CPSC 260 Project: Social Network Analyzer

Names: John Henderson and David Wong
Student Numbers: 28132108 and 44627107
Date: November 26th 2013

1. Introduction

The purpose of this project is to utilize Object-Oriented Programming to create a Social Network and a Social Network Analyzer. This Social Network particularly focuses on the interaction between users of the Social Network with a calendar of events. The Social Network allows the creation of new users and deletion of existing users. As well, the Social Network allows the creation, deletion, and participation of events. A user of the Social Network can query the Social Network for various information, such as popular events happening in the Social Network, mutual friends between other users and distance between users.

2. Features

The following lists the features available in the Social Network:

- Making a “friend” connection between two users
- Adding a user to an event
- Organizing events into categories such as: Sports, music, festival, school, business, party, holiday
- Querying the shortest path of mutual friends between two users
- Querying the mutual friends between two users
- Querying the percentage of genders within the Social Network and particular events in the Social Network
- Querying a particular range of age of users within the Social Network and particular events in the Social Network

3. Rationale for the implementations

Rationale for General Design Decisions:

To fully incorporate Object-Oriented Programming to the design of our project, we utilized existing classes that were created from taking the course. The decision to go forward with an event-based Social Network was due to the already implemented Appointment, Calendar, and Date class from Lab 6. By using these existing classes and methods, we are able to meet a core concept of abstraction in Object-Oriented Programming. At the lowest level, the abstraction of events happening within a Social Network is nearly identical to abstraction of storing appointments for multiple people in a calendar.

In general, data in the social network is stored in vectors, consisting of pointers to objects. Vectors were used in the Social Network because vectors is an easy data structure to implement dynamically. Vectors also provide easy access to elements when compared to a data structure such as a linked list. Access is very important in social network; typically a social network will access user data more often than add new users.
The pointers to objects are used to minimize the amount of memory used as objects are stored in multiple areas in the Social Network. For example, references to a unique user are stored in the social network, the user's friends' friends lists, and the participants list of the events that the user is attending.

Rationale for Implementation of Key Features and Queries:

Shortest Distance between 2 Friends:
The algorithm used to calculate the distance between 2 friends is Dijkstra's algorithm. Dijkstra's algorithm requires an input of all friend links to function. The friend links can be represented by nxn matrix, where n is the number of users. However, this is inefficient for a social network as a user will typically have a small number of friends in proportion to the overall number of friends, thus generating a sparse matrix. For this reason, the friends links are instead input using vectors in the yale format.

Rationale for Social Network Class:

The social network class acts as a container for the social network users, day-slots and events. It also provides functionality for analysis of the social network. Grouping both the storage of the social network data and the analysis of the social network is logical as the analysis makes use of the former.

Rationale for User:

The User class represent a user of the social network; it is an abstraction of an actual person. To this end, user contains the key identifying information of a person (such as age, name, etc.) This information allows analysis to be performed on the social network.

As the users are members of the wider social network, they have links to their friends. Their friends are in turn represented by instantiations of the user class. Thus, a user's friends are stored in a vector of pointers to respective users.

For comparison purposes, it is useful to assign an unchanging, unique identification number to a user. This number is generated through the social network. This is done by incrementing a counter within the social network and assigning the counter to new users.

Rationale for Events:

The Event class represents an event of the social network; it is an abstraction of an actual event. To this end, event contains key event information (such as start and end times, description, category and list of user participants).

Rationale for Day-Slot Class:

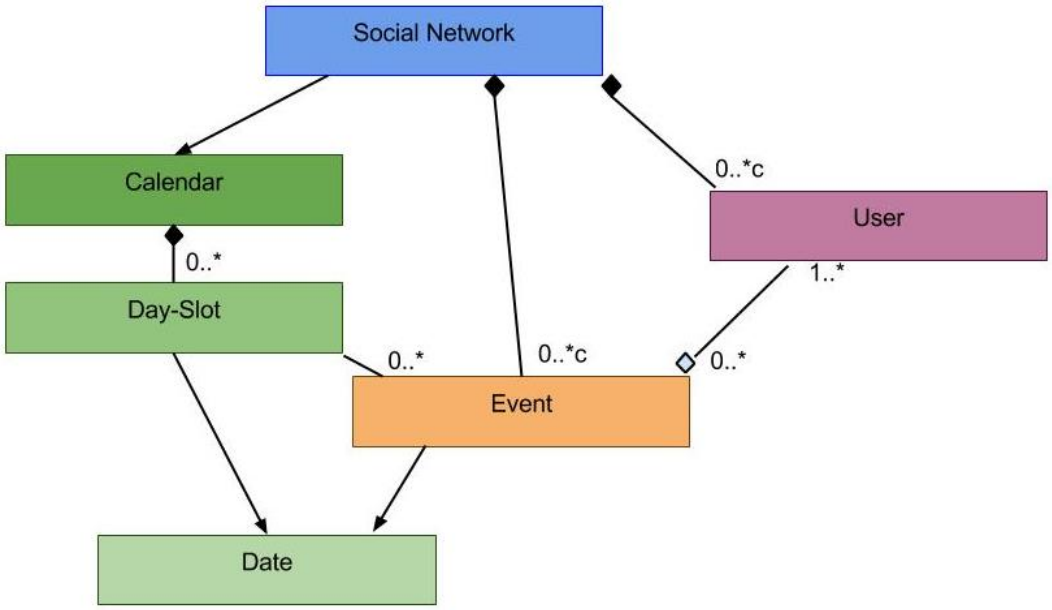
The Day-Slot class is implemented to provide a layer of sorting of events. A day-slot is associated with a particular day through a Date member and holds all of the pointers to the Events that are on that particular day. This allows events to be sorted by day as the events from a particular day can be access through their day-slot.

Rationale for Calendar Class:

The Calendar class is largely a container for an assembly of Day-Slot instances. This could likely have been forgone and the day-slots could simply have been stored in the social network itself, however looking forward into future functionality (such as a UI), it may be useful to have an abstraction of the Calendar and so it was included.

4. High-level view of the architecture of program

UML Class Relationship Diagram:



List of Classes:

Class	Description
Social Network	Contains master list of users and master calendar and master list of events. Contains analysis methods.
User	Represents a user of the social network. Associated with Events.
Calendar	Contains Day-Slots.
Day-Slot	Contains Events; associated with a particular day
Event	Represents an event attended by users. Contains a list of said users.
Date	Represents a day.

5. Contribution of each person

It is difficult to distinctly separate the contribution of each person as sections were collaboratively written and others directly reviewed and corrected collaboratively. That being said, here are some sections which may have been of individual focus.

The following class implementations were contributed by David:

- Social Network
    - Abstraction of the class
    - Querying methods
    - Common friends between two users method
- Social Network Driver Presentation
- Event
    - Querying methods
    - Addition of users to event class algorithm
- DaySlot
    - Querying methods
- User
    - Addition of events to user class algorithm

The following class implementation were contributed by John:

- Social Network
    - Shortest ‘path’ to friends method
    - User deletion algorithm
    - Accessor methods to access users in the class
- Event
    - Abstraction of the class
- DaySlot
    - Abstraction of the class
- User
    - Abstraction of the class

6. Future extensions for project

Some of the future extensions and functionality of the project which may be added are listed below:

Querying for a particular event that is popular with the Social Network, amongst category, amongst genders, amongst a range of age, and for a particular date:
Currently, events are given a category label. Using this label and the event participants, it will be possible to do further queries and analysis.

Create UI and input features:
Currently, user and event data is input (hard-coded) into the main driver file of the SN. Furthermore, all output from the SN to the user or SN administrator is currently displayed on the console. Thus, an important future extension for the project would be to implement a user-interface and input functionality.

Add interests of each user and recommendations for friends and events for user:
Currently, the implementation of user does not take account into the user’s interest. This parameter is important in providing recommendations of friends and events for users, as this parameter provides a common similarity basis on each user. A method to overcome the missing user’s interest feature is to implement a machine learning algorithm to determine the user’s interest based on the user’s participation and user’s friend’s participation in various event categories.

Making the events able to accommodate more than one day:
Currently, the duration of an event can only span one day. Because of the implementation of the DaySlot class, in order to allow the duration of an event to span more than one day, the event would have to be included in multiple day-slots.

Add other sorting classes into SN:

Currently, the SN has functionality through the Calendar and DaySlot classes to sort the events based on the day of the event. However, the SN could also have an additional class/container to sort the events based on other event attributes (such as category). The SN could also have additional classes/containers to sort users based based on user attributes.

Loading and storing Users and Events to and off memory

Currently, the Social Network stores all the information of Users and Events in the memory. Therefore, the User and Event informations cannot be transferable or saved elsewhere. A database implementation that stores the User and Event information to a file can overcome this problem.

7. Appendix A: UML diagrams

UML Diagram for User:

User
- user_number_: int - email_: string - first_name_: string - last_name_: string - age_: int - gender_: bool - friends_: vector<User*> - user_events_: vector<Event*>
+ User() + ~User() + getUserNumber() ; int + getEmail() : string + getFirstName() : string + getLastName() : string + getAge() : int + getGender() : bool + getFriends() : vector<User*> + getEvents : vector<Event*> + setEmail(email: string) : void + setFirstName(firstname : string) : void + setLastName(lastname: string) : void + setUserNumber(userNumber: int) : void + addFriend(new_friend: User*) : void + addEvent(new_event: Event*) : void + addEventM(new_event: Event*) : void

UML Diagram for Social Network:

Social Network
- user_number_counter : int - master_calendar_: Calendar - users_: vector<User*> - event_list_: vector<Event*>
+ SocialNetwork() + ~SocialNetwor() + getCalendar() : Calendar + getNumUsers() : int + getUser(user: User*) : User + getUserBasedOnName(firstname: string, lastname: string) : User* + getUsers() : vector<User*> + getEvent(event_name: string) : Event*

+ addNewUser(new_user: User*) : bool + addNewEvent(date: Date*, beginning_hour: int, ending_hour: int, name: string, description: string, event_creator: User*, category int) : bool + deleteUser(user: User*) : bool + findSmallestConnections(User1: User*, User2: User*) : int + findSimilarFriends(User1: User*, User2: User*) : int + getPercentageMale() : double + getPercentageFemale() : double + getPercentageAge(age_upper: int, age_lower: int) : double - incrementUserCounter() : void - CalculateShortestPath(NumUser : int, StartUser : int, EndUser : int, PathRecord: vector<vector<int>>, FriendsOfUser: vector<int>, UserInitialIndex: vector<int> - UpdatePathRecord(PathRecord: vector<vector<int>>*, CurrentNode int, DestinationNode: int)

UML Diagram for Event:

Event
- date_: Date* - beginning_hour_: int - ending_hour): int - length_: int - name_: string - description_: string - participants_: vector<User*> - category_: int - categories_list_[]: string
+ Event() + Event(event_date: Date*, beginning_hour: int, ending_hour: int, name: string, description: string, event_creator: User*, category: int) + ~Event() + getUserName() ; string + getDescription() : string + getCategory() : string + getDateString() : string + getDate() ; Date* + getParticipants() : vector<User*> + getBeginning_Hour() : int + getPercentageMale() : double + getPercentageFemale() : double + getPercentageAge(age_upper: int, age_lower: int): double + addUser(user: User*): bool + addUserM(user: User*): bool + deleteUser(user: User*): bool + setName(name: string): void + setDescription(description: string): void + setDate(date: Date*): void + setCategory(int category): void

UML Diagram for DaySlot:

DaySlot
- events_: vector<Event*> - date_: Date*
+ DaySlot() + DaySlot(date: Date*) + ~DaySlot() + getDate() : Date* + getEvents() : vector<Event*> + getEventsInHour(hour: int) : vector<Event*> + deleteEvent(event: Event*) : bool + addEvent(event: Event*) : bool + getPopularEvent() : vector<Event*> + getPopularEventForMales() : vector<Event*> + getPopularEventForFemales() : vector<Event*> + getPopularEventForAge(upper_limit: int, lower_limit: int) : vector<Event*>

UML Diagram for Date:

Date
- year_: int - month_: int - day_: int
+ Date() + Date(year: int, month: int, day: int) + Date(year: int, month: int, day: int) + ~Date() + getMonth() : int + getDay() : int + getYear() : int + toString() : string + setMonth(month: int) : void + setDay(day: int) : void + setYear(year: int) : void + setDate(year: int, month: int, day: int) : bool + setDate(year: int, month: string, day: int) : bool - monthStr2Num(month: string) : int - intToString(i int) : string - CheckforNeg(year: int, month: int, day: int) : bool - CheckforRealDate(year: int, month: int. day: int) : bool - CheckforLeapYear(year: int) : bool

UML Diagram for Calendar: