A4 – Modeling Covid-spreading with Trees

Table of Contents

1. Introduction

Note: Keep track of the time you spend on this assignment. You have to give it to us when you submit.

We all get bugs or infections from time to time. Sometimes they are just an annoyance. Right now, the world is dealing with a highly infectious and dangerous one, COVID-19. The United States Centers for Disease Control and Prevention (CDC) tracks how a virus is spreading and attempts to determine where it will spread next, so preparations can be made to fight it. People develop programs to simulate the spread of viruses in order to learn about them, just as programs are written to simulate weather. If you are interested in learning more about using Math/CS to model and understand infectious bugs, check out some of these resources:

- http://idmod.org/home
- https://en.wikipedia.org/wiki/Mathematical_modelling_of_infectious_disease
- http://ocw.jhsph.edu/courses/epiinfectiousdisease/pdfs/eid_lec4_aron.pdf

Here's an article on how computer models simulate the spread of the new coronavirus.

www.scientificamerican.com/article/heres-how-computer-models-simulate-the-future-spread-of-new-coronavirus/

A4 uses trees to model the spread of covid-19. The root of the tree represents the first person to get covid; the children of each node represent the people who were infected by contact with the person represented by that node.

In A4, you write code to manipulate and explore the tree. *We supply you with code that simulates the propagation of covid* as well as a GUI (Graphical User Interface) that allows you to set parameters and visualize the results on the screen. But the simulation won't work until you complete several methods to process the tree.

Learning objective: Become fluent in using recursion to process data structures such as trees. Learn how to test recursive methods on trees.

Collaboration policy: You may do this assignment with one other person. If you are going to work together, then, as soon as possible —and certainly before you submit the assignment —get on the CMS for the course and do what is required to form a group. Both people must do something before the group is formed: one proposes, the other accepts.

If you do this assignment with another person, you must *work together*. Usually, we say that it is against the rules for one person to do some programming on this assignment without the other person sitting nearby and helping. You should take turns "driving" —using the keyboard and mouse. If you are the weaker of two students on a team and you let your teammate do more than their share, you are hurting only yourself. You can't learn without doing.

However, during this pandemic, where you and your partner may not be physically together, we can't require the above. However, you should zoom often and program together, looking at each other's suggestions for a method, say, and agreeing on the best solution. This includes looking at and discussing test cases in the JUnit testing class. In Eclipse, using Saros allows you and your partner to work together on this project.

With the exception of your CMS-registered partner, you may not look at anyone else's code, in any form, or show your code to anyone else, in any form. You may not look at solutions to similar previous assignments in 2110. You may not show or give your code to another person in the class. While you can talk to others, your discussions should not include writing code and copying it down.

Academic Integrity: With the exception of your CMS-registered partner, you may not look at anyone else's code from this semester or from a similar assignment in a previous semester, in any form, or show your code to anyone else, in any form. You may not show or give your code to others until after the last deadline for submission. Finally, groups must follow the rules stated above in section *Collaboration policy.*

Getting help: If you don't know where to start, if you don't understand, if you are lost, etc., See someone IMMEDIATELY—a course instructor, a TA, a consultant, the Piazza for the course. Do not wait.

Tutorials —important: Before starting to work on A4, *watch the two tutorials about recursion on trees* in the JavaHyperText: Look in the second horizontal navigation bar at the top of the page for "Recursion".

Every recursive method that you write in A4 has a comment like this:

```
// State whether this is a searching or a counting method:
```

We expect you to type either searching or counting after the colon. Point will be deducted if you don't do this. Reason: Countless people in the past have not watched the tutorials about recursion on trees for similar assignments, didn't know what they were doing, and had all sorts of errors in these recursive methods. They wasted their time, and staff time, because of not preparing properly. We are just attempting to get you to prepare properly, so that you have an easier time with this assignment.

2. Covid-infection propagation

A4 uses a very simple model of an infectious covid, which provides a real-world example of the general tree data structure. You are not responsible for writing any of the code that implements the simulation of the covid spreading, but the simulation won't work until you write methods that manipulate trees.

The model is initialized with a population of people —1, 2, 10, 50— however many you choose. Each human has a health range, say 0..10. A health of 10 means the human is very healthy and 0 means the human has died. In the beginning, everyone is as healthy as possible. Generally speaking, the larger the health range, the longer the program will run since there are more possible states of the simulation.

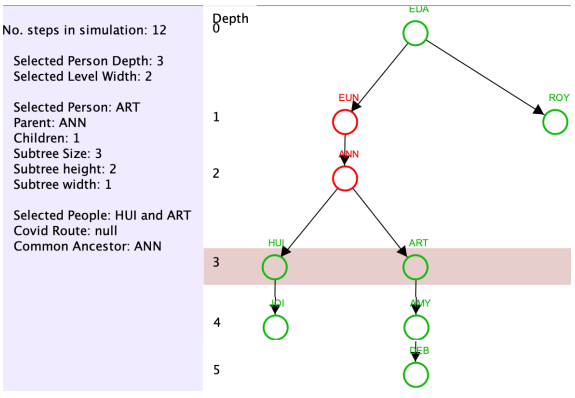
At the beginning of the simulation, you supply several probabilities: (1) The probability that any two humans are close to each other —call them *neighbors*. A probability of 1 means everyone is close to everyone; a probability of 0 means no one is close to anyone. (2) The probability that a human will get covid from a neighbor. (3) The probability that a human will become immune —once immune, the person is healthy forever.

Once you have given this input, the program starts by making one random person ill and making that person the root of a new covid tree. Initially, that is the only node in the tree.

A series of time steps follows. In each time step, several things happen.

(1) one random healthy neighbor of each ill human may get covid and be inserted into the covid tree as the ill human's child.
(2) Each ill human either gets well and henceforth immune from covid or gets more ill (their health decreases).
(3) An ill human whose health decreases to 0 dies.

These time steps continue until everyone in the tree is either healthy or dead. When the simulation is over, the results are shown in a GUI on your monitor, as shown to the right.

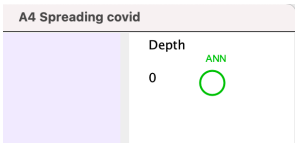


EDA is at depth 0 (the root); EUN and ROY are at depth 1; ANN is at depth 2, etc.

EDA got covid first. EDA infected EUN and ROY. EUN infected ANN. ANN infected HUI and ART. You can infer the rest. JOE infected DEB; and DEB infected EUN. All of them got ill, but all of them became immune and healthy (indicated by a green circle) except EUN and ANN, who died (indicated by a red circle).

The information in the panel to the right appears only when you click some node of the tree. The data changes as you click on different nodes of the tree. The data that appears in the panel comes from selecting (with your mouse) HUI and then ART. Both are at depth 2 in the tree and they are the only people at that level. The common ancestor of HUI and ART is ANN. Info is given for the last selected person, ART: his depth is 3; parent ANN; number of children 1; subtree size 3; subtree height 2 and subtree width 1.

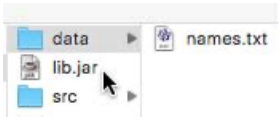
An execution may result in a tree with one node, as shown to the right. This happens when the one person with covid doesn't infect anyone —they may have no neighbor or, when generating a random number to test whether a neighbor gets infected, the result is always "no".



3. Installation

1. Download the A4 assignment zip file from the Canvas Assignment page for A4.

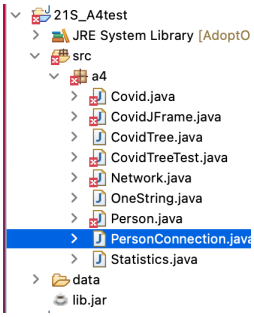
2. Unzip the downloaded file. The folder should contain file lib.jar and two folders, data and src. Folder data should look as shown to the right.



3. Create a new Java project (call it a4, for example). **IMPORTANT** – you must use java 11 for this project. Do NOT create a new module-info.java file.

Copy-paste all three items in the downloaded folder into the root of the new Java project in Eclipse (i.e. data, lib.jar, and src). When asked how to copy, select "Copy files and folders", then OK. Also, if asked to replace src, do it.

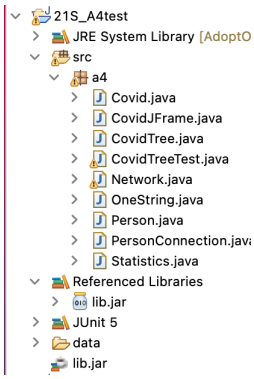
To the right, you see what folder src should be. Five of the files have syntax errors, which will be fixed in a moment.



4. Add lib.jar to the build path by right clicking lib.jar and selecting Build Path → Add to Build Path. After this, the only file that should show an error is CovidTreeTest.java.

5. Create a new JUnit testing class as usual (File → New → JUnit Test Case). Select *New JUnit Jupiter test*. Later, if it asks whether JUnit 5 should be added to the build path, say "yes". Then delete the newly added JUnit test case file (because you don't need it).

Your project should now look as shown to the right, and you're ready to go!



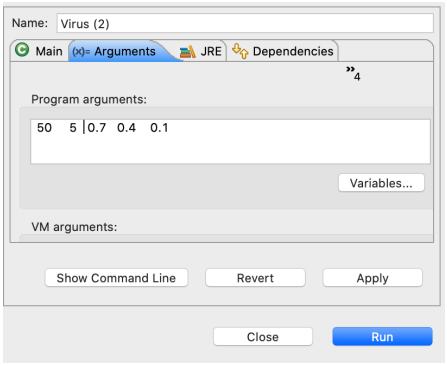
4. Running

The executable class of this project is Covid.java. To run the project, open Covid.java and click run (green button with white arrow) or use menu item Run -> Run. You will be prompted for a few values via the console at the bottom of Eclipse. These are:

1. Size of population: how many people to model. A positive integer. A higher number may result in a larger tree.
2. Amount of health per person: how long a person can have covid before dying. A positive integer. A higher number may result in a larger tree.
3. Probability of connection: how likely two random people are likely to be neighbors). A float in the range [0,1]. A higher number may result in a larger tree.
4. Probability of getting covid: how likely a person is to get covid when in contact with an infected neighbor. A float in the range [0,1]. A higher may number result in a larger tree.
5. Probability of becoming immune: how likely a person with covid will become immune (fight off covid). A float in the range [0,1]. A *lower* number may result in a larger tree.

A nice set of starting values is: (50, 5, 0.7, 0.4, 0.1)

If you want to run Covid.java repeatedly with the same five arguments, put them in the program arguments instead of having to type them into the console every time you run the program. This is done the usual way (Run Configurations… → arguments). For example, the run configurations shown to the right would run the program with the above arguments every time the run button is clicked.



If there is any issue with the arguments provided, either though the console or the program arguments (health is less than 0, for example), you will be re-prompted to enter the arguments through the console. Thus, if you entered arguments in the arguments tab but are still prompted to enter arguments via the console, there may be something wrong with the arguments you entered.

From there, the covid infection will run —starting with a randomly chosen patient and spreading across that human's connections until no one is left alive and infected.

After the simulation has finished, the full tree is printed out by calling `toStringVerbose()`; then the tree explorer GUI pops up. But the GUI doesn't print anything.

5. Your Tasks

All the code you write is in `CovidTree.java` and `CovidTreeTest.java`. Complete and test each method marked with a //TODO comment (in Eclipse, they are marked in blue on the right of the text) in the order in which they are numbered. *Do not delete the //TODO comments*. If a precondition is false, any behavior is acceptable; you need not write assert statements for Preconditions, but you can if you want.

Do the following before you code:

1. Read the comments at the top of `CovidTree.java`.
2. Study the already written methods in `CovidTree.java`. You will learn a lot by that.
3. You *must* read the Piazza pinned A4 note @1003 on this assignment. It continues useful and necessary info.

Recursion is your friend! The bulk of these functions are best and most easily written using recursion. Iteration (using loops instead of recursive calls) may be possible but will certainly be more work both to reason through and to debug.

Hint: Some of the functions you are asked to write can be written very simply or even trivially (one or two lines) simply by relying on previous functions you have already written or ones that we wrote.

Warning: Every time application `Covid` (i.e. method main in `Covid.java`) is called, a new, random, unrepeatable `CovidTree` is created, so you cannot debug the methods you are writing using that application. It is best to use the JUnit testing class to test your methods and to run the program only when you know your methods are correct. See the pinned Piazza note @566, *A4: Recursion on trees*.

Dos and Don'ts:

- Before you begin coding, read the comments at the top of class `CovidTree`! Also, read the specs of the methods, including the ones you don't have to write.. You may choose to read the Javadoc in other files, but it should not be too important. Read their source code only if you are particularly interested in them —you don't need to know more than their javadocs in order to complete the assignment.
- Don't alter any of the other files given to you. You won't be able to submit them, so your file `CovidTree.java` must work with unaltered versions of the other files.
- Don't change the method signatures of any method in class `CovidTree`. The name and types of parameters should not be changed.
- Don't leave println statements (which you may have added to help debug) in your code when you submit. You may lose up to 5 points. Comment them out or delete them before submitting.

- You may add new methods to class `CovidTree` to help complete the required functions. When we practiced the assignment, we wrote one or two. Make the methods you add are **private** and write good javadoc (/** … */) specifications.

6. What to submit

Change the value assigned to static field `CovidTree.timeSpent` from -1 to the time you spent to complete assignment A4. It is the first field in class `CovidTree`. Read the comment above it for help in doing this.

In the comment at the beginning of file `CovidTree.java`: put your netid(s), names(s), and a comment on what you thought about the assignment.

Submit file `CovidTree.java` on the CMS.

7. Suggested timetable for doing A4

You will find this assignment easier to accomplish and a more educational experience if you do it at a more leisurely pace, starting right now. When you run across a problem, you'll have time to think about it and to get help, if you need it. Our staff will have an easier time, too. Here is a suggested timetable for A4.

- 26 Mar. Read the handout and comments at the top of `CovidTree.java`, study the functions that are already written. Watch the two short videos on recursion with trees.
- 28 Mar. Write and test functions `insert`, `size`, `contains`. Methods size and contains give instructions on what to read to learn about testing tree-changing methods.
- 30 Mar. Write and test functions `depth` and `widthAtDepth`.
- 31 Mar. Write and test `covidRouteTo` and `commonAncestor`. *Be prepared to spend time on them*.
- 02 Apr. Write and test `equals`. *This also will take time if you do not understand everything*.
- 04 Apr. Look over everything. Submit, before the deadline.

8. Testing/debugging methods that process trees

We give you JUnit class `CovidTreeTest`. It contains a testing procedure for every method you have to write. Each has *one* testcase. *It is up to you to add more test cases and test your functions thoroughly*. Use Eclipse's code coverage feature to help you. If you don't know about Eclipse's code coverage feature, type *coverage* into JavaHyperText and read about it. Also, don't forget to test extreme/corner cases.

You do not have to submit your test cases, as you did for A3.

For assignment A3 on linked lists, we showed you how testing became easy when `toString` and `toStringR` were available. You didn't have to worry about which fields to test, you just tested them all!

Here, we have more problems testing because trees are a lot more complicated than linked lists. We see two issues: (1) Which trees should be tested and how do we construct them? (2) How do we simplify writing lots of test cases? Thinking about these issues ahead of time and preparing for testing can help.

To help you address these issues, consider the following.

1. Procedure test1Insert(). Our given test case shows you a bit about testing. First, class CovidTreeTest has several static variables of type Person: personA, personB, personC, ..., personL. Use these as shown in the code in this method. Note two things:

(a) personA.name() returns the name "A", and similarly for the others.

(b) The code uses the call toStringBrief(st) to get a string representation of CovidTree st. Look at the specification of toStringBrief for examples of calls on this method. This will show you how we can represent a tree in a String.

Here's an interesting thing about method toStringBrief(st): When processing the set of children of a node, it sorts them based on their root name. This achieves a unique representation of any tree. Without this, at different times,

the children of a tree could appear in different orders in the output.

2. Procedure test2Size(). At the beginning of this procedure, we direct your attention to a discussion near the top of the class that tells you how to construct a tree and test using this tree. You will learn more about personA and other static variables in this class. We are using a "just-in-time" method of teaching you here; there is no need for you to know about this material until now.

**3. Procedure test3contains(**). Again, in this method, we direct your attention to a further discussion of material that you can use to test this method (and later ones).

4. Procedure test4depth(). Again, in this method, we direct your attention to a further discussion of material that you can use to test this method (and later ones).

9. Hint

To learn about the foreach statement to process the elements of a set, e.g. for (String s : set), see JavaHyperText entry "foreach loop".