Report

Sistema di analisi relazioni umane e lavorative

Antonio Calagna S234496
Alessandro Scisca S233924
Clara Salvalai S235438

Politecnico di Torino - Ingegneria Elettronica A.S. 2017/2018

Introduzione

- Il programma è fondato sull'univocità dell'ID, stringa alfanumerica che identifica univocamente un determinato account a prescindere dalla sua tipologia.
- Le tre possibili tipologie di Account sono organizzate in una struttura gerarchica. Account è la classe madre da cui discendono le tre derivate "User", "Company" e "Group".
- Le notizie che gli Account hanno la facoltà di pubblicare sono definite attraverso la classe Post.
- La memorizzazione e la gestione delle relazioni che intercorrono tra i vari account è affidata alla classe Graph.
- Al fine di implementare nel main la sola interfaccia con l'utente esterno, l'esecuzione dei comandi è affidata alla classe Shell mentre l'elaborazione dei dati interni è affidata alla classe Manager.
- Controlli, gestione degli errori, acquisizione e redazione dei Files sono affidati alle classi FileHandler, Social_Handlers e IOBuffer.
- Il passaggio dei file tramite linea di comando è ottimizzato in modo da essere indipendente dal sistema operativo in esecuzione, tramite #ifdef _WIN32 ed #elif __APPLE__ e, a seconda del suddetto, verrà adottato di volta in volta l'opportuno percorso relativo per accedere ai file.
- Ad ogni avvio i file vengono sottoposti a riorganizzazione in modo da ottimizzare il salvataggio run-time delle modifiche sui files.

Gerarchia degli Accounts

Le tre classi “User”, “Company” e “Group” erediteranno da Account i seguenti dati:

- Nome e ID (string)
- Tipo di Account (char)
- Data di iscrizione alla rete (oggetto di classe Date)

Ogni tipologia di account presenterà infine dati e metodi privati specifici di quella tipologia.
I principali metodi:

- Overloading degli operatori di uguaglianza e disuguaglianza, fondati esclusivamente sugli ID (identità e confronto in lunghezza)
- Membri di natura static:
  - **IDValid**: atta a verificare la validità di un ID in termini di dimensioni e caratteri speciali: la stringa deve essere di tipologia alfanumerica con dimensione compresa fra i 3 e i 30 caratteri non speciali, tranne al più “_”.
  - **typeValid**: atta a verificare la validità di un carattere destinato alla tipologia di account. Essa controlla che il carattere sia uno dei tre static sopra citati.
  - **nameValid**: atta a verificare la validità di un nome in termini di caratteri permessi: la stringa deve essere di natura alfanumerica consentendo al più i caratteri speciali “ “(spazio), “ ‘ (apostrofo) ” e “-“.
  - *user_type*, *company_type* e *group_type*

Al fine di gestire in maniera ottimale relazioni la classe Account è corredata del namespace “Relation” contenente:

- Stringhe di natura “const” indicanti il tipo di relazione: amicizia, conoscenza, paternità, relazione affettiva, partnership tra aziende, assunzione lavorativa e appartenenza ad un gruppo.
- La funzione **isValid**: atta a verificare che una generica stringa, destinata alle relazioni, appartenga effettivamente alla suddetta lista di relazioni possibili.
- Le funzioni **isSymmetrical**, **isDominant** e **getInverse**: atte ad analizzare la natura della relazione e ottimizzare i processi di acquisizione e di redazione da e su files.

In un’ottica non binaria circa alla identità di genere degli utenti semplici, la classe User è corredata di un namespace “Gender” contenente:

- Un array di caratteri indicati, ciascuno, un determinato genere:
  - **M**: Male, **F**: Female, **N**: Non-Binary, **A**: Agender, **B**: Bigender, **O**: Other
- La funzione **isValid**: atta a verificare l’appartenenza al namespace di un determinato carattere destinato al genere.

Post

Una singola notizia comprende i seguenti dati:

- Una stringa contenente il testo della notizia
- Data e ora di pubblicazione, oggetti delle classi Date e Clock
- Due set di stringhe contenenti, rispettivamente, gli ID degli account che hanno interagito con tale notizia fornendo un like o un dislike.

La scelta dei **set<string>** come container è legata a questioni di ottimizzazione, non essendoci necessaria l’indicizzazione del container e dovendo spesso ricercare all’interno dello stesso.
I metodi implementati prevedono:

- Aggiunta, ricerca, rimozione e statistiche di interazioni
- Overloading degli operatori: uguaglianza (notizia, data e ora) e disuguaglianza (numero di likes).

Il singolo Post è dunque svincolato dall’appartenenza ad un certo utente, sarà il Manager ad occuparsene.

Graph

La classe Graph, generalizzata a due parametri template <**node_T**, **edge_T**> , implementa le funzionalità di un grafo direzionato.
I nodi vengono rappresentati in una mappa, container della STL, la cui chiave è il valore del nodo e il cui elemento mappato è un'ulteriore mappa che rappresenta gli archi uscenti dal nodo stesso. Gli archi, difatti, sono rappresentati come mappe che legano un pointer al nodo di arrivo (per evitare di duplicare le informazioni sul nodo di arrivo, che viene già rappresentato attraverso la chiave della prima mappa) al valore dell'arco (di tipo edge_T).
I nodi sono dunque costanti, assumendo sempre il ruolo di parametri per le funzioni, ed eventuali sostituzioni vengono implementate con opportuni algoritmi.
L'operatore [] permette l'accesso posizionale ad i nodi, ritornando una const reference al nodo nella posizione richiesta
I metodi prevedono:

- Calcolo del numero di nodi, numero di archi, grado di un nodo (in entrata, in uscita) e numero di archi uscenti con un determinato peso
- Prelievo di un arco tra due nodi, reference ad un nodo in una certa posizione, vettore di nodi, complessivamente o accomunati da una certo arco (**branches**)
- Aggiunta, modifica, rimozione di un nodo
- **setEdge/bsetEdge**: Aggiunta di un arco pesato mono/bi direzionale
- La rimozione di un arco è effettuata imponendo come peso "**no_edge**" fornito come membro static
- Overloading degli operatori: assegnazione e "[]" per l'accesso alla posizione di un certo nodo.

Main & Shell

L'interazione tra utente e programma avviene attraverso un'interfaccia in stile console basata su uno shell.
Il comando viene legato alla propria definizione attraverso una mappa stringa - pointer a funzione, utile a ricercarlo, interpretarlo e a richiamare, se esiste, la funzione ad esso associata.
Ogni comando, come da standard, segue la forma: <nome_comando> [<lista_dei_parametri>…].
La scelta dell'utilizzo dei pointer a funzioni implica che tutti i comandi siano definiti in funzioni dalla stessa struttura: ritorno void e con lo stesso numero di parametri. In particolare tutte le funzioni hanno accesso via reference al comando utente (contenuto in uno stringstream), al Manager e a due IOBuffer utili per la raccolta delle nuove informazioni e dei dati da eliminare.
Una lista di comandi e delle loro relative spiegazioni è disponibile richiamando il comando "help".
Sono anche presenti delle funzionalità di salvataggio e autosalvataggio, ambedue controllabili tramite appositi comandi. Grazie all'utilizzo degli IOBuffer è stato possibile non sovrascrivere i file finché non esplicitamente richiesto dall'utente tramite il comando "save" oppure, mantenendo attivo l'autosalvataggio, finché non vengono incontrate certe condizioni (ovvero l'esecuzione di un certo numero di comandi oppure la modifica di molte informazioni), il tutto per conferire una maggiore fluidità al programma durante l'esecuzione.
Le fasi svolte dal main sono:

- Controlli sui files, tramite le funzioni **checkFile** del FileHandler
- Acquisizione delle informazioni dai files, tramite le funzioni **fetchData** del FileHandler
- Riorganizzazione dei files in modo da ottimizzare le fasi di esecuzione e salvataggio delle modifiche
- Esecuzione ciclica di comandi, forniti dall'utente da tastiera, richiamando opportunamente le relative funzioni della libreria Shell.

Alla chiamata del comando "exit" il programma salva i dati e l'esecuzione si conclude.

Manager

La classe Manager si occupa di gestire e interfacciare gli oggetti definiti dalle classi precedenti. Essa comprende:

- Tre **unordered_map** aventi per chiave una stringa (ID) e per dato una determinata tipologia di Account e una **unordered_map** avente per chiave una stringa (ID) e per dato un vettore di Post, relativo, dunque, ad un determinato Account.
- Un oggetto di classe **Graph** in cui nodi e archi, di natura template, vengono fissati a stringhe. I nodi saranno gli ID degli Accounts e gli archi le relazioni tra gli stessi.

La natura associativa delle mappe e la loro efficiente complessità, costante, al più logaritmica, consentono una gestione efficace degli algoritmi sopratutto circa alla ricerca ed eliminazione delle chiavi, operazioni ricorrenti frequentemente nel corso del programma.
I metodi privati prevedono:

- Inizializzazione dei nodi del grafo e delle chiavi delle mappe
- Controllo di esistenza di un certo ID, facendo riferimento ai nodi del grafo
- Controllo sull'età tra due utenti semplici al fine di ottimizzare i controlli sulle relazioni di parentela

I metodi pubblici prevedono:

- Setters e getters dei dati privati
- Gestione degli Accounts:
  - Prelievo di un determinato Account a partire dal suo ID, di tutti gli account in un unico vettore binary-sorted o, ancora, di una precisa coppia ID — vettore di Post
  - Aggiunta, rimozione e rimpiazzo dei vari tipi di Accounts alle/dalle mappe per polimorfismo
- Gestione delle relazioni:
  - Aggiunta e rimozione di una relazione tra due ID, facendo capo al namespace delle relazioni situato in Account.h e gestendo opportunamente la simmetria/bidirezionalità delle stesse.
  - Prelievo di un vettore di ID legati ad un certo nodo di partenza tramite una determinata relazione
- Gestione dei post:
  - Aggiunta e rimozione di un determinato Post associato ad un certo utente
  - Aggiunta e rimozione di una interazione al post da parte di un ID (like/dislike) in forma polimorfica fornendo: Post, coppia ID(autore)-Post, ID e posizione nel vettore dei post
- Funzioni di natura statistica:
  - Numero di Account
  - Numero di amici di un certo utente semplice
  - Numero di parenti di certo utente semplice
  - Numero di impiegati di una certa azienda
  - Numero di consociate di una certa azienda
  - Numero di membri di un gruppo
  - Numero di utenti semplici nati dopo una certa data
  - Azienda con il maggior numero di impiegati
  - Partnership tra aziende con il maggior numero di impiegati
  - Utente semplice con più amici
  - Utente semplice con più conoscenze
  - Età media degli utenti semplici
  - Ritorno della coppia ID—Post (**pair<string, Post>**) per il post con più likes o più dislikes o ancora con il miglior/peggior rapporto di interazioni
  - ID dell'utente che ha ricevuto più interazioni (likes/dislikes) o dell'utente con il miglior rapporto di interazioni

- **LonerPeople**: Ricerca dei lupi solitari il cui principio è fondato su parametri forniti dall'utente. Se un determinato parametro è fissato a 0, esso non sarà rilevate ai fini della determinazione dei lupi solitari. La funzione ritorna un **unordered_set** di ID conformi alle istruzioni fornite.

- **Alberi genealogici**:

La creazione dell'albero genealogico di un utente avviene tramite un algoritmo simile alla breadth-search.
L'ID di un utente viene passato alla funzione **GenealogicalTree**, che si occupa della ricerca dei suoi familiari e del loro ordinamento in una struttura tabulare le cui righe rappresentano le generazioni e le cui colonne rappresentano i membri di queste, come da esempio:

	Membro 0	Membro 1	Membro 2
Generazione 0	familiare(0, 0)	*non tutte le generazioni sono ugualmente popolate*	
Generazione 1	familiare(1, 0)	familiare(1, 1)	familiare(1, 2)

Per generazione 0 si intende la generazione del membro più anziano della famiglia.
Le strutture scelte per la fase di ricerca sono le seguenti:
- Una mappa rappresenta la tabella delle generazioni.
- Una deque tiene traccia dei nodi in attesa di essere analizzati.
- Un set tiene conto dei nodi già analizzati.

Non essendo possibile conoscere in anticipo il numero di generazioni e la generazione di appartenenza dell'utente di partenza, a questo viene assegnata la generazione 0 e la tabella viene sviluppata in positivo e in negativo.
Viene tenuto conto anche della generazione di appartenenza dei nodi in attesa, calcolata in base alla relazione tramite la quale sono stati incontrati per la prima volta: se un utente viene messo in lista dal proprio padre, apparterrà alla generazione successiva, e viceversa.
La fase di ricerca consiste in un ciclo che si ripete fino allo svuotamento della deque, nel quale vengono svolte le seguenti operazioni:
- Estrazione dell'utente e della sua generazione dalla deque
- Ricerca di genitori e figli dell'utente
- Controllo che i parenti ottenuti non siano già presenti in lista e che non siano già stati analizzati
- Inserimento dei parenti validi in lista
- Inserimento dell'utente in analisi nella mappa delle generazioni

Al termine della fase di ricerca la mappa delle generazioni viene convertita in un vector<vector<string>> contenente, ad ogni indice, tutti gli ID concorrenti allo stesso livello generazionale e viene ritornato come risultato della funzione.
La funzione **FormatTree** si occuperà di manipolare l'albero restituito dalla GenealogicalTree arricchendolo e formattandolo nella maniera opportuna, evidenziando eventuali partner e il predecessore da cui discendono. Essa restituirà un singolo vettore di stringhe avente, per ogni indice, la descrizione di una precisa generazione.
Infine **PrintTree** e **PrintAllTrees** si occuperanno di formattare ulteriormente l'albero, stavolta, ai fini della stampa, a schermo o su file.

IOBuffer:

La classe IOBuffer fornisce oggetti la cui funzione è relativa alle operazioni di input/output.
Gli operatori << e >> vengono dotati della possibilità di interagire con utenti, gruppi, compagnie, relazioni e post.
Alcuni tipi vengono sin da subito ridefiniti in forma di std::pair al fine di accoppiare opportunamente le informazioni concomitanti, ad esempio, l'ID dell'account proprietario di un post e la struttura relazionale del grafo.
I dati vengono infine raccolti in una serie di deque al fine di mantenere l'ordine di inserimento quando viene richiesta l'estrazione di un dato e agevolare le operazioni.

FileHandler & Social_Handlers

La classe FileHandler fornisce una serie di funzioni generiche atte a interagire con un singolo file. Ad un oggetto FileHandler viene associato un singolo file, gestito da un dato privato di tipo fstream. Lo scambio di dati tra la classe e le librerie che ne fanno uso viene mediato da oggetti di tipo IOBuffer che raccolgono i dati da accodare oppure da eliminare dai file.La generalità delle suddette funzioni viene inoltre ristretta, tramite overloading, all'abito trattato complessivamente dal programma.
La classe è dotata di tre metodi virtuali, che devono essere implementati in apposite classi derivate affinché l'interazione sia possibile. Tali metodi sono dedicati a:

- Controllare la correttezza formale [1] di una *singola* riga del file, ignorando eventualmente righe vuote e righe commento, indicate con "//"
- Acquisire [2] i dati da una *singola* riga letta dal file e caricarli su un buffer
- Scaricare [3] il contenuto di un buffer nel file

FH::Error è il principale tipo di ritorno, definito all'interno della libreria stessa.
Esso una struttura composta da:

- Un codice errore, la cui tabella di interpretazione è redatta opportunamente e fornita nella documentazione annessa al programma.
- Un'ulteriore informazione numerica relativa all'ambito di utilizzo, ad esempio, la riga nella quale l'algoritmo avesse eventualmente riscontrato un problema.

Le classi **AccountsHandler**, **RelationsHandler** e **PostsHandler** specializzano la classe FileHandler e ne implementano le funzioni virtuali.

1. Le righe vengono analizzate una per volta. Per questo motivo non sono possibili controlli di coerenza dei dati, che vengono comunque eseguiti dal Manager durante la fase di acquisizione. Delle righe viene controllata la sola correttezza formale e logica tramite il richiamo di funzioni fornite dalle librerie che definiscono il dato sotto controllo.

2. Durante la fase di acquisizione non vengono svolti controlli: viene assunto che la fase di controllo sia stata correttamente eseguita e che abbia avuto un esito positivo. In tal caso è possibile leggere i dati dal file (che potrebbero ancora contenere errori di coerenza).

3. Durante la fase di stampa non vengono effettuati controlli sui dati forniti, ma vengono formattati i dati che possono generare errori (come il messaggio di un post, in cui vengono inseriti caratteri di controllo che differenziano i caratteri di formato dal contenuto di un campo del dato).