PROGRES - Mini-Projet 3
Campagne de Simulation

Sébastien Tixeuil - Sebastien.Tixeuil@lip6.fr

Le but de ce mini-projet est d'utiliser un simulateur fourni pour mener une campagne de simulation : exécuter le simulateur avec de nombreux paramètres différents, collecter et manipuler les données produites par le simulateur, obtenir des graphiques résumant l'impact des paramètres suivis sur les indicateurs choisis.

Ce simulateur est écrit en Java et modélise la propagation d'un virus dans une population. Il est fournit sous la forme d'un fichier `Virus.jar` qui intègre toutes les bibliothèques nécessaires. On peut lancer le simulateur avec les paramètres par défaut avec la commande (Java doit être installé sur la machine qui exécute) :

```
java –jar Virus.jar
```

Le code source du simulateur est fourni en annexe. Ce code source est fourni à titre indicatif pour comprendre la signification de chaque paramètre passé au simulateur. En particulier, **il n'est pas nécessaire de modifier le code source du simulateur pour réaliser le mini-projet.**

Note: En raison du grand nombre de simulations à mener lors de la campagne, il est fortement conseillé de commencer les exécutions multiples du simulateur dès que possible.

Exercice 1. Collecter des données expérimentales

Réaliser en Python et en utilisant la bibliothèque `subprocess` un programme qui exécute plusieurs fois le simulateur avec les mêmes paramètres de simulation. Ces exécutions multiples seront utilisées pour obtenir des intervalles de confiance suffisants lors de la phase de visualisation des données expérimentales. Les résultats intermédiaires doivent être stockés dans un fichier unique pour l'ensemble des simulations menées (pour un ensemble de paramètres donné).
Dans un deuxième temps, réaliser en Python un programme qui exécute le simulateur en faisant varier un paramètre de la simulation, en réalisant plusieurs simulations pour chaque ensemble de paramètres. Les résultats intermédiaires doivent être stockés dans un fichier ou un ensemble de fichiers pour l'ensemble des simulations menées.

Exercice 2. Manipuler des données expérimentales

A l'aide des fichiers produits lors de l'exercice 1, réaliser en Python et en utilisant les bibliothèques `numpy` et `pandas` un programme qui permette d'obtenir les informations suivantes :

- Quel est l'impact du nombre initial d'infectés sur (i) la durée de l'épidémie (ii) la proportion maximale de la population qui est infectée, (iii) la distribution des multi-infections ?
- Quel est l'impact du rayon de mobilité sur (i) la durée de l'épidémie (ii) la proportion de la population qui est infectée, (iii) la distribution des multi-infections ?

- Quel est l'impact de la durée d'infection sur (i) la durée de l'épidémie (ii) la proportion de la population qui est infectée, (iii) la distribution des multi-infections ?
- Quel est l'impact de la durée de contagiosité sur (i) la durée de l'épidémie (ii) la proportion de la population qui est infectée, (iii) la distribution des multi-infections ?
- Quel est l'impact de la durée d'immunité sur (i) la durée de l'épidémie (ii) la proportion de la population qui est infectée, (iii) la distribution des multi-infections ?
- Quel est l'impact de la densité de population sur (i) la durée de l'épidémie (ii) la proportion de la population qui est infectée, (iii) la distribution des multi-infections ?

Les informations ainsi produites seront stockées dans un ou plusieurs fichiers.

Exercice 3. Tracer des données expérimentales

A l'aide des fichiers produits lors de l'exercice 2, réaliser en Python et en utilisant la bibliothèque `matplotlib` un programme qui produit des graphiques pertinents pour évaluer l'impact des paramètres suivis (nombre initial d'infectés, rayon de mobilité, durée de contagiosité, durée d'immunité, densité de population) sur les indicateurs sélectionnés (la durée de l'épidémie, la proportion de la population qui est infectée, la distribution des multi-infections).

Annexe 1. Fichier `Virus.java`

```
import io.jbotsim.ui.JViewer;

public class Virus {
    public static void main(String[] args){
        int nb_nodes = 100;
        int snapshot = 10;
        int nb_snapshots = -1;
        int width = 1000;
        int height = 1000;
        int gui = 1;
        int infected = 2;
        int printout = VirusTopology.CATEGORIES;
        int show_parameters = 1;
        int stop_all_sane = 0;

        int infection_period = 100;
        int contagion_period = 200;
        int immune_period = 300;
        int travel_distance = 200;

        if(args.length > 0) {
            for(String s:args) {
                String[] option = s.split("=");
                if(option[0].equals("-nb_nodes")) {
                    nb_nodes = Integer.parseInt(option[1]);
                }
                if(option[0].equals("-snapshot_period")) {
                    snapshot = Integer.parseInt(option[1]);
                }
                if(option[0].equals("-nb_snapshots")) {
                    nb_snapshots = Integer.parseInt(option[1]);
                }
                if(option[0].equals("-width")) {
                    width = Integer.parseInt(option[1]);
                }
                if(option[0].equals("-height")) {
                    height = Integer.parseInt(option[1]);
                }
                if(option[0].equals("-gui")) {
                    gui = Integer.parseInt(option[1]);
                }
                if(option[0].equals("-nb_infected")) {
                    infected = Integer.parseInt(option[1]);
                }
                if(option[0].equals("-infection_period")) {
                    infection_period = Integer.parseInt(option[1]);
                }
                if(option[0].equals("-contagion_period")) {
                    contagion_period = Integer.parseInt(option[1]);
                }
                if(option[0].equals("-immune_period")) {
                    immune_period = Integer.parseInt(option[1]);
                }
                if(option[0].equals("-travel_distance")) {
                    travel_distance = Integer.parseInt(option[1]);
```

```
                }
                if(option[0].equals("-printout")) {
                    printout = Integer.parseInt(option[1]);
                }
                if(option[0].equals("-stop_all_sane")) {
                    stop_all_sane = Integer.parseInt(option[1]);
                }
                if(option[0].equals("-show_parameters")) {
                    show_parameters = Integer.parseInt(option[1]);
                }
                if(option[0].equals("-help")) {
                    System.out.println("Parameters:");
                    System.out.println(" -contagion_period: [=X] nodes
contaminate others for X time units on average (default 200)");
                    System.out.println(" -gui:              [=0] do not show
GUI");
                    System.out.println("                    [=1]* show GUI
(default)");
                    System.out.println(" -height:           [=X] area is X
units tall (default 1000)");
                    System.out.println(" -help:             show this
message");
                    System.out.println(" -immune_period:    [=X] nodes remain
immune for X time units on average (default 300)");
                    System.out.println(" -infection_period: [=X] nodes get
infected if they remain close to an infected node for X time units on average
(default 100)");
                    System.out.println(" -nb_infected:      [=X] initial
number of infected nodes (default 2)");
                    System.out.println(" -nb_nodes:         [=X] initial
number of total nodes (default 120)");
                    System.out.println(" -nb_snapshots:     [=X] total number
of snapshots before simulation stops (default -1: infinite)");
                    System.out.println(" -printout:         [=0] do not print
anything");
                    System.out.println("                    [=1]* print
#sane, #infected, #immune (default)");
                    System.out.println("                    [=2] print the
status of every node");
                    System.out.println(" -show_parameters:  [=0] do not show
parameters");
                    System.out.println("                    [=1]* show
parameters (default)");
                    System.out.println(" -snapshot_period:  [=X] make a
snapshot every X time units (default 10)");
                    System.out.println(" -stop_all_sane:    [=0]* do not stop
the simulation when all nodes are sane (default)");
                    System.out.println("                    [=1] stop the
simulation when all nodes are sane");
                    System.out.println(" -travel_distance:  [=X] nodes can
move within a X*X square (default 200)");
                    System.out.println(" -width:            [=X] area is X
units wide (default 1000)");
                }
            }
        }
        if(show_parameters!=0) {
            System.out.println("-----");
```

```
            System.out.println("Simulation parameters:");
            System.out.println("Using " + nb_nodes + " nodes.");
            System.out.println("Using " + snapshot + " snapshot period.");
            System.out.println("Using " + nb_snapshots + " as maximum number
of snapshots.");
            System.out.println("Using " + width + " as maximum number of
snapshots.");
            System.out.println("Using " + height + " as height.");
            System.out.println("Using " + gui + " as GUI option.");
            System.out.println("Using " + infected + " infected nodes
initially.");
            System.out.println("Using " + infection_period + " as infection
period.");
            System.out.println("Using " + contagion_period + " as contagion
period.");
            System.out.println("Using " + immune_period + " as immune
period.");
            System.out.println("Using " + travel_distance + " as travel
distance.");
            System.out.println("Using " + printout + " as printout option.");
            System.out.println("Using " + stop_all_sane + " as stop_all_sane
condition.");
            System.out.println("-----");
        }
        VirusTopology tp = new VirusTopology(snapshot, nb_snapshots,
printout, stop_all_sane );
        tp.setDefaultNodeModel(VirusNode.class);
        tp.setDimensions(width,height);
        tp.generateNodes(nb_nodes, travel_distance);
        tp.infectNodes(infected);
        VirusNode.setParameters(infection_period, contagion_period,
immune_period, travel_distance);
        if(gui == 1) {
            new JViewer(tp);
        }
        tp.start();
    }
}
```

Annexe 2. Fichier `VirusTopology.java`

```
import io.jbotsim.core.Topology;
import io.jbotsim.core.Node;
import java.util.concurrent.ThreadLocalRandom;

public class VirusTopology extends Topology {
    public static final int NOTHING = 0;
    public static final int CATEGORIES = 1;
    public static final int NODES = 2;

    int snapshot_period = 10;
    int nb_snapshots = -1;
    int remaining_clock = 0;
    int printout = CATEGORIES;
    int stop_all_sane = 0;

    VirusTopology( int snapshot, int nb_max_snapshots, int print, int stop )
{
        snapshot_period = snapshot;
        nb_snapshots = nb_max_snapshots;
        printout = print;
        stop_all_sane = stop;
    }

    public void generateNodes(int nb, int distance) {
        for(int i=0; i<nb; i++) {
            int x = ThreadLocalRandom.current().nextInt(distance/2,
getWidth()-distance/2);
            int y = ThreadLocalRandom.current().nextInt(distance/2,
getHeight()-distance/2);
            addNode(x,y);
        }
    }

    public void infectNodes(int nb) {
        for(int i = 0; i<nb; i++) {
            int id = ThreadLocalRandom.current().nextInt(0,
getNodes().size());
            ((VirusNode)findNodeById(id)).setStatus(VirusNode.INFECTED);
        }
    }

    public boolean isAllSane() {
        for(Node n: getNodes()) {
            VirusNode vn = (VirusNode)n;
            if(!vn.isSane()) {
                return false;
            }
        }
        return true;
    }
    public void printCategories() {
        int nb_sane = 0;
        int nb_infected = 0;
        int nb_immune = 0;
        for(Node n: getNodes()) {
```

```
            VirusNode vn = (VirusNode)n;
            if(vn.isSane()) {
                nb_sane++;
            }
            else {
                if(vn.isInfected()) {
                    nb_infected++;
                }
                else {
                    if(vn.isImmune()) {
                        nb_immune++;
                    }
                }
            }
        }
        System.out.println("" + nb_sane + " " + nb_infected + " " +
nb_immune);
    }
    public void printNodes() {
        for(Node n: getNodes()) {
            VirusNode vn = (VirusNode)n;
            if(vn.isSane()) {
                System.out.print("0 ");
            }
            else {
                if(vn.isInfected()) {
                    System.out.print("1 ");
                }
                else {
                    if(vn.isImmune()) {
                        System.out.print("2 ");
                    }
                }
            }
        }
        System.out.println();
    }
    @Override
    public void onClock() {
        if(nb_snapshots == 0) {
            System.exit(0);
        }
        if( stop_all_sane != 0 ) {
            if(isAllSane()) {
                System.exit(0);
            }
        }
        if( remaining_clock == 0 ) {
            pause();
            if(printout == CATEGORIES) {
                printCategories();
            }
            else {
                if(printout == NODES) {
                    printNodes();
                }
            }
            remaining_clock = snapshot_period;
```

```
            if(nb_snapshots != -1) {
                nb_snapshots--;
            }
            resume();
        }
        else {
            remaining_clock--;
        }
    }
}
```

Annexe 3. Fichier `VirusNode.java`

```
import io.jbotsim.core.Node;
import io.jbotsim.core.Message;
import io.jbotsim.core.Color;
import io.jbotsim.core.Point;
import java.util.concurrent.ThreadLocalRandom;

public class VirusNode extends Node{
    static final int SANE = 0;
    static final int INFECTED = 1;
    static final int IMMUNE = 2;
    static final Color all_colors[] = { Color.green, Color.red, Color.blue };

    static int infection_period = 100;
    static int contagion_period = 200;
    static int immune_period = 300;
    static int travel_distance = 100;

    static void setParameters( int infection, int contagion, int immune, int
distance ) {
        infection_period = infection;
        contagion_period = contagion;
        immune_period = immune;
        travel_distance = distance;
    }

    int status = 0;
    int contact_since = 0;
    int infected_since = 0;
    int immune_since = 0;
    int distance = 0;
    Point initial_coord = null;
    int infection_period_random = 0;
    int contagion_period_random = 0;
    int immune_period_random = 0;

    void setStatus(int s) {
        status = s;
        setColor( all_colors[status] );
    }
    boolean isSane() {
        return status == SANE;
    }
    boolean isInfected() {
        return status == INFECTED;
    }
    boolean isImmune() {
        return status == IMMUNE;
    }

    public void randomizeDestination() {
        int next_x = ThreadLocalRandom.current().nextInt((int)initial_coord.x
- (int)(travel_distance/2), (int)initial_coord.x + (int)(travel_distance/2));
        int next_y = ThreadLocalRandom.current().nextInt((int)initial_coord.y
- (int)(travel_distance/2), (int)initial_coord.y + (int)(travel_distance/2));
        setDirection(new Point(next_x,next_y));
```

```
        distance = (int)Math.round(distance(next_x,next_y));
    }
    public void randomizePeriods() {
        infection_period_random =
ThreadLocalRandom.current().nextInt(infection_period - infection_period/3,
infection_period + infection_period/3);
        contagion_period_random =
ThreadLocalRandom.current().nextInt(contagion_period - contagion_period/3,
contagion_period + contagion_period/3);
        immune_period_random =
ThreadLocalRandom.current().nextInt(immune_period - immune_period/3,
immune_period + immune_period/3);
    }
    @Override
    public void onStart() {
        // JBotSim executes this method on each node upon initialization
        initial_coord = getLocation();
        randomizeDestination();
        randomizePeriods();
    }

    @Override
    public void onSelection() {
        // JBotSim executes this method on a selected node
        if( isSane() ) {
            setStatus(INFECTED);
        }
    }

    boolean canBeInfected() {
        for(Node node: getNeighbors()){
            if(((VirusNode)node).isInfected()){
                return true;
            }
        }
        return false;
    }

    @Override
    public void onClock() {
        // JBotSim executes this method on each node in each round
        if( isSane() ) {
            if (canBeInfected()) {
                contact_since++;
                if (contact_since > infection_period_random) {
                    contact_since = 0;
                    setStatus(INFECTED);
                }
            } else {
                contact_since = 0;
            }
        }
        else {
            if( isInfected() ) {
                infected_since++;
                if (infected_since > contagion_period_random ) {
                    infected_since = 0;
                    setStatus(IMMUNE);
```

```
                }
            }
            else {
                if( isImmune() ) {
                    immune_since++;
                    if(immune_since > immune_period_random ) {
                        immune_since = 0;
                        setStatus(SANE);
                        randomizePeriods();
                    }
                }
            }
        }
        setStatus(status);
        move(1);
        distance--;
        if( distance <= 0) {
            randomizeDestination();
        }
    }
}
```