Federico Marcolongo 866862

# Progetto di Algoritmi – Dizionario (Giugno 2018)

**Strutture dati**

La traccia del progetto richiede di costruire un grafo non orientato a partire da una lista di parole data in input da file.

Il primo aspetto riguarda l'individuazione delle strutture dati necessarie (in questo caso principalmente grafi e array) e la scelta dell'implementazione più comoda rispetto alle richieste specifiche del problema. Tra queste, spicca più delle altre la necessità di determinare i diametri delle componenti connesse del grafo. Per facilitare questa operazione, ho scelto di implementare la struttura di grafo tramite lista di adiacenza. L'implementazione con lista archi infatti è particolarmente scomoda da utilizzare con algoritmi di visita (per ogni nodo è necessario scorrere l'intera lista degli archi) e l'implementazione con matrice di adiacenza si rivela comportare un'occupazione di memoria eccessiva rispetto alla sparsità dei grafi forniti come esempio, da me presi come riferimento per considerazioni a spanne sul caso medio nello studio della complessità spaziale e temporale degli algoritmi usati (nonostante il campione sia molto ridotto, tutti gli esempi, anche in lingue diverse, tendono ad avere molte somiglianze, come la presenza di una componente connessa molto più grande di tutte le altre formata perlopiù da parole molto brevi).

Nota: nel corso di tutto il progetto, il primo elemento di una lista (lista di componenti connesse, di nodi, di adiacenza) ha indice 0 e non 1.

Tutte le strutture che ho introdotto possiedono un costruttore e un distruttore. Per evitare di perdere l'indirizzo di aree di memoria allocate dinamicamente, mi sono attenuto a un criterio First-In-Last-Out, in cui ogni struttura allocata, al momento di essere rimossa dalla memoria, chiama per prima cosa i distruttori delle strutture che ad essa si appoggiano.

La struttura Graph consta di una lista (implementata come tabella) di elementi di tipo Node, di variabili intere per registrare le dimensioni del grafo (numero di nodi, di lati e di componenti connesse) e di un puntatore a un elemento di tipo Info, appendice del grafo che contiene le informazioni relative alle richieste specificate nella traccia del progetto.
La lista di elementi Node è implementata in modo perfettamente analogo all'implementazione della struttura List, implementata come tabella di interi.

La struttura Node è formata da tre elementi: un campo di tipo stringa (char[]) allocata staticamente con lunghezza MAX_STR_LEN = 30 (da traccia) + 1 (per il carattere '\0'), uno di tipo int che registra la componente connessa a cui il nodo appartiene (oltre ad essere utilizzato come marker sia per la Breadth-First-Search per le componenti connesse, sia per la BFS nel calcolo dell'eccentricità di ogni nodo), e un campo di tipo List (tabella di interi) che registra la lista di adiacenza del nodo come lista degli indici con cui i nodi adiacenti compaiono nel grafo (nella sua lista di nodi).
Il grado di un nodo è memorizzato nel campo "active" della lista di adiacenza, infatti esso corrisponde proprio alla lunghezza di tale lista.

La struttura List non è altro che una lista di elementi di tipo int implementata come tabella, ovvero come tripla di puntatore a int per il primo indirizzo di memoria, numero di elementi allocati e numero di elementi utilizzati (attivi). Ho usato questo tipo di struttura per gestire elenchi di numeri di cui non si conosce a priori la lunghezza, mentre per elenchi di lunghezza nota (come per esempio l'elenco dei gradi dei nodi) mi sono limitato all'utilizzo di array nativi di C, non essendo necessario aggiungere elementi oltre la lunghezza già allocata. Oltre a un semplice metodo di lettura dell'elemento i-esimo, ho implementato un metodo di accodamento di elementi alla lista che, nel momento in cui la lista stessa fosse piena (n° di elementi attivi =: M == N := n° di elementi allocati), alloca 2N celle di memoria, copia nelle prime N celle il contenuto della lista (elemento per elemento) e infine libera la memoria di partenza. Questo tipo di implementazione garantisce che la memoria allocata sia sempre per più di metà utilizzata (complessità spaziale lineare rispetto al numero M di elementi nella lista) e che, nel caso peggiore in cui una lista parta da vuota, il processo di riallocazione venga eseguito ceil(log2(M)) volte (complessità temporale logaritmica rispetto a M).
L'accodamento di nodi nella lista presente nel tipo Graph è gestita esattamente allo stesso modo.

La struttura Info è stata implementata per poter lasciare una certa flessibilità alla struttura di grafo: la struttura Graph descritta finora, trascurando il puntatore a Info, contiene già tutte le informazioni strutturali necessarie a descrivere un grafo (nodi e archi, e in questo caso anche componenti connesse) e può essere utilizzata così com'è per rappresentare un qualunque grafo non orientato. La struttura Info interviene declinando la struttura più generale di grafo alle necessità specifiche del progetto, ovvero offrendo un compendio, o un'appendice, delle caratteristiche informative deducibili dalle caratteristiche strutturali dell'oggetto madre, come la lista dei diametri delle componenti connesse, la lista dei gradi dei nodi ordinata per frequenza assoluta e così via. Questo accorgimento è a metà strada tra l'implementare tutte queste variabili direttamente nella struttura Graph, rendendola per certi versi eccessivamente pesante, e tenerle all'intero del main, fuori dalla definizione della struttura, sacrificando una buona dose di modularità del codice.
La struttura Info può essere costruita solo da un grafo attraverso la funzione addInfo. Questa legge direttamente le dimensioni del grafo e alloca i vettori membro di conseguenza, senza necessità di liste implementate come List.

Le strutture Graph e Info contengono, inoltre, variabili flag utilizzate da vari metodi interni. Qui e per il resto della relazione intendo, con metodo ("di una struttura"), una funzione che abbia come primo argomento un puntatore a un'istanza della struttura stessa, in perfetta analogia con l'implementazione di classi in linguaggi Object-Oriented.

Nota: nell'appendice Info compaiono anche i vettori di interi sortedNodes e sortedComponents. Questi sono implementazioni di dizionari (int : int) che ho utilizzato come ghost array nell'ordinamento sia delle componenti connesse (con l'ordine descritto nella traccia), sia delle parole in ordine alfabetico: l'algoritmo di ordinamento confronta i valori ma agisce (swap di elementi in-place) sul vettore delle chiavi. Questo ad esempio alleggerisce lo swap di char[31], riducendolo a uno swap di int.

## Algoritmi

La traccia del progetto richiede di compiere una serie di operazioni su una struttura di grafo, prima per popolarlo, e poi per determinarne caratteristiche rilevanti. Questa serie di operazioni conduce a una suddivisione naturale del problema in sottoproblemi.
L'ordine di questi sottoproblemi rispecchia l'ordine con cui sono stati affrontati nel codice e non quello delle richieste della traccia di esame: per alcuni di essi è indifferente l'ordine con cui vengono eseguiti e in ogni caso le loro soluzioni vengono salvate in memoria per poi essere stampate su file nell'ordine e nel formato richiesti dalla traccia.

Nota: a meno di specificare altrimenti, userò le lettere n, e, c rispettivamente per il numero di nodi (di parole), il numero di lati (di legami tra le parole) e il numero di componenti connesse del grafo in esame. Inoltre userò le stesse lettere con pedice i (per esempio $n_i$) per denotare le stesse quantità relative all'i-esima componente connessa.

### 1. Nodi, lati e componenti connesse

#### 1.1 Popolamento del grafo

Il programma riceve in input, come primo argomento, il nome di un file sorgente contenente una parola per riga. Il metodo importGraph riceve questo nome come parametro e compie tre operazioni:

- Apre uno stream di lettura con il file sorgente controllando che esso esista (O(1))
- Aggiunge una parola per volta al grafo e chiude lo stream (O(n): il tempo di riallocazione per la lista di nodi è logaritmico e diventa asintoticamente ininfluente)
- Determina l'esistenza di archi tra due suoi nodi tramite la funzione di confronto areRelated (in totale la procedura compie n(n-1)/2 confronti, ognuno dei quali richiede tempo costante, essendo la lunghezza massima delle parole fissata: $O(n^2)$)

Il numero di nodi e il numero di lati vengono automaticamente memorizzati nelle relative variabili.

Osservo due cose: i gradi dei nodi sono già memorizzati nella variabile "active" di ogni loro lista di adiacenza. Inoltre il tempo di riallocazione delle liste di adiacenza è trascurabile perché nel caso peggiore di grafo completo ogni nodo si trova ad avere n-1 elementi nella lista di adiacenza in tempo ceil(log2(n-1)), per un tempo complessivo (somma su n nodi) O(n*logn). Tempo trascurabile rispetto a $O(n^2)$.

#### 1.2 Determinazione delle componenti connesse

Per determinare le componenti connesse opero una visita in ampiezza (BFS) con la variabile cComponent, membro di Node, come marker inizializzato a -1 ("non marcato"). Il resto è implementato come una normale BFS con coda dei nodi non ancora visitati.
Questo algoritmo controlla, per ogni nodo raggiunto, se ciascuno dei nodi nella sua lista di adiacenza è già stato marcato. Questo controllo è eseguito in tempo costante.

Per ogni componente connessa, l'algoritmo controlla $e_i$ lati, dove $e_i$ è il numero di lati nella componente i-esima. Sommando sulle componenti connesse, si ottiene un numero di controlli pari a e, il numero di lati del grafo. Nel caso di grafo completo, e = n(n-1)/2.

La BFS richiede quindi, nel caso peggiore, un tempo quadratico nel numero di nodi.
In tutti gli esempi forniti, tuttavia, il grafo è molto più sparso e il numero di lati sembra essere dello stesso ordine del numero dei nodi. Questo comporta che la complessità della BFS, nel nostro caso, sia meglio rappresentata da un andamento lineare o leggermente sovralineare, osservazione che sarà rilevante nella sezione relativa al calcolo dei diametri.
La visita in ampiezza lavora sul grafo già presente in memoria introducendo solo la coda di visita, di dimensione n. L'utilizzo di memoria rimane quindi lineare.

## 2. Grandezze rilevanti

Questa sezione è tutta contenuta nella chiamata del metodo getInfo, preceduto dal metodo addInfo che alloca dinamicamente la memoria necessaria, con le dimensioni prese automaticamente dalle dimensioni dell'oggetto Graph dato come parametro, a contenere le informazioni rilevanti per la soluzione del progetto: il grado di ogni nodo, il grado massimo tra i nodi del grafo, la frequenza (assoluta) di ciascun grado, la cardinalità delle componenti connesse, l'eccentricità di ogni nodo, il diametro e la prima e l'ultima parola di ciascuna componente.
Tutti i vettori citati in questa sezione sono membri della struttura Info e occupano una quantità di memoria lineare in n o in c (quindi sicuramente O(n)).

### 2.1 Gradi dei nodi, cardinalità e prima e ultima parola delle componenti connesse

Per prima cosa popolo il vettore degrees con il grado di ciascun nodo, nello stesso ordine con cui compare nel grafo. Per far ciò è sufficiente iterare sui nodi e leggere il membro active della lista di adiacenza. Nel corso di questa lettura tengo traccia del grado massimo, che mi servirà per allocare (+1 per gli elementi di grado 0) il vettore degreesCount, che conta il numero di nodi aventi un determinato grado.

Per il vettore cardinalities mi basta iterare sui nodi e, per ogni nodo, aumentare di uno il contatore relativo alla componente connessa a cui esso appartiene.
Applico lo stesso procedimento al vettore degreesCount iterando sul vettore degrees, ottenendo all'i-esima posizione il numero di nodi aventi grado i. Per costruzione il vettore è ordinato per gradi crescenti.

Infine, per i vettori firstWord e secondWord, conduco una semplice ricerca di massimo/minimo (con funzione di confronto strcmp) lineare, iterando sui nodi del grafo, confrontando l'i-esima parola con il valore attualmente registrato nel vettore xxxxWord alla posizione corrispondente alla componente connessa relativa all'i-esimo nodo.

Tutte queste operazioni richiedono un tempo O(n), relativo all'iterazione sui nodi.

### 2.2 Calcolo dell'eccentricità di un nodo

Per poter determinare il diametro delle componenti connesse, calcolo, per ogni nodo, la sua massima distanza da un altro nodo nella stessa componente connessa. Questa distanza è chiamata eccentricità. Il diametro della componente sarà quindi il massimo sulla componente connessa delle eccentricità dei suoi nodi.
La visita in ampiezza si presta molto bene al calcolo dell'eccentricità. Infatti, dato un nodo sorgente, a ogni altro nodo a lui connesso può essere associato induttivamente un intero che ne indica la profondità e che fa da marcatore della BFS, in questo modo: un nodo non ancora marcato ha profondità -1, il nodo sorgente ha profondità 0, e ogni nodo (in coda) che ne raggiunge uno non ancora marcato gli associa una profondità pari alla propria + 1.

La correttezza della procedura è garantita dall'algoritmo di visita: essendo i lati del grafo non pesati, ogni nodo viene visitato dalla BFS, a partire dal nodo sorgente, lungo il più breve cammino tra esso e la sorgente stessa. Questo permette di riconoscere nella profondità di un determinato nodo la distanza di questo dalla sorgente e di memorizzare la massima profondità registrata dall'algoritmo di visita come eccentricità del nodo sorgente.

La complessità temporale di questa procedura segue lo stesso ragionamento della visita in ampiezza operata per la determinazione delle componenti connesse. L'unico accorgimento è che la visita, questa volta, è estesa a una sola componente, per cui il numero di controlli è $e_i$. Continuano a valere il tempo quadratico (in questo caso in $n_i$) nel caso peggiore (componente completamente connessa) e il ragionamento sulla sparsità dei grafi dati come esempio.

2.3 Calcolo del diametro delle componenti connesse

Per calcolare il diametro di ogni componente connessa è sufficiente, arrivati a questo punto, calcolare l'eccentricità di ciascun nodo presente nel grafo e determinarne, componente per componente, il massimo (con una ricerca lineare per confronto diretto).

La complessità della ricerca del massimo è trascurabile rispetto a quella del calcolo delle eccentricità. Il calcolo dei diametri richiede un tempo $T = \sum e_{c(j)}$, dove la somma è estesa a tutti i nodi del grafo, mentre $e_{c(j)}$ indica il numero di lati della componente contenente l'i-esimo nodo. Osservando che $e_{c(j)}$ è costante sulla componente (è lo stesso per ogni suo nodo), posso raccogliere un po' di termini e riscrivere $T = \sum(n_i \cdot e_i)$ , dove questa volta la sommatoria è sulle componenti).

Nel caso peggiore di componenti completamente connesse, $e_{c(j)} = n_{c(j)} \cdot (n_{c(j)}-1)/2$, per cui $2 \cdot T = \sum n_i^3 - \sum n_i^2$, che è definitivamente crescente nella norma $l_3$ del vettore $(n_i)$. Il dominio della funzione è dato da $D = \{(n_i) \text{ in } R^c \mid \sum n_i = n,\ n_i \geq 0,\ 0 \leq i < c\}$ (iperpiano di $R^c$ perpendicolare alla diagonale). Su questo iperpiano, la norma massima di $(n_i)$ è realizzata sulle intersezioni con gli assi coordinati. Il caso peggiore è quindi realizzato per $n_i = n$, $n_j = 0$ per $i \neq j$, ovvero nel caso di grafo completo. In questo caso, riprendendo l'espressione di T, l'algoritmo richiede un tempo $O(n^3)$.
Con un ragionamento analogo nel caso $e_i \approx n_i$, in questo caso con la norma $l_2$, si ottiene una stima per $T = O(n^2)$. Come osservato sopra, questo caso ($e \approx n$) sembra rappresentare meglio il tipo di dati fornito come esempio.

**3. Ordinamento**

L'ultimo problema da affrontare rimane l'ordinamento delle componenti connesse rispetto alla relazione d'ordine specificata nella traccia e delle parole date in input in ordine alfabetico.
L'ordinamento delle parole è un problema che si presterebbe in modo perfettamente naturale all'implementazione di un algoritmo di tipo bucketsort, che opera in tempo lineare nel numero di parole. Questo algoritmo tuttavia è estremamente sensibile alla modifica dell'alfabeto di riferimento come l'introduzione di ulteriori caratteri rispetto a quelli nei file su cui viene testato. Inoltre, cosa ancora più rilevante per questo progetto, il tempo richiesto dal calcolo dei diametri è estremamente limitante e mette in secondo piano la differenza tra tempi O(n) e O(n·logn), tipici questi ultimi di altri tipi di algoritmi di ordinamento.

Per questo motivo ho preferito affrontare parallelamente i due problemi di ordinamento, implementando dapprima, come template, un algoritmo di tipo quicksort-in-place per vettori di interi, per poi riadattarlo, agendo solo sulle funzioni di confronto, ai casi delle componenti connesse e dei nodi. Per poter utilizzare l'algoritmo template ho introdotto due vettori di interi, sortedComponents e sortedNodes, di dimensione rispettivamente c e n, che ho utilizzato come dizionari tra l'ordine risolvente il problema e l'ordine con cui i nodi e le componenti sono stati aggiunti al grafo. Per esempio, a problema risolto, il vettore sortedNodes avrà in posizione 0 l'indice del nodo (nella lista di nodi del grafo) corrispondente alla prima parola in ordine alfabetico presente nel file, in posizione 2 l'indice corrispondente alla seconda e così via. Questo tipo di approccio ha inoltre il vantaggio di non compiere scambi di oggetti Node ma solo di elementi di tipo intero.
Si potrebbe perfino implementare un algoritmo di ordinamento generico che ammetta la funzione di confronto come argomento, eliminando il problema di adattare l'implementazione caso per caso.

Il quicksort da me implementato agisce per ricorsione (appoggiandosi al record di attivazione) e in-place, in modo da avere un utilizzo di memoria lineare nella dimensione del vettore da ordinare. Sia essa m per non creare confusione con il numero di nodi, denotato con n.
Inoltre l'algoritmo seleziona sempre come pivot il primo elemento del vettore dato in ingresso.
Il tempo richiesto da quicksort è O(m·logm) nel caso medio, apparentemente ben rappresentato dai file campione. Il caso peggiore si ha prendendo l'elemento di valore minimo o massimo come pivot e richiede un tempo $O(n^2)$. Nell'eventualità in cui i file sorgente fossero parzialmente ordinati, basterebbe implementare una riga di codice in cima alla definizione della funzione quicksort che selezioni un elemento di indice casuale, tra quelli validi, come pivot.

**Appendice: funzioni di confronto**

Per affrontare i problemi di popolamento del grafo e di ordinamento ho utilizzato la funzione strcmp (libreria <string.h>) e implementato le funzioni areRelated e comesBefore.
La funzione strcmp ammette due stringhe come parametri e restituisce un valore intero (negativo se la prima viene prima della seconda in ordine alfabetico, nullo se le stringhe coincidono e positivo nel caso opposto al primo).

Nota: l'ordine alfabetico indotto dalla funzione strcmp è in realtà la relazione d'ordine tra gli interi rappresentanti i caratteri nella codifica ASCII, per questo motivo i caratteri accentati compaiono in posizioni esterne all'intervallo a-z.

**1. areRelated:** determina se due parole differiscono per una lettera

Prototipo: Bool areRelated(const char *firstWord, const char *secondWord)

Il tipo Bool è implementato come unsigned char, FALSE e TRUE sono definiti risp. come 0 e 1
La funzione areRelated agisce su due livelli:

1. Restituisce FALSE se la lunghezza $l_1$, $l_2$ delle due stringhe differisce per più di due caratteri, altrimenti separa il controllo in tre casi: $l_1 - l_2 = -1, 0, 1$.

2. Caso 0: scorre entrambe le stringhe carattere per carattere e restituisce FALSE appena trova una seconda anomalia.

   Caso -1, 1: individua la posizione del primo carattere anomalo e, quando lo trova, salta avanti di un carattere nella stringa più lunga. Dopodiché riprende il confronto e restituisce FALSE appena trova una seconda anomalia.

Restituisce TRUE altrimenti.

**2. comesBefore:** determina se una componente connessa viene prima di un'altra nell'ordine custom (cardinalità – diametro – prima parola)

Prototipo: int comesBefore(Graph *graph, unsigned int c1, unsigned int c2)

Prende in input l'indice di due componenti connesse all'interno del grafo graph e restituisce un valore intero con la stessa convenzione tenuta in strcmp, di cui ricalca in parte l'implementazione.

Perché ciò avvenga, la funzione lavora sulla differenza di due valori, partendo dalle cardinalità delle due componenti.

- Se la differenza è nulla, i due valori coincidono e la funzione passa al livello successivo (cardinalità -> diametro, diametro -> prima parola).

- Se invece non è nulla restituisce il valore della differenza dei valori con segno, senza passare al livello successivo.

Nel confronto tra le prime parole delle due componenti connesse la funzione comesBefore chiama direttamente strcmp.

La funzione comesBefore non può restituire 0: perché ciò accada è necessario che due componenti connesse condividano la loro prima parola. Una parola tuttavia non può appartenere contemporaneamente a due diverse componenti connesse.