l'aggiunta degli archi. Tale lista nel caso peggiore può contenere al più un numero di archi uguale al numero di nodi di *G'* meno *1*. Pertanto il costo dell'algoritmo dipende dal numero di nodi di *G'*, che a sua volta dipende dal numero di *SCC* del grafo G, e quindi la complessità asintotica è $O(|V|)$ con $V \in G$.
La complessità dell'ultimo algoritmo eseguito, *BFS*, è nota essere $O(|V|+|E|)$, come si è visto a lezione. Volendo rendere il più accurata possibile l'analisi, si osserva che la *BFS* viene eseguita partendo dalla radice r, che per Definizione 1 sappiamo raggiungere tutti i nodi del grafo. Quindi l'esecuzione dell'algoritmo ricade sempre nel caso peggiore, cioè ogni nodo e ogni arco viene esplorato, per cui la complessità asintotica in realtà è $\Theta(|V| + |E|)$.
In conclusione, l'algoritmo che domina tra tutti per complessità è *BFS*, e quindi la soluzione risulta avere una complessità finale pari a $\Theta(|V| + |E|)$.

5.2 Correttezza

Per la valutazione di correttezza della soluzione proposta ci si sofferma a dimostrare solamente quelle parti degli algoritmi che costituiscono una variazione rispetto agli algoritmi presenti in letteratura.
Si dimostra la correttezza dell'aggiunta introdotta nell'algoritmo *"StrongConnect"*.

Teorema 1. Nell'algoritmo *"StrongConnect"*, se il nodo v attualmente visitato incontra, durante la visita della sua lista di adiacenza, un nodo w colorato di *BLACK* allora l'arco *(v,w)* collega due *SCC* diverse.

Dimostrazione. La dimostrazione segue dall'invariante dello stack dell'algoritmo di *Tarjan*. I nodi sono collocati sullo stack nell'ordine in cui sono visitati. Quando la ricerca in profondità visita ricorsivamente un nodo v e i suoi discendenti, quei nodi non sono tutti necessariamente prelevati dallo stack quando queste chiamate ricorsive terminano. La proprietà invariante cruciale è che un nodo rimane nello stack dopo che è stato visitato se e solo se esiste un percorso nel grafo da esso verso qualche nodo precedente nello stack. Alla fine della chiamata che visita il nodo v ed i suoi discendenti, sappiamo se v stesso ha un percorso verso qualsiasi nodo precedente nello stack. Se è così, la chiamata termina, lasciando v sullo stack per preservare l'invariante. Altrimenti, v deve essere il rappresentante della sua *SCC*, che consiste in v assieme con tutti gli altri nodi successivi a v nello stack (questi nodi hanno tutti percorsi che ritornano fino a v ma non a qualsiasi nodo precedente, perchè se avessero percorsi a nodi precedenti, allora anche v avrebbe percorsi a nodi precedenti, che è falso). La *SCC* rappresentata da v viene poi prelevata dallo stack colorando ogni suo

nodo di *BLACK*. Pertanto tali nodi non saranno più visitati, ed ogniqualvolta saranno raggiunti da un arco uscente da un altro nodo w vorrà dire che tale arco collega due *SCC* diverse. □

Si dimostra che l'algoritmo *"getRootAndMinEdges"* determina effettivamente il minimo numero di archi.

Teorema 2. L'algoritmo *"getRootAndMinEdges"* determina il numero minimo di archi da aggiungere al grafo G in modo tale che ammetta una radice r.

Dimostrazione. Sia *G'* il *DAG* dei rappresentanti delle *SCC* del grafo G, si suppone che abbia n nodi, di cui m (con $m \leq n$) non hanno archi entranti. Presa come radice r di *G'* uno qualsiasi di questi m nodi, il minimo numero di archi da aggiungere a *G'* in modo tale che ammetta r come radice è banalmente uguale a *m-1*, e corrispondono a tutti gli archi che vanno dalla radice r agli altri *m-1* nodi. Per tutti gli altri *n-m* nodi esiste già un percorso da uno degli m nodi per raggiungerli, e quindi l'albero finale è costituito dalla radice r che ha come figli gli *m-1* nodi, e gli altri *n-m* nodi sono a loro volta figli degli m nodi. Quindi tale albero di radice r è stato costruito aggiungendo il minimo numero di archi possibile, che saranno gli stessi archi da aggiungere al grafo G di input. □

Capitolo 6

Osservazioni finali

La misurazione dei tempi di esecuzione è stata effettuata su grafi orientati con un numero progressivo di nodi $\{500, 600, 700, \ldots, 2000\}$ e con un numero incrementale di archi tra essi. La cardinalità del campione generato si basa sull'algoritmo *9* degli appunti, richiedendo che l'intervallo di confidenza Δ sia minore di *1/100* del tempo medio misurato. Si è scelto di utilizzare un valore di Δ relativo, pari a una frazione del tempo medio misurato, piuttosto che assoluto. Questo perchè, per input molto grandi, avere un valore fisso per Δ può portare a fare troppe iterazioni senza guadagnare molta precisione. Infine si è fissato un coefficiente di confidenza pari a *95%*. Nella pagina seguente si trovano la Tabella 1 contenente i dati sperimentali ottenuti dalle misurazioni sull'algoritmo, e il grafico in Figura 1 che ne fornisce una loro visualizzazione e la funzione rappresentata.
Per concludere si osservi come l'andamento del grafico in Figura 1 sia di tipo lineare, confermando lo studio teorico fatto in Sezione 5.1 sulla complessità asintotica della soluzione, la quale risulta essere lineare nel numero di archi e nodi del grafo.

Tabella 1: Dati sperimentali

$\|V\|$	$\|E\|$	$\|V\| + \|E\|$	Time(ms)
500	52 665	$5.32{\cdot}10^4$	1
600	76 089	$7.67{\cdot}10^4$	2
700	103 309	$1.04{\cdot}10^5$	3
800	134 690	$1.35{\cdot}10^5$	5
900	171 224	$1.72{\cdot}10^5$	7
1 000	210 631	$2.12{\cdot}10^5$	9
1 100	255 043	$2.56{\cdot}10^5$	12
1 200	303 109	$3.04{\cdot}10^5$	15
1 300	356 951	$3.58{\cdot}10^5$	18
1 400	412 642	$4.14{\cdot}10^5$	21
1 500	475 051	$4.77{\cdot}10^5$	25
1 600	539 326	$5.41{\cdot}10^5$	30
1 700	610 073	$6.12{\cdot}10^5$	33
1 800	683 761	$6.86{\cdot}10^5$	37
1 900	761 291	$7.63{\cdot}10^5$	42
2 000	843 528	$8.46{\cdot}10^5$	47

Figura 1: Grafico dei dati sperimentali