Relazione Laboratorio

Leonardo Necordi

**Struttura Scelta:**

Variante dei grafi con liste di adiacenza.

Dopo aver riflettuto sulle strutture possibili tra quelle viste a lezione per implementare il DataBase, quella dei grafi con liste di adiacenza mi è parsa la più azzeccata per la sua dinamicità e libertà nell’implementazione.

Ho valutato anche la versione con matrice di adiacenza ma nonostante avrebbe forse reso più efficiente qualche operazione, avrebbe peccato in quanto a dinamicità dovendo ogni volta riallocare la matrice.

Pensando al modo migliore per implementare questo data base, mi è immediatamente giunta l’intuizione di dividere i due tipi di Entità: Film e Attori in modo da poter accedere più rapidamente all’entità desiderata dovendo scorrere una porzione ridotta dell’insieme.

Dividere i due tipi di entità ha fatto nascere una necessità: disporre di una cella **Database** di partenza da cui ci si potesse muovere distintamente tramite puntatori verso la lista dei film o delle persone, cella che ho deciso di sfruttare al meglio conservando in essa anche alcune informazioni sulla base di dati stessa quali il numero di Entità e Relazioni divise per tipologia; conservare queste informazioni complica di un’ unica operazione le funzioni di inserimento ma permette alle due interrogazioni numeriche di avere complessità costante.

Per l’implementazione delle due strutture Vertex ed Edge mi sono attenuto a quella standard con poche variazioni, la prima tra tutte il nome, in modo da renderlo più intuitivo per ciò che stiamo rappresentando: Entity e Relationship i nomi delle strutture, Nodo ed Arco i puntatori ad esse.

Ho inoltre rinominato il resto delle funzioni e tipi sempre per accordarli alla struttura DataBase.

Per le relazioni ho deciso di mantenere come label la stringa contenente il nome dell’entità adiacente, evitando i puntatori che complicano l’implementazione e risulterebbero utili unicamente per il Numero di Bacon, questa funzione è stata l’unica penalizzata dalle mie scelte, la sua complessità concettuale mi ha infatti impedito di esaminarla accuratamente in fase di decisione della struttura, ma ne parleremo più nel dettaglio più avanti.

Proprio per questa funzione però, la struct Entità ottiene il campo “bacon\_number”, una variante della variabile booleana “visited”, che oltre a dirci se è un’entità non è ancora stata visitata (-1), ci fornisce informazioni sulla distanza più breve fra l’Entità e Kevin Bacon.

La struttura ausiliaria Lista invece è una lista collegata semplicemente, con inserimento e rimozione in testa, diversa da una stack per le funzioni isPresent e removeDuplicates.

Alcune note sulla notazione per le complessità:

Quando ne vengono indicate due, la prima è per il caso migliore, la seconda per il caso peggiore.  
Indico con f il numero dei film del database, con p il numero di persone, con (f o p) il numero di film o di persone, concorde con l’elemento preso in considerazione, con a il numero delle relazioni tipo “ACTED”.

Con r il numero di relazioni di un’entità, con R il numero di relazioni del database, con E il numero delle Entità del database.

**Funzioni:**

Liste:

Indico con n il numero di elementi della lista

Clear

**Scopo:** Svuota la lista

**Complessità:** Theta(n)

**Descrizione:** Arriva ricorsivamente alla fine della lista, da lì elimina ogni elemento tramite backtracking

addFront, createEmpty, isEmpty, pop

Implementazione standard, complessità costante.

printList, isPresent

Implementazione standard, complessità Theta(n).

removeDuplicates

**Scopo:** Rimuove i duplicati dalla lista

**Complessità:** Theta(n2)

**Descrizione:** Crea una nuova lista, scorre quella precedente chiamando ogni volta isPresent per vedere se l’elemento è già stato inserito in quella nuova, in caso contrario lo inserisce in essa; alla fine pulisce quella vecchia e ritorna quella nuova.

Data Base:

createEmpty

**Scopo:** Crea un DataBase vuoto

**Complessità:** Theta(1)

**Descrizione:** Alloca la cella centrale del DataBase, impostando a zero tutte le variabili informative.

*Aggiunta Entità:*

addEntity

**Scopo:** Gestisce la cella DataBase per l’aggiunta dell’entità

**Complessità:** Theta(1) Theta(f o p) [Dipende da recursiveAddEntity]

**Descrizione:** Chiama recurisveAddEntity sulla lista delle Entità del tipo di quella da inserire, se l’inserimento ha avuto successo ritorna true e incrementa di uno il numero di Entità di quel tipo.

recurisveAddEntity

**Scopo:** Aggiunge l’entità alla lista corretta

**Complessità:** Theta (1) [Esiste già elemento in prima posizione] Theta(f o p) [Elemento non esiste e viene quindi inserito]

**Descrizione:** Scorre ricorsivamente la lista di tipo t fino al fondo, se non ha trovato l’elemento lo inserisce in fondo.

*Aggiunta Relazione:*

addRelationship

**Scopo:** Gestisce la cella DataBase per l’aggiunta della relazione.

**Complessità:** Theta(1) [Trovo subito dove aggiungere e la relazione esiste già in prima posizione] Theta(E + r) [Le entità si trovano nelle ultime posizioni]

**Descrizione:** Chiama verticalAddRelationship sui film e sulle persone, se entrambe ritornano true aggiorna il numero delle relazioni di quel tipo e ritorna true a sua volta.

verticalAddRelationship

**Scopo:** Scorre le entità per l’aggiunta della relazione.

**Complessità:** Theta(1) [Trovo subito dove aggiungere e la relazione esiste già in prima posizione] Theta(f o p + r) [L’entità si trova nell’ultima posizione]

**Descrizione:** Scorre ricorsivamente le entità del tipo t, se trova quella in cui deve inserire chiama horizontalAddRelationship e ritorna true, ritorna false altrimenti.

horizontalAddRelationship

**Scopo:** Inserisce la relazione in coda se non esiste già

**Complessità:** Theta(1) [Trovo la relazione in prima posizione] Theta(r) [Non trovo la relazione e la aggiungo]

**Descrizione:** Scorre ricorsivamente le relazioni dell’Entità presa in considerazione, se non la trova la aggiunge in coda.

*Interrogazioni Numeriche:*

numEntitiesPerType, numRelationshipsPerType

**Scopo:** Ritorna il numero di entità o relazioni di un certo tipo

**Complessità:** Theta(1)

**Descrizione:** Accedo ai campi numerici della struct DataBase e ne ritorno il contenuto.

*Stampe:*

printDB

**Scopo:** Gestisce la cella Database e imposta la stampa.

**Complessità:** Theta(E + R)

**Descrizione:** La stampa di tutto il Database ritorna prima le informazioni numeriche con i totali delle Entità e delle Relazioni divise per tipo, poi chiama entitiesPrintDB per stampare le entità.

entitiesPrintDB

**Scopo:** Stampa le entità

**Complessità:** Theta(f o p + R)

**Descrizione:** Stampa ricorsivamente tutte le entità di un tipo, chiamando poi relationshipsPrintDB su ognuna di esse.

relationshipsPrintDB

**Scopo:** Stampa le relazioni di un’entità

**Complessità:** Theta(r)

**Descrizione:** Stampa ricorsivamente tutte le relazioni di un’entità, affiancandola dal numero corrispondente al tipo (0 ACTED, 1 DIRECTED, 2 PRODUCED)

*Query:*

coActors, actorsProducedBy, actorsDirectedBy

**Scopo:** Gestisce lista e individua il tipo di Query.

**Complessità:** Theta(1) [Non ci sono entità del tipo richiesto] Theta(f + R + n2)) dove n è la lunghezza della lista di output prima della rimozione dei duplicati [Viene iterato su tutti i film, auxQuery chiamata su ogni film finisce con l’iterare su tutte le Relazioni e removeDuplicates ha complessità n2, quindi dipendendo da variabili diverse queste complessità si sommano]

**Descrizione:** Inizializza la lista e chiama auxQuery ricorsivamente sul tipo di relazione ricercato, inizializza e resetta ogni iterazione la variabile “found”, che permette alle auxQuery su un determinato film se è stata trovata l’Entità cercata e di non aggiungere gli attori colleghi in caso contrario.

Infine rimuove dalla lista i duplicati e la ritorna.

auxQuery

**Scopo:** Se nel film indagato ha recitato l’entità richiesta, aggiunge i colleghi alla lista

**Complessità:** Theta(1) [Non ci sono entità del tipo richiesto] Theta(r)

**Descrizione:** Scorre ricorsivamente le relazioni di un film, aggiungendo ogni volta la persona in questione alla lista di output, se trova l’entità richiesta lo segnala alle altre tramite la variabile found (che essendo passata per riferimento, è visibile da tutte le funzioni in tempo reale); alla fine, se non ha trovato l’entità fra quelle in relazione con il film, elimina tramite backtracking tutte persone erroneamente aggiunte.

*Numero di Bacon:*

baconNumber

**Scopo:** Resettare le variabili bacon\_number, trovare Kevin Bacon, ritornare il percorso più breve

**Complessità:** Caso migliore: Theta(1) [Cerco Kevin Bacon] Caso Peggiore: Theta(r(di Kevin Bacon) \* f \* r(di ogni film) \* (complessità auxBaconNumber)) [Trascuro sia la complessità di clearBacon che del primo ciclo, entrambe decisamente dominate dal resto, che corrisponde all’elevatissima complessità di auxBaconNumber calcolata in seguito, moltiplicata per la lista di adiacenza di Kevin Bacon, moltiplicata per tutti i film, moltiplicata per la lista di adiacenza di ogni film]

**Descrizione:** Innanzitutto, chiamo clearBacon per ripulire le variabili bacon\_number nel caso non fosse il primo calcolo di numero di Bacon, successivamente trovo Kevin Bacon tramite un ciclo, in modo da poter partire da 0 e impostare i valori dei cerchi di adiacenza intermedia in ordine crescente.

Tramite una serie di cicli innestati (il punto cruciale penalizzato dalla mia implementazione della struttura: non dispongo di un puntatore al nodo adiacente e pertanto necessito di un ciclo per trovarlo) chiamo auxBaconNumber su tutti gli attori colleghi di Bacon in qualche film, ognuna di queste chiamate mi restituirà la lunghezza del percorso trovato, fra di essere conservo unicamente quella minore (per fare ciò uso la classica tecnica del valore minimo inizializzato ad un valore talmente alto da essere irraggiungibile).

Finite le chiamate e i cicli, potrò quindi ritornare il numero che ho conservato.

clearBacon

**Scopo:** Resettare le variabili bacon\_number

**Complessità:** Theta(p)

**Descrizione:** Scorre iterativamente le persone del database, impostando ogni volta bacon\_number a NOT\_VISITED (-1), ovvero il valore che ci indica che non abbiamo ancora visitato tale entità.

Ricordo che bacon\_number serve sia per capire se un’entità è già stata visitata, sia quanto sia lungo il percorso migliore trovato fino a quel momento passando per l’entità.

auxBaconNumber

**Scopo:** Esplorare tutti i percorsi tra un dato attore e quelli che hanno recitato con lui, cercando l’attore l.

**Complessità:** Caso Peggiore: Theta(r(dell’attore) \* r(di ogni film) \* f \* a) [Il primo ciclo per trovare l’attore è di complessità p e si aggiungerebbe al resto, divenendo quindi trascurabile, la serie di cicli innestati itera su tutti i film in cui l’attore ha recitato, moltiplicato per tutti i film (per trovarlo ogni volta, non avendo il puntatore ad esso), moltiplicato per tutti gli attori che hanno recitato nel dato film, la funzione si chiama poi ricorsivamente su tutti gli attori (per qualche costante moltiplicativa che corrisponde a quando si passa più volte per un nodo avendo trovato un percorso più breve)].

È decisamente una complessità alta ed è l’unica veramente penalizzata dalle mie scelte strutturali.

**Descrizione:** Innanzitutto verifica che la funzione chiamante non abbia trovato l, in tal caso ritorna il livello di vicinanza raggiunto con questo percorso.

Altrimenti si porta sul nodo in questione, se il miglior percorso trovato passando per quella posizione è migliore di quello raggiungibile tramite quello corrente (quindi se bacon\_number è diverso da -1 e minore o uguale del livello corrente) ritorna NO\_BACON\_NUMBER in modo da ignorare questo percorso quando si valuterà quale sia il minimo nella funzione chiamante.

In caso contrario imposta il bacon\_number del nodo come il livello corrente e si chiama ricorsivamente su tutti gli attori che hanno lavorato con quello del nodo corrente, aumentando il livello passato per parametro di uno e conservando poi (come nella funzione principale) solo il risultato minore.

**Riguardo allo stile:**

Durante la stesura del codice ho cercato di favorire il principio di modularità, spezzando i problemi in diverse funzioni che svolgono un singolo task.

In particolare, ho spesso affidato alla funzione principale la gestione della cella contenente le informazioni generali del DataBase e a funzioni ausiliarie la gestione delle Entità o delle Relazioni.

Nei cicli capitava spesso di iterare fino all’ultimo elemento di una lista, al fine di rendere il codice leggibile sarebbe buona pratica utilizzare una funzione isEmpty o un tipo emptyList o simili, ma trattando diversi tipi di puntatore ho ritenuto confusionaria la creazione di diverse funzioni o tipo con il medesimo scopo, di conseguenza ho lasciato in diversi punti while(puntatore), nonostante la mancata eleganza di tale scrittura.

Comando per compilare:

g++ \*.cpp -Wall -o nome\_eseguibile (naturalmente, si può omettere sia il -Wall che il -o a seconda delle preferenze)