CSP301 Product 2: Social Network Simulation

The goal of this product is to answer simple english language queries about a social network. In particular, you will simulate an academic social network, spread potentially across multiple universities. In PART A, you will implement a discrete event simulator to create such a network. PART B will be able to answer user queries, posed in Engligh language, to answer queries about the network.

The Network.

The network has two categories of people: students and faculty. A (residential) university comprises departments, hostels, and faculty housing. All faculty join at the beginning before all students. The students join once a year. The faculty join a department and are allocated a house each. The students join a department and are allocated a hostel room each. In addition, each person has a set of interests/hobbies. The year is divided onto two (six month) semesters, and a list of courses are floated each semester. Each course is taught by one faculty member and a taken by a set of students. Each member is likely to make friends with people they come in contact with either by being neighbors, or being in the same class, or by having a common interest. Friends can also introduce other friends.

You will begin by reading an input file with parameters about courses, students, faculty etc. Based on these parameters and a randomization, you will "generate" individual members who join the university and continue simulating their befriending activities for a certain length of simulated time. This will produce the social network graph. In PART B, you will perform simple analysis on the graph.

PART A: **NetworkGen**

This module creates agents and simulates their behavior to produce a graph with nodes and edges. It consists of two processes.

1. One is ***TimeKeeper***. It keeps time in terms of the number of simulated seconds since its start. It advances the seconds at simulation speed as follows. It takes "alarm" requests by other "client" processes for a future time. If the request is for a past time, the request is denied (and the requester informed). Otherwise, the requester is woken up at the appropriate time. It can also be queried for the current time. Timekeeper advances time to the next alarmed event's time in sequence unless a client has requested a "time pause." Initially, the time is paused. You are free to choose the communication method – sockets and signals are candidates. It's clients are the threads of the ***Generator*** process described next.
2. The second process, ***Generator***, first runs submodule *setEnvironment* and then forks four co-operating generator threads. ***Generator*** executable takes a command-line parameter for the number of days the generator simulates as "-d num_days". Module *Network* owns the graph data structure and exposes function calls to generator threads. Submodule *setEnvironment* reads and parses an input file that contains information about each participating university with the following schema:

   UNIVERSITY university_name
   DEPARTMENT dept_name num_faculty num_students_per_year semester_dept_courses semeseter_nondept_courses
   COURSE course_number dept_that_offers frequency_per_year
   INTEREST interest_name popularity

HOSTEL hostel_name
HOUSELOCALITY locality_name
FRIENDSHIPRATE req_per_minute
OPENNESS out_probability
FRIENDLINESS recip_probability
RANDOMSEED faculty_random student_random course_random friend_random

Keywords are in caps. Values are in small letters. A UNIVERSITY keyword indicates the start of a new university's data. Note that courses, hostels, locality, etc. are specific to a university and another university using the same name or number are actually different. However, the interests are global across universities. program_course_requirement is the number of courses taken per semester by students of that department. An interest's popularity is the probability that a person is interested in it. Each hostel and locality has unit addresses starting from 1. As many addresses are generated as required by the number of people allocated to the hostel or locality. RANDOMSEED is global to the simulation and only the last instance of this keyword should be applied.

generateFaculty, creates faculty nodes. All faculty members are generated at time 0, at the beginning of simulation. Each generated faculty member has a sequentially increasing employee code, and a random name, department, house, and set of interests. These parameters are to be generated randomly to satisfy the DEPARTMENT parameters read by *setEnvironment*.

Thread g*enerateStudents* generates Student nodes, each with a sequential entry no and randomly generated name, department, hostel and a set of interests.

Thread *generateCourses* creates courses "once every semester." It generates courses and registers students. In other words it edits students' course data. It does not create any new nodes. Recall that the number of times a course is offered per year is specified in the input file. This frequency may be a real number. For example, a frequency of 1.5 means it is offered once in three semester – but more importantly this means it is offered in each semester with probability 1.5/2 (two semester in a year). This also means a frequency of 2 (or greater) implies a probability of 1. For each semester, this decision is made for each course in the order of the course numbers floated by the department. For each floated course, a faculty member is assigned in a round-robin fashion, in the order of their employee code. *generateCourses* also registers students in a course – each student is registered for "semester_dept_courses" courses of their department and "semester_nondept_courses" courses of some other (randomly chosen) departments. These numbers can be real. The fractional part is used to adjust it per student. For example, 3.4 means that three courses are added per student of the department but the fourth course is taken with the probability 0.4.

Thread *generateFriends* generates friending activities based on an exponential distribution with rate req_per_minute, r (i.e., pdf is $p(x) = re^{-rx}$). The following affinities are maintained for each friend event.

1. A random person is chosen to make a friendship request.
2. With out_probability, the request goes to any other random person and with 1-out_probability, the request goes to someone sharing a course, department, a neighborhood, or an interest, each with (conditional) probability 1/4.
3. A request is accepted with probability recip_probability (of the university to which the requested person belongs).

generateFriends logs all friendship requests and whether they were accepted. For random

number generators, you should use re-entrant random class from C++11. The provided random seed should be used for each generator thread at its beginning. You may uses *openMP* or *pthreads* to create threads. After all generator threads are completed, the last generator requests the *Network* submodule to save the generated graph in GraphML format (see http://graphml.graphdrawing.org for help and code). Make sure you manually kill *TimeKeeper* when the generators complete.

PART B: **GraphGyani**

This module is made of a perl script, *Gyani*, and and a C executable, *Analyzer*. *Analyzer* is started from within *Gyani*. *Gyani* reads query lines from stdin and parses each line. It makes the best guess about the query being made and packs the query to *Analyzer* and relays the answer back to stdout. *Analyzer* begins by reading the graph from the GraphML provided on the command line and performs computations to answer queries. It supports the following queries (you should implement Dijkstra's shortest path and Floyd–Warshall all-pair shortest path algorithms):

1. Size of Clique of a given person (identified by university name and ID)
2. The length of the shortest path between two given people
3. The list of people on the shortest path between two given people
4. The shortest path in the graph (between any pair of people)
5. The importance of the given person (the importance is the number of the all-pair shortest paths that include that person)
6. Is any of the friends of a given person more important that him/her

As a part of your deliverable, you must also produce documentation etc. as in the first product, In addition you should also produce valgrind and profiler output of both ***Generator*** and ***Analyzer*** along with your analysis of the output. You should use both to debug and analyze your program. Make sure the modules, submodules and threads are all in separate files. You should have a comprehensive makefile to build your apps and a bash script to run the apps in the right sequence so that a graph is first generated and then the queries are answered.