Relazione Prova di Laboratorio 2015/2016

Algoritmi e Strutture Dati

Dante Qyshka

Matricola 124660

qyshka.dante@spes.uniud.it

16 Maggio 2017

Indice

CAPITOLO 1

Introduzione

Consegna

Definizione 1: Sia ∑ un alfabeto finito e ordinato(arbitrariamente) tale che il carattere "ь" appartenga a ∑ ("ь" è il blank o spazio vuoto). Dato T ∈ ∑*, definiamo parola in T una qualsiasi stringa ∈ $(\sum \setminus \{ ь \})^+$ (stringa non vuota di caratteri diversi da "ь").

Definizione 2: Data una parola w di un testo T ∈ ∑, indichiamo con ^w il vettore dei caratteri di w, cioè il vettore di lunghezza |∑| tale che, per ogni i ∈ {1, … |∑|},

$$^w[i] = |\{ j \mid w[j] \text{ è l'i-esimo carattere nell'ordine di} \sum \}|.$$

Diciamo che ^u > ^v se, per ogni i ∈ {1, …|∑|},

$$^u \neq ^v \ \wedge \ ^u[i] \geq ^v[i]$$

Dato un testo T ∈ ∑*, si $G_T = (V_T, E_T)$ il grafo orientato i cui nodi ed archi sono così definiti:

$$V_T = \{w \mid w \text{ è una parola di } T\},$$

$$E_T = \{ (u, v) \mid u, v \in V_1{}^T \wedge u \neq v \wedge ^u > ^v \}.$$

N.B. Parola uguali corrispondono ad un unico nodo.

Problema 1: *Dato un testo T si calcoli il cammino di massima lunghezza di un cammino tra due nodi in G_T.*

1.1 Definizioni Teoriche

Un grafo G è una struttura dati caratterizzata da una coppia ordinata G = (V, E) di insiemi in cui V è un insieme di nodi mentre E di archi.
In particolare, ogni elemento appartenente ad E è rappresentato da una coppia di elementi di V, quindi $E \subseteq V \times V$.
L'ordine di un grafo è |V| (il numero dei vertici). La dimensione di un grafo è |E|.
Si dice *cammino* in un grafo un percorso dato da una sequenza di nodi $v_i \in V$ a due a due distinti (ma non per forza non ripetuti) e di archi $(v_i, v_j) \in E$ che li collegano.
Un cammino in cui vengono ripetuti uno o più nodi è detto ciclo.
Un grafo può disporre di diverse proprietà:

- Orientato, se E è un insieme di coppie ordinate di nodi (u, v) con $u, v \in V$.
- Non Orientato, se E è un insieme di coppie non ordinate di nodi $\{u, v\}$ con $u, v \in V$.
- Aciclico, se non contiene cicli.
- Ciclico, se contiene cicli.

Le *liste di adiacenza* di un nodo N sono vettori L_G di una determinata lunghezza dove sono presenti elementi contenenti i nodi raggiungibili direttamente da N.

Un *cammino di lunghezza massima* è una sequenza di nodi, senza ripetizioni, misurabile dal numero di archi (o peso).

1.2 Specifiche

Il progetto è stato sviluppato in Java con l'IDE Intellij. I test di funzionamento e di calcolo sono stati effettuati su un Asus X53S.
Il notebook è dotato di un processore Intel(R) Core™ i7-2670QM CPU da 2.20GHz e di una RAM DDR3 da 6 GB e sistema operativo Windows 10 pro.

Le seguenti istruzioni sono state utilizzate sul prompt dei comandi dopo averlo aperto dalla cartella contenente le classi:

../Main_Algorithm/

```
javac Main.java
java Main < testfile.txt > outputfile.dot
```

../Analysis/

```
javac Time.java
java Time > timelist.txt
```

1.3 Tipologia Input

Il file di input viene utilizzato come previsto e richiesto dalla consegna del progetto, ovvero viene associato tramite standard input.
Il numero di spazi e righe è ininfluente dato che sono stati passati alla classe di lettura verranno eliminati in modo da isolare le parole componenti la stringa di testo.

Sono presenti delle restrizioni:

- Il file di testo deve contenere il nome del grafo seguito da "=" in modo da sapere quando inizia il testo interessato.
- Nel caso siano presenti caratteri come " e / si possono riscontrare problemi nella visione del contenuto del file di output.

L'input viene passato per una funzione di acquisizione che suddividerà la stringa in un array di stringhe (contenenti le parole) e dopo aver fatto i necessari controlli aggiungerà i nodi alla *node_list* del grafo.

Nel caso dell'analisi dei tempi non è presente un file di testo esterno ma vengono generati automaticamente.
Vengono accettati solo alcuni caratteri, i rimanenti vengono trasformati in spazi vuoti in modo da delimitare le parole.

CAPITOLO 2

Soluzione proposta

2.1 Problema

La consegna del progetto richiede la generazione di un grafo orientato in cui i nodi contengono una parola ottenuta da una stringa di testo e riconoscibile perché inclusa tra 2 caratteri rappresentanti spazi vuoti.

Bisognerà poi creare le strutture dati supportanti il progetto e sviluppare le funzioni per la creazione e inserimento dei nodi, controllo per vedere che tipo di relazione hanno le parole tra loro, popolamento delle liste di adiacenza ed infine individuare il cammino di lunghezza massima.

2.2 Algoritmo

L'idea portante della soluzione è quella di utilizzare delle euristiche durante il popolamento del grafo con nodi e archi in modo da ottenere le liste di adiacenze dei nodi in entrata e dei nodi in uscita in modo da facilitare lo scorrimento per i controlli lungo le liste e la ricerca del cammino di lunghezza massima.

L'algoritmo si divide in quattro fasi principali:

- Creazione e inserimento dei nodi che andranno a popolare il grafo e le liste di adiacenza ed ordinamento delle parole in ordine crescente tramite **Quicksort**;
- La seconda fase prevede l'esecuzione della funzione di connessione dei nodi e popolamento delle liste di adiacenze, include la terza fase;
- Nella terza fase viene eseguita la funziona per controllare le parole;
- Nella quarta fase viene eseguita una variante dell'ordinamento topologico in modo da poter eseguire scorrimenti del grafo e percorrere gli archi che non portano a nodi senza genitori, i quali sono gli obiettivi finali dello scorrimento. Qui verrà individuata la distanza massima tra 2 nodi connessi.

Per controllare se una parola contiene completamente un'altra vengono fatti dei controlli in cui si scorre per ogni lettera della parola più corta tutte le lettere della parola più lunga e qualora venga trovato un riscontro si elimina la lettera trovata in modo da permette l'identificazione di caratteri ripetuti.

Per individuare il cammino di lunghezza massima parto dai nodi agli estremi del grafo e procedo lungo la lista di adiacenza del nodo corrente escludendo però i nodi senza genitori.
In questo modo quando arriverà alla fine della sua iterazione avrà scartato tutti i percorsi più corti in ordine crescente (il primo scartato è il più corto).

2.3 Elaborazione dati

Anzitutto se non vengono ottenuti dei nodi da un testo nullo anche l'output sarà nullo.

I dati ottenuti vengono gestiti tramite strutture dati specifiche e sono:

- Node: Ogni istanza della classe Node, rappresenta un vertice del grafo con i relativi attributi necessari per il corretto funzionamento di esso e dell'algoritmo. Contengono delle liste di adiacenza per identificare i vertici vicini.
- Graph: Classe prettamente utilizzata per semplificare la manipolazione e gestione del grafo. Il suo costruttore è una dichiarazione di un ArrayList contenente i nodi memorizzati.

ALGORITMI

La classe principale contenente il main del progetto è Main.java; all'interno di essa sono richiamate tutte le funzioni necessarie per la risoluzione del problema posto.

1. **connectGraph(Graph g)** ha il compito di creare gli archi tra i nodi, popolare le liste di adiacenza dei figli e dei genitori e settare, quando necessario, i flag per controllare se un nodo è all'estremo o meno del grafo.
2. **test(String searchIn, String searchFor)** ha il compito di controllare se una parola è completamente contenuta in un'altra indipendentemente da ripetizioni o ordine dei caratteri. Per fare ciò esegue due cicli in cui controlla se i caratteri che compongono le stringhe sono uguali o meno.
   Ritorna un boolean.
3. **quickSort(String[] words, int floor, int ceil)** esegue un quicksort. È già stato presentato e dimostrato a lezione.
4. **longestPath(Graph g)** è una procedura impiegata per ottenere la distanza massima implementando una versione dell'ordinamento topologico (semplificato grazie alle euristiche adottate).
   Ritorna una lista.
5. **visit(Node n, ArrayList<Node> L)** è un metodo ausiliario a **longestPath** utilizzato per scorrere le liste di adiacenza dei genitori individuare il percorso più lungo tramite l'incremento della variabile rappresentante la distanza.
6. **String printGraph(Graph g, String graphName)** scorre prima i nodi del grafo e poi le loro liste di adiacenza, applicando rispettivamente delle funzioni per impostare la corretta formattazione dei file .dot.

7. **readGraph(Graph myGraph)** legge i dati ricevuti dallo standard input.

2.4 Output

L'output viene generato come una stringa con l'utilizzo della funzione printGraph e, dopo essere stata richiamata nel Main, stampa sul file di output la stringa ritornata. È necessario usare il comando di indirizzamento dello standard output
> [nome_file_output].dot

Per l'analisi dei tempi viene solo indirizzato vero un file di testo il tempo ottenuto per ogni iterazione del ciclo.

CAPITOLO 3

Analisi computazionale

3.1 Complessità

Analisi della complessità computazionale delle procedure:

- **ReadWrite(Graph g)** Ha complessità $O(n^2)$ dove n è il numero di parole. La complessità di questa funzione risiede nell'inserimento dei nodi nel grafo e nel controllo che non ci siano ripetizioni.
  Per avere la sicurezza che non ci siano ripetizioni di nodi **[verifyNode(Node nodeCheck)]** viene utilizzata una funzione che scorre la lista del grafo e controlla se esiste un nodo con la stessa Label. Tecnicamente questa funzione ha complessità $\Theta(n^2)$ ma, applicando prima il Quicksort, eventuali ripetizioni saranno sequenziali e quindi individuate immediatamente.
- printGraph(Graph g, String graphName) Ha complessità $\Theta(|V+E|)$.
  Questa funzione visita tutti i nodi e tutti gli archi una sola volta in modo da poterli aggiungere alla stringa che verrà stampata nel file di output.
- **test(String p1, String p2)** Ha complessità $O(|p1 \times p2|)$. Questa funzione esegue un controllo sulle parole (sono ordinate tra loro) e scorre i caratteri della parola minore(p2) e per ognuno di questi controlla se è presente nella parola di dimensione maggiore(p1). Viene utilizzato il metodo indexOf () per individuare la posizione della lettera ricercata in p1.
- **connectGraph(Graph g)** Ha complessità $O(|V|^2)$.
  La funzione si occupa di popolare le liste di adiacenza di ogni nodo e per questo motivo deve scorrere tutti i nodi per ogni parola a trovata in modo da poter fare il test di contenimento tra le due stringhe individuate.

- **quickSort(String[] words, int floor, int ceil)** Ha complessità O(n*logn) dove n è il numero di parole.
  È una quicksort già vista e dimostrata a lezione.
- **longestPath(Graph g)** Ha complessità Θ(|V+E|).
  Questa funzione prima scorre i nodi per individuare quelli identificabili come foglie (ovvero senza figli) e li aggiunge ad una lista ausiliaria; questo mi permette di ridurre il numero di iterazione da fare.
  Viene poi eseguito un ciclo di visita dei nodi in modo da identificare il percorso massimo partendo però dalle foglie.
  Non essendoci l'uso dei colori nei nodi la complessità della funzione combinata alla visita è Θ(|S|*|V+E|) dove S è la cardinalità delle foglie, mentre V e E sono i nodi e archi visitati per arrivare ai nodi senza genitori("radici").

3.1 Complessità totale e Correttezza

Nel calcolo della complessità asintotica totale le operazioni di lettura dell'input non hanno un costo abbastanza influente da essere significativo nell'analisi della complessità teorica.

La complessità finale dell'algoritmo è O(|V|*|V+E|) rendendola così quadratica.

La correttezza dell'algoritmo si basa sulla correttezza delle funzioni prese individualmente.
Le funzioni per la risoluzione del problema sono varianti di algoritmi già visti a lezione come il Quicksort e l'ordinamento Topologico. Per questo motivo la funzione **longestPath** terminerà sempre dato che il grafo non possiede nodi all'indietro e di conseguenza non ha cicli all'interno, permettendo così di poter passare tutti i percorsi senza la preoccupazione di dover controllare il colore per evitare di eseguire visite già eseguite prima. Inoltre durante **visit** l'algoritmo esegue una visita che scarta gli archi vicini che portano direttamente ai nodi finali, garantendo così che verrà scelto sempre il percorso più lungo a condizione che siano presenti dei nodi di partenza e dei nodi di arrivo già definiti.

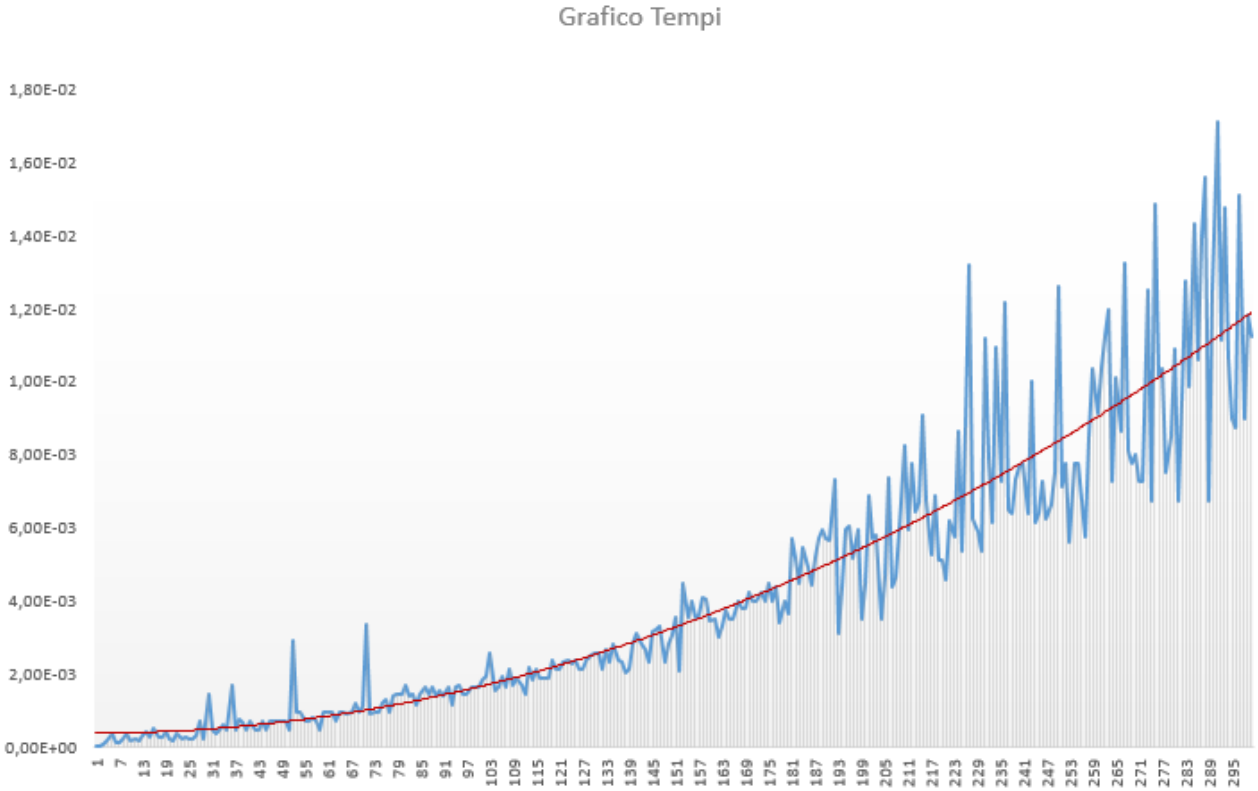
È quindi possibile affermare che l'algoritmo è corretto nella sua completezza e soddisfa le richieste fatte nella consegna del progetto.

3.1 Rappresentazione dei tempi

Lo studio e rappresentazione dei tempi è stato svolto implementando gli algoritmi presentati all'interno del file con gli appunti forniti facendo uso delle capacità del computer in dotazione.

Per svolgere l'analisi bisogna individuare le casistiche ottime, medie e peggiori dell'algoritmo. In questo caso l'unica variabile adottata per modificare l'algoritmo è stato il numero di nodi dato che il numero di archi non era gestibile.
Il caso migliore identificato è quello in cui è presente solo un nodo, in questo modo si ottiene direttamente la distanza massima, essendo questa invariata da quella iniziale. Il caso peggiore invece è identificabile come un grafo dotato di |V| nodi ed il nodo con la label di lunghezza massima è collegato a tutti i nodi con label di lunghezza inferiore e questi a loro volta sono collegati a tutti i nodi inferiori. In questo modo vengono generate liste di adiacenze con dimensione massima ad ogni "livello".

Per recuperare i dati relativi ai tempi ho generato grafi con dimensioni comprese tra 1 e 300. La generazione del testo invece è stata eseguita sfruttando una variante dell'algoritmo **RandomNumberGenerator** fornito in cui vengono generati casualmente dei caratteri (lo spazio vuoto ha una probabilità pari a 32/150) e generà caratteri fino ad ottenere 300 parole.



Nel grafico riportato è possibile vedere come il tempo di esecuzione (linea blu) tende ad una curva con esponente quadratico (linea rossa) con delle variazioni

dovute al numero di archi presenti a cause delle relazioni delle parole formanti il grafo. Conferma quindi che la complessità sia O(|V|*|V+E|).

CAPITOLO 4

Conclusione

4.1 Difficoltà

Durante lo sviluppo del progetto sono state riscontrate diverse difficoltà nell'implementazione del codice e nella fase di analisi dei tempi.

Inizialmente la funzione test doveva avere complessità lineare grazie alla conversione dei caratteri nel loro valore decimale che sarebbe stato memorizzato in un array di dimensione pari a quella dei caratteri senza ripetizioni con all'interno del singolo elemento un indice che ne indicasse le ripetizioni. Questo sarebbe poi stato passato ad un array con dimensione pari a quella dell'alfabeto e sottratto ai valori della seconda parola. Così facendo sia avrebbe avuto una velocità pari alla dimensione delle due parole. Ciò però non è stato possibile a causa dell'impossibilità di convertire i caratteri speciali in valori decimali ordinati.

Nella fase di analisi dei tempi invece sono state riscontrate difficoltà nella generazione dei tempi dato che in alcuni casi risultava impossibile generare tutti i tempi quando i nodi erano più di 3000. Altre volte il prompt dei comandi non generava alcun valore ma ciò succedeva solo in rari casi indipendenti dal tipo di grafo generato.

4.1 Osservazioni

Con l'analisi dei grafici presentati nella pagina precedente è possibile appurare che la correttezza individuata sia valida e di come il valore ottenuto nel caso migliore (un solo nodo) sia inferiore al medio e di conseguenza peggiore.

Ritengo quindi che l'algoritmo prodotto ed implementato sia buono poichè risolve il problema posto. È possibile però implementare migliorie per incrementarne l'efficienza.

Bibliografia

[1]Alberto policriti, per la stima dei tempi del progetto 2016-17 , Pseudo Random Numbers Generators

http://users.dimi.uniud.it/~alberto.policriti/home/?q=node/3

[2] Cormen T.H., Leiserson C.E., Rivest R.L, Stein C, Introduction to Algorithms, MIT Press, Third edition, 2009

[3] Appunti personali presi alle lezioni di Algoritmi e Strutture dati