COMP1002 Assignment

Documentation

Bharath Sukesh

19982634

May 17, 2020

# Overview

This program is designed to simulate the spread of disease through a sample population. The program can be compiled by opening up a Terminal to the source folder and entering **javac \*.java**. The main method is located in the **healthSim.java** file, and the program runs by typing **java healthSim**. This will run the main() method and inform the user on how to operate the program with their desired mode. For either mode, input files are required to be given to load in the network, and the connections within. These input files have a very specific format that must be followed in order to load in everything appropriately. The format for the input files needed to be given in via command-line arguments are as follows:

|  |  |
| --- | --- |
| Type of file | Format |
| Network file | <Person name>**,**<Person age> |
| Edge file | <Person #1 name>**,**<Person #2 name> |

*Note: These String properties are split by a comma, with no spaces in between properties (Spaces are used only to separate first name and surnames, but not between two properties).*

An example of these input files are RandomNames30.csv & RandomEdges30.csv in the same working directory as this document and program.

If the interactive mode is chosen, the user interface/menu is run as specified in the assignment specification, but with added detail. The program uses a directed graph to represent a network of people in a population. The nodes within the graph represent one person, and the connections between nodes (edges) represent any contact shared between people and what type of contact they share between them. This means that a connection between Person 1 & Person 2 represents that they are in contact. Each type of contact between people has its own level of danger during a pandemic. A person may meet someone at school, where there are a large number of people within one vicinity; this evidently increases the risk of diseases spreading, whereas if you’re completely isolated to others at home, it is a much safer environment. I implemented the type of contact using an edge class for my Graph. Certain types of contact will increase or decrease the likelihood of infection.

If the simulation mode is chosen, several parameters are required to be given as command-line arguments (as shown in the prompt from **java healthSim**). The user will then be greeted by a prompt from the system that requires the user to enter who is the Patient Zero i.e. who is the first person that is infected. Once a valid person’s name is entered, the system prompts the user to run the timestep process. He/she can enter 1 (indicating ‘Yes’ - as shown on screen) until they feel happy with the model and then press 0 to exit the time-stepping process. Upon the conclusion of the time-stepping process, all relevant statistics will be appended to a unique file name saved to the same directory as the program. I chose to separate my network file and my edges file into two to avoid an overly messy settings file.

The ADT’s I ended up using were LinkedLists, Stacks/Queues within certain methods, and a Graph. I altered the graph class itself to my liking for the assignment. I could have used a Heap for the SIR model however I opted with LinkedLists for simplicity. I didn’t plan on using any other ADT’s as it would require a refactor/overhaul of my code. I opted for using CSV’s as input files however I do think I could have serialized my input to make user data safer (not safe enough, but safer than my current design with CSV files).

# Traceability Matrix

|  |  |  |  |
| --- | --- | --- | --- |
| **Feature** | **Requirements** | **Design/Code** | **Test** |
| **Driver/Menu & Modes** | 1.1. System displays usage if called without arguments. | healthSim.main() | UI Test: Live demonstration |
| 1.2. System displays interactive menu with "-i" argument. | healthSim.main() | UI Test: Live demonstration |
| 1.3. In interactive mode, user enters command and system responds. | healthSim.main() | UI Test: Live demonstration |
| 1.4. System enters simulation mode with "-s" argument. | healthSim.main() | UI Test: Live demonstration |
| 1.5. System returns appropriate error messages from wrong user input. | healthSim.main() | UI Test: Live demonstration |
| **File IO** | 2.1. To read file, System prompts user for filename. | UI.InteractiveMode() | UI Test: Live demonstration |
| 2.2. After reading valid file, the correct graph is produced. | FileIO.loadNetwork()  FileIO.loadEdges() | UnitTestDSAGraph.java or  Live demonstration |
| 2.3 If invalid file name, System returns appropriate error message. | FileIO.loadNetwork()  FileIO.loadEdges() | UI Test: Live demonstration |
| **Set Rates**  Side-note: “Invalid rates” can be both non-number rates and rates not between 0 and 1 | 3.1. In interactive mode, system prompts user for rates | UI.InteractiveMode()  which calls setRate functions in Network.java | UI Test: Live demonstration |
| 3.2. In simulation mode, rates are to be passed in via command-line arguments | UI.SimulationMode() | UI Test: Live demonstration |
| 3.3 If in interactive mode and user enters an invalid rate, System returns an appropriate error message. | healthSim.main() Calls setRate functions in Network.java | UI Test: Live demonstration |
| 3.4 If in simulation mode and user gives invalid rate on command-line argument, system displays error. | UI.SimulationMode() Calls setRate functions in Network.java | UI Test: Live demonstration |
| **Node operations** | 4.1. To add or remove nodes, system prompts user for node properties, system executes operation. | Network.newPerson()  which calls DSAGraph.addVertex() | UnitTestDSAGraph.java or  Live demonstration |
| **Edge operations** | 5.1. To add edge, system prompts user for name of two vertices to connect. | Network.newEdge() which calls DSAGraph.addEdge() | UnitTestDSAGraph.java or  Live demonstration |
| **New infection** | 6.1. To add infection, system prompts user for Patient Zero ID. System then updates status of patient zero in graph. | Network.newInfection () | UI Test: Live demonstration |
| **Display network** | 7.1. System displays network graph as an adjacency list. | Network.displayGraph() which calls DSAGraph.displayAsList() | UnitTestDSAGraph.java or  Live demonstration |
| **Update (timesteps)** | 8.1. In both modes, system runs update method. User can enter the same option again to continue time stepping. | Network.update() | UI Test: Live demonstration |
| **Display statistics – population** | 9.1. System displays overall population stats | Network.displayPopulationStatistics() | UI Test: Live demonstration |
| **Display statistics – person record** | 10.1. System prompts user for a person’s ID. System returns with output (find node operation) | Network.displayPersonStats() | UI Test: Live demonstration |
| **Display statistics – categories** | 11.1. System outputs categorized graph nodes | Network. displayHealthCategories() | UI Test: Live demonstration |
| **Save network** | 12.1. System saves nodes and edges into a file with unique file name. | FileIO.saveNetwork()  FileIO.saveEdges() | UI Test: Live demonstration |
| **Interventions** | 13.1 In Interactive mode, user selects desired intervention, system executes operation. | Network.applyIntervention() | UI Test: Live demonstration |
| 13.2 In simulation mode, system executes intervention operation based on command-line argument. | Network.applyIntervention() | UI Test: Live demonstration |

# UML

# Classes

* Unit test classes have been excluded from this section, as they share similar behaviour – they test all methods within their respective class and contain exception handling. The ‘nomenclature’ for these harnesses are to test class ‘X’, where the name of the test class is UnitTestX.
* I also chose to exclude most ADT classes (bar DSAGraph) as most of them were implemented from the lectures for the practicals with some edits made to cater to my program. The data structures I used were graphs, linked lists and queues.

## healthSim

Acts as a sort of ‘wrapper class’ for my system. This handles the command-line arguments of the program and calls the UI, based on user input. I also used this class for any last-minute exception handling. I decided to make this separate from the UI for the sake of class responsibility.

## UI

The UI class handles the network for both simulation and interactive mode. Most program outputs come from this class. The menu can be found in this class, and I added methods to simplify the code for getting user input for Interactive mode – scString/scDouble/scInt.

## Network

The network class does most of the heavy lifting for this simulation program. This class contains mutators and accessors for classfields of different Objects, as well as other functions that perform most of the system requirements like adding/removing edges/nodes etc. All of the timestep methods used in Simulation mode can be found in this class. The timestep algorithm itself is based on a mathematical modelling technique called an ‘SIR’ model.

## Person

An inner class within Network, Person class stores a single person’s name, age, status, timestep and most importantly, their infection rate. This class is mainly used for creating a Person object which is done elsewhere in the program. Inside Network, an instance of this class is used as a value for a node within the network graph. This class allows me to store all information contained into an object so that it can be stored in any ADT as a value.

## FileIO

This program relies on File IO to load the network graph and run the simulation. I thought it would be a good idea to isolate all the file handling for both reading and writing, into its own separate class. This class handles saving and loading for network files, connection files as well as appending statistics to files. I separated my file reading into two files instead of one. The reason for this is to keep the connections between people more secure rather than keeping all information between people into one ‘database’/place of information.

## DSAGraph

This class is an ADT of a bidirectional graph. This class contains the standard graph functionality of adding/removing both nodes & edges. It has been altered to include an edge class, which allows for functionality I desire within the system – to add a property to an edge – the type of contact between people. This class is important as it acts as the skeleton for our population network.

# Justification

## Data Structure Selection

I didn’t use many ADT’s apart from Linked Lists with (Queues within methods), Queues for traversing the graph and of course a Graph itself. These are the only ADT’s I thought were necessary to tackle this program, and I didn’t think any other ADT’s could work better bar maybe the use of Heaps and Hash Tables.

The system uses the Graph class to represent a network of people in a given population. People are represented as nodes and edges as connections between people. The graph used is undirected. This helps to enhance clarity, as the connection between users are inherently bidirectional in nature. Operations that must be performed within this directed graph are find/add/remove vertices and find/add/remove edges. Duplicate vertices can’t exist, and with the context of a simulation over a population, only one directed edge exists between 2 nodes. My graph vertex class (which represents a person) contains a Person object as aforementioned. I chose to make this vertex class public, as I deemed there was not a big deal of security concern/risk as the only personal information that is public, is the person’s name and age. Using a graph allows me to think graphically about how to approach the model and think graphically about how to complete all the operations in the program. In terms of overheads and time complexity, a graph is costly, especially with how many linked lists I used, but due to the integral role it plays in the program, I thought it was worth the sacrifice.

Edges & vertices are stored using linked lists. I also used linked lists for storing each infection occurring in the network. Linked lists are useful because of its variable size (it is a dynamic data structure); ability to store objects that contain information of any datatype, and simple methods for most operations e.g. inserting into and deleting from the data structure. A benefit to my program using linked lists, was its ability to store an object type as its value. This object can contain classfields of any datatype, making the linked list generic. This is commonly known flaw in other structures such as arrays, where you can only store one type of data in the entire data structure. Traversing the linked list is also very simple, by using either for-each loops or iterators, each with their similar but distinctive behaviour. For the implementation of interventions, I decided to design it such that not only does it alter the transmission rate (and hence the spread of infection) in a population, but it also removes connections between people. This means that I assume that everyone in a population will abide by the rules of an intervention for the sake of this simulation.

I did not use Hash Tables as I didn’t have enough time to refactor my code to suit it, however, I do admit that the use of a Hash Table to store data would have been much more optimized than a linked list, as the linked list approach gives a time complexity of O(#of V), and O(#E + V) for removing and finding edges. Adding vertices and edges are usually O(1), however since you need to check if there are vertices/edges already present within the graph, the time complexity is equivalent to that of a find operation. Despite this, a linked list worked well with my program as its large memory overhead was insignificant at this small of a scale.

*Note: All these ADT’s were previously submitted for DSA practicals. Linked lists and Queues were submitted for Practical 04, and Graphs in Practical 05.*

## Approach for Statistics & Timestep

There are three statistics to show – statistics on a certain person, per health category, and population statistics. People in my system were contained within a Person object – created by a constructor within my Person class. Each Person object contains data about that single person, e.g. their name, age, ID, health status. Statistics on a certain person was simple to implement, as you simply had to look up a person’s respective info/classfields and print it out. One of the classfields within my Person object was their health status. If at any time during the pandemic they got infected, or they recovered from an infection, their health status was to be updated using mutator methods. This allows me to display a sample population’s health statuses by category. I felt this was the easiest way to implement this. The third and final statistic was population statistics, which was implemented by storing the number of susceptible, infected, recovered & passed away, all into a linked list after each timestep. This then allowed me to iterate through the linked list to display all data per timestep. The time-stepping/updating process was implemented from example code given to us with the additional use of linked lists as aforementioned to store all information regarding time step.

The infection process itself was implemented using Queues to traverse through the network and find people that have been in contact with our patient zero (specified at the start of the program if in simulation mode or done via a menu option in interactive mode). This method was inspired by a breadth-first search algorithm, which could be seen as representative of a method used currently during COVID-19 called contact tracing. The likeliness of someone getting infected was judged by their individual infection rate. As I said in the overview, each person’s chances of getting infected are determined by their exposure to other people in an area and for how long that is. Each person has their individual infection rate, which is compared to a random number generator. If their infection rate is greater than that random number, they have the bad luck of being infected. If their infection rate is less, they’re assumed safe (for now).

## Modifications to generic ADT classes

Out of the ADT’s I used – graphs, linked lists, queues, the only ADT I did not alter, were queues.

## Linked Lists

* Linked lists were customised to suit my system by adding a customised *toString()* method. This was necessary as the output from this method is specific to the output of my timestep – exactly four contents per method call. These four outputs are specific statistics produced at each time an update is run; containing the current timestep, and the number of susceptible, infected, recovered & passed away.

## Graphs

* The graph was the most altered class in my program. I added an edge class, which allowed me to add specific properties to an edge (the type of contact between people). This also meant that I needed to change how my graph class itself worked to cater to an additional classfield – a ’edges’ linked list. I also changed how some generic operations worked – e.g. changing the parameters required to add a vertex to include an object. This makes each vertex contain some form of data. In my graph, each vertex will contain data unique to that person.

## Generic

As aforementioned, queues were untouched in my implementation of this program. I was happy with what the class provided and felt no need to change anything.