CENG 242

Homework #4

(Due: May 6th 2011, Friday 23:55)

In this homework, you will implement a class representing a social network of people connected by friendships, and a class encapsulating the information related to members of the network, with C++. The `SocialNetwork` class will provide an interface for manipulating and querying the community; and the `Person` class will contain information about the members of a network.

A social network can be considered as a graph in which the vertices are identified as members and the edges are identified as friendships. In the homework, this analogy is also used. When a path is mentioned between two members of a social network, a sequence of vertices, connected by edges (friendships), starting from the first user and ending on the second user, is meant. A simple path is a path that does not have repeating vertices.

As in the previous homework, you will provide an interface, hiding the internals of the class. You are free to implement helper methods within the class namespaces.

`Person` class has the following declaration:

```
class Person {

public:

     string name, email;

     gEnum gender;

     unsigned int age;

     bool operator==(const Person & p2);

};
```

Equality comparison operator (==) should return true if two `Person` instances have the same name, email, gender and age. You should use equality when checking whether a member is present in a social network.

You should implement the following methods in the `SocialNetwork` class (as public members of the class):

- `SocialNetwork();`

The constructor will initialize a `SocialNetwork` instance having zero members.

- `SocialNetwork(const SocialNetwork & rhs);`

Copy constructor will deep copy contents of the `SocialNetwork` object instance, `rhs`.

- `~SocialNetwork ();`

You should take care to clean up the memory content that will not be available after the destructor is called. Proper memory management will count towards your grade.

- `SocialNetwork & addMember(Person p);`

This method will add a new member to the community. It should return the calling `SocialNetwork` instance.

If the new member already exists in the network, method should throw an `Error` typed value, `MEMBEREXISTS`.

- `SocialNetwork & removeMember(Person p);`

This function will remove the member from network. All the associated friendships should be removed as well (remember that friendships are bidirectional, hence when a member is removed, he/she will be removed from the friend lists of his/her friends. It should return the calling `SocialNetwork` instance.

If a user `p` does not exist in the network, function should throw an `Error` typed value `MEMBERDOESNOTEXIST`.

- `SocialNetwork & addFriendship(Person p1, Person p2);`

This method will set up a bidirectional friendship between `p1` and `p2`. It should return the calling `SocialNetwork` instance.

If `p1` or `p2` does not exist in the network, function should throw an `Error` typed value `MEMBERDOESNOTEXIST`.

If the users are already friends, function should throw an `Error` typed value `ALREADYFRIENDS`.

- `SocialNetwork & removeFriendship(Person p1, Person p2);`

This function should remove the friendship between member `p1` and member `p2`. It should return the calling `SocialNetwork` instance.

If `p1` or `p2` does not exist in the network, function should throw an `Error` typed value `MEMBERDOESNOTEXIST`.

If the users are not friends, function should throw an `Error` typed value `NOTFRIENDS`.

- `SocialNetwork getFoFNetwork(unsigned int n);`

This function should return a Friend-of-a-Friend network, a new social network, in which a friendship is set up between two users if there exists a path starting from first user and ending with the second user with at most `n` *intermediate* members (vertices) in the calling `SocialNetwork` instance.

The new social network should have the same set of members. If there exists a path with at most `n` intermediate vertices (or `n+2` total vertices) in the calling social network between two members, they should be friends in the new network.

Clearly, when `n = 0`, the same social network should be returned.

- `void getRecommendations(Person p, unsigned int n, bool isSorted, Person const * & recommendations, int & rNum);`

This method will return a list of recommendations for the member `p`. `n` denotes the desired number of recommendations. You should place recommendations in an array of `Person` instances starting at `recommendations`. After the method is executed, the number of allocated person instances should be in `rNum` (0 if none is allocated).

If `isSorted` is false, then the recommendations will consist of a set of members that are closest to `p` (with respect to the length of the shortest path between members). The recommendation set may not be unique; you are free to generate any of the feasible recommendation sets. You should not include a `Person` who is already a friend of `p`. Recommendations should be sorted with respect to the shortest path distance to `p`, (order of equally distant vertices is not specified) with the closest vertex being the first element in `recommendations (recommendations[0])`.

If there are not `n` feasible recommendations, then you should return the maximal set of recommendations ordered as specified above.

If a user `p` does not exist in the network, function should throw an `Error` typed value `MEMBERDOESNOTEXIST`.

You should consider using `getFoFNetwork` method rather than the `getPath` method in the implementation of `getRecommendations`.

There is also 20 point bonus part in the implementation of `getRecommendations`. It is possible assign a weight to a path between two vertices as λ^k where k is the number of edges in the path and λ is a real number between 0 and 1. It is possible to define the affinity between two members as the sum of weights of simple paths between the vertices, i.e.

$$s(p_1, p_2) = \sum_{p \in P(p_1, p_2)} w(p),$$

where $P(p_1, p_2)$ is the set of simple paths between p_1 and p_2, $w(p)$ is the weight of path p. If when `isSorted` is true, you can generate a set of recommendations that consist of the members with highest affinity values between `p,` you will get the bonus. The recommendations should be sorted with respect to the affinity such that the person with the highest affinity to `p` should be the first element of `recommendations`. You should take $\lambda = 0.8$.

- `void getPath(Person p1, Person p2, Person const * & path, int & pathLen);`

This function will generate a simple path starting from user `p1` and ending on user `p2` (with p1 being the first element of `path`). You should place vertices in an array of `Person` instances starting at `path`. After the method is executed, the number of allocated person instances should be in `pathLen`.

If `p1` or `p2` does not exist in the network, function should throw an `Error` typed value `MEMBERDOESNOTEXIST`.

If there does not exist a path that connects users, function should throw an `Error` typed value `MEMBERSNOTCONNECTED`.

Any simple path connecting the members will suffice, you do not have to compute the shortest path.

- `SocialNetwork operator+ (const SocialNetwork & s2);`

`+` operator will merge two social networks. The resulting network should have the members from both communities. This method should identify two `Person` instances as one if the equality operator returns true when they are compared. In this case, only one `Person` instance will be present in the resulting social network.

In the resulting network, a friendship between two members should exist if such a friendship is present in one of the networks.

- `SocialNetwork operator- (const SocialNetwork & s2);`

This operator will return the intersection of two social networks. The members of the resulting network should be the intersection of the sets of members of two networks. In the resulting network, a friendship between two members should exist if these members are friends in both networks.

- `bool operator<= (const SocialNetwork & s2);`

`A <= B` will return true if social network A is a subset of social network B. A is a subset of B only if the members of A is a subset of members of B and the friendships in A is a subset of friendships in B.

- `bool operator==(const SocialNetwork & s2);`

`A == B` should return true only if members of A equal members of B and friendships in A equal friendships in B.

- `SocialNetwork& operator= (const SocialNetwork & s2);`

`=` operator will equate the members and friendships to members and friendships in `s2` (as in copy constructor you should deep copy contents of `s2`).

`Error` is an enumerated type:

- `enum Error {MEMBEREXISTS, MEMBERDOESNOTEXIST, NOTFRIENDS, ALREADYFRIENDS, MEMBERSNOTCONNECTED};`

For printing the content of a social network “<<” operator will be overloaded.

- `friend ostream & operator<<(ostream & ost, const SocialNetwork & s);`

<< operator should output the member details and emails of friends of the member. Members will be sorted with respect to the email field. There will not be two members in a social network having the same email. The formatting for the output is as follows:

```
member1.name            //You will print user details first, separated by
member1.email           //a newline. Users should be sorted with respect
member1.age             //to email field.
member1.gender          //Print gender as Male|Female (not as an int)
     friend1.email      //After user details, you should print the emails
     friend2.email      //of friends. They should be sorted as well.
     friend3.email      //Put 1 tab before the email.
          .
          .
          .
member2.name            //After the details and friends of first member
member2.email           //are printed, you should print the details of
member2.age                  //second member in order.
member2.gender
     friend1.email
     friend2.email
          .
          .
          .
member3.name            //Member details are printed until the last
     .                  //member of community.
     .
     .
```

Example:

Consider the following set of calls:

```
SocialNetwork s1;
Person p1 = {"Okan","okanATceng.metu.edu.tr", Male,25};
Person p2 = {"Kerem","keremATceng.metu.edu.tr",Male,25};
Person p3 = {"Can","hosgorATceng.metu.edu.tr", Male,25};

s1.addMember(p1);
s1.addMember(p2);
s1.addMember(p3);
s1.addFriendship(p1, p2);
s1.addFriendship(p2, p3);
s1.addFriendship(p1, p3);
```

If we identify Okan with node C, Kerem with node E and Can with node F, we obtain the social network in Figure.1.b.

Figure.1 illustrates the usage of operators, `getFoFNetwork` and `getPath`.
Clearly, `s1<=s2` or `s2<=s1` or `s1==s2` will return false.

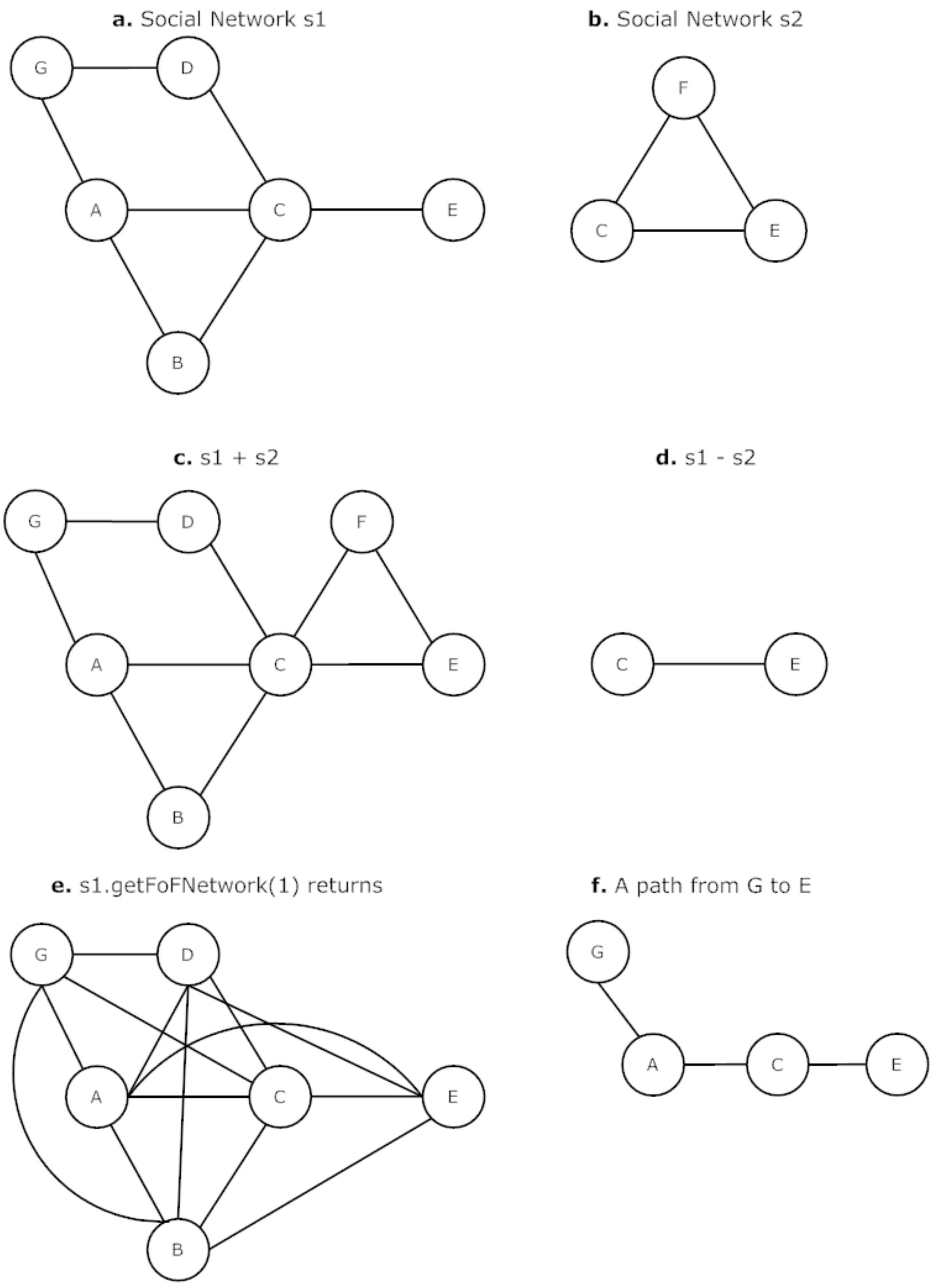

Figure 1

Two recommendations for E in `s1` are:

[D, A] or
[D, B] or
[A, B]

Three recommendations for E in `s1` are:

[D, A, B]

Four recommendations for E in `s1` are (note that G has to be the last element in the array, but the order of D, A, B is not important):

[D, A, B, G]

If we print the contents of s2, we obtain:

```
//--Output starts from the following line--
Can
hosgorATceng.metu.edu.tr
25
Male
      keremATceng.metu.edu.tr
      okanATceng.metu.edu.tr
Kerem
keremATceng.metu.edu.tr
25
Male
      hosgorATceng.metu.edu.tr
      okanATceng.metu.edu.tr
Okan
okanATceng.metu.edu.tr
25
Male
      hosgorATceng.metu.edu.tr
      keremATceng.metu.edu.tr
//--Output ends at the end of previous line--
```

Specifications:

1. There is not a restriction on the maximum number of members or friendships in a network.
2. Your implementation will also be tested on the basis of memory management with Valgrind. You can also use Valgrind to detect memory leaks in your implementation.
3. You can make additions to the provided hw4.h file; however you cannot remove or change any of the declarations. Even if you do not implement a method, include a dummy definition to make sure that your code compiles.
4. You are **not** allowed to use STL facilities with the exception of string and ostream classes.
5. All the work should be done individually.
6. In evaluation, black box method will be used. So be careful about the name of methods, data structures etc.
7. You will submit your code through Cow system.
8. In error conditions mentioned above, only throw the error. Do **not** write any code to catch it in your submission.
9. Your codes should be written in C++ and you should submit a tar file **hw4.tar**. The tar file should include **hw4.h**, **hw4.cpp**. In hw4.h, you should include class declarations, structures, enumerated types (Error, etc). In hw4.cpp, you should include the definitions. You should not have **main** function or any other global function in these files.
10. You should test your codes in **inek** machines by compiling with **g++** before submitting.