TP4 : Simulation de la propagation d'une maladie au sein d'une population

Jacques KOZIK, Arij HADDA et Killian GALFRE–VEYRET

Version du 06 avril 2024

Table des matières

1 Introduction

La modélisation de la propagation des maladies dans une population est un outil essentiel pour comprendre et gérer les épidémies.

Les modèles multi-agents offrent une approche puissante pour simuler les interactions individuelles au sein d'une communauté et étudier l'émergence de phénomènes collectifs tels que la propagation d'une maladie infectieuse. Dans cette étude, nous développons un modèle multi-agent simple pour simuler la propagation d'une maladie au sein d'une population, en mettant en œuvre des mécanismes tels que l'infection, l'exposition, la récupération et la perte d'immunité.

Le modèle repose sur une grille en deux dimensions où chaque individu est caractérisé par son statut de santé, représenté par les états Susceptible (S), Exposé (E), Infecté (I) et Rétabli (R), ainsi que par les durées de vie associées à chaque état. Nous utilisons un espace toroïdal pour éviter les artefacts de bordure et permettre une interaction fluide entre les individus.

L'évolution de la maladie est modélisée sur une échelle de temps discrète, où chaque pas de temps correspond à un jour. Les individus se déplacent aléatoirement à chaque pas de temps, simulant ainsi les interactions spatiales au sein de la population. L'infection se produit lorsque des individus susceptibles entrent en contact avec des individus infectés dans leur voisinage, avec une probabilité calculée en fonction du nombre d'infectieux présents.

Nous intégrons également des mécanismes de transition d'état asynchrones, où les individus passent de l'état exposé à l'état infecté après une période d'exposition, puis de l'état infecté à l'état rétabli après une période infectieuse. De plus, nous prenons en compte la perte d'immunité avec le temps, où les individus rétablis peuvent redevenir susceptibles après une période donnée.

En initialisant la simulation avec un nombre spécifique d'individus dans chaque état de santé et en définissant les paramètres de durée de vie de manière aléatoire, nous examinons comment différents scénarios affectent la dynamique de la maladie au fil du temps.

Dans cette étude, nous présentons le cadre général de notre modèle multi-agent de propagation de la maladie et discutons de son application potentielle pour comprendre et prévoir les épidémies dans des contextes réels.

2 Description du modèle

Pour mettre en œuvre cette simulation, nous avons opté pour une approche combinant des classes Java avec l'utilisation du générateur Mersenne Twister. Et pour le rendu, nous avons inscrits nos résultats dans des fichiers CSV puis avons, à l'aide d'un notebook Jupyter, généré des graphiques pour les illustrer.

2.1 Générateur Mersenne Twister (MTrandom)

Le générateur Mersenne Twister, largement utilisé en Java pour la génération de nombres aléatoires, est apprécié pour sa haute qualité et sa performance. En Java, il

est implémenté sous forme de classe, offrant une interface simple et efficace pour la génération de nombres aléatoires. Cette méthode de génération de nombres aléatoires est basée sur un algorithme développé par Matsumoto et Nishimura, qui produit des séquences de nombres pseudo-aléatoires présentant d'excellentes propriétés statistiques. En utilisant le générateur Mersenne Twister en Java, les développeurs peuvent facilement introduire des éléments de stochasticité dans leurs simulations, ce qui est crucial pour modéliser des phénomènes complexes tels que la propagation des maladies dans une population. La robustesse et la fiabilité de Mersenne Twister en font un choix populaire pour de nombreuses applications nécessitant une génération de nombres aléatoires de haute qualité en Java.

Nous avons également ajouter une méthode negExp dans la classe MTrandom, qui prend en paramètre un réel mean, et qui renvoie un réel entre 0 et mean.

```
  /*method negExp, qui genere un nombre entre 0 et inMean*/
  public double negExp(double inMean) {
      return -inMean * Math.log(1 - this.nextDouble());
   }
```

2.2 Implémentation de Classes

2.2.1 Humain

Pour commencer, nous avons décider de créer une classe Humain, avec comme attributs un statut (statut) sous forme de caractère, un entier temps (temps), un entier de durée de temps exposés (dE), un entier de durée de temps infectés (dI) et enfin un entier de durée de temps de récupération (dR).

On peut créer la classe avec le code suivant :

```
public class Humain {
   private char statut;
   private int temps;
   private int dE;
   private int dI;
   private int dR;
   private MTRandom random = new MTRandom();

   //Constructeur

   public Humain(char s){
      statutValide(s);
      this.statut=s;
      this.temps=0;
      this.generate_dE();
      this.generate_dI();
      this.generate_dR();
   }
```

```
    private void statutValide(char s){
        if ((s!='S')&&(s!='E')&&(s!='I')&&(s!='R')){
            throw new IllegalArgumentException("statut non
                valide");
        }
    }


    //Setteurs

    public void SetStatut(char s){
        statutValide(s);
        this.statut=s;
    }

    public void SetTemps(int n){
        this.temps = n;
    }

    public void generate_dE(){
        this.dE = (int) random.negExp(3);

    }

    public void generate_dI(){
        this.dI = (int) random.negExp(7);

    }

    public void generate_dR(){
        this.dR = (int) random.negExp(365);
    }

}
```

Les méthodes generate_dE, generate_dI et generate_dR permettent de générer des durées de temps exposés, de temps infectés et de récupération respectivement avec l'aide de la méthode negExp de la classe MTrandom.

Nous avons également des Getteurs pour récupérer la valeur des attributs privé :

```
    //Getteurs

    public char GetStatut(){
        return this.statut;
    }
    public int GetTemps(){
        return this.temps;
```

```
    }
    public int GetdE(){
        return this.dE;
    }
    public int GetdI(){
        return this.dI;
    }
    public int GetdR(){
        return this.dR;
    }
```

2.2.2 GrilleHumain

Suite à cela, nous avons opté pour la création d'une classe appelée 'GrilleHumain'. Son objectif est de générer des instances représentant une grille toroïdale, où chaque case peut contenir une liste d'humains. Plusieurs humains distincts peuvent se trouver sur une même case. On peut la créer avec le code suivant :

```
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.ArrayList;
import java.util.List;


public class grilleHumain {
    // Attributs
    private ArrayList<Humain>[][] grille;
    public static MTRandom random = new MTRandom();

    // Constructeur

    /* grilleHumain, methode de contruction de la classe
        grilleHumain qui creer un tableau a deux dimensions,
     * et qui place un tableau ou peuvent etre placer des objets
         de classe Humain dans chaque case. */

    @SuppressWarnings("unchecked")
    public grilleHumain(int lignes, int colonnes) {
        this.grille = new ArrayList[lignes][colonnes];
        for (int i = 0; i < lignes; i++) {
            initialiserLigne(i, colonnes);
        }
    }

    /* InitialiserLigne, methode utiliser dans le constructeur
        pour initialiser les lignes du tableau. */
```

```
private void initialiserLigne(int ligne, int colonnes) {
   for (int j = 0; j < colonnes; j++) {
      grille[ligne][j] = new ArrayList<Humain>();
   }
}
```

Nous avons ensuite créer deux méthodes permettant de remplir cette grile avec un nombre n d'individus. La première, addHumain() prend en paramètre un humain et le place de manière aléatoire sur une case de la grille. Pour cela, elle fait appel à la fonction negExp() qui génère un nombre compris entre 0 et la taille de la grille (300 dans notre cas) en se basant sur le générateur de nombre de Mersenne Twister (cf. section 2.1). Une seconde, addAllHumains(), qui se charge de créer les humains qui doivent être positionnés sur la grille puis qui fait appel à la fonction susmentionné pour le faire.

```
/* addHumain, méthode qui prend en paramètres un entier
    ligne, un entier colonne et un humain, et qui
 * l'ajoute àla case donnée.*/

public void addHumain(Humain humain) {
   int ligne = (int) random.negExp(grille.length);
   int colonne = (int) random.negExp(grille.length);
   grille[this.HorsTab(ligne,grille.length)][this.HorsTab(colonne,
       grille.length)].add(humain);
}

public void addAllHumains(int n, int infected){
   for(int i = 0 ; i < infected ; i++){
      Humain infect = new Humain('I');
      this.addHumain(infect);
   }
   for(int i = 0 ; i < (n - infected) ; i++){
      Humain h = new Humain('S');
      this.addHumain(h);
   }
}
```

Dans un troisième temps, nous avons créés les méthodes permettant de calculer le nombre d'humains inféctés par rapport à une case donnée. Il y en a 3, la première récupère une liste d'humain (il y en a une dans chaque case). La seconde, compte le nombre d'humain dont le statut et 'I' (pour Infected) dans la case et la troisième fait répéter ce comptage sur les 8 cases autour.

```
/* getHumain, méthode qui prend en paramètre une ligne et
    une colonne et qui renvoie le tableau d'Humain
 * situé dans cette case. */

public List<Humain> getHumains(int ligne, int colonne) {
   validerPosition(ligne, colonne);
```

```
    return grille[ligne][colonne];
}

/* calculInfected, méthode qui prend en paramètre un entier
    ligne et un entier colonne, et qui renvoie le nombre
 * d'humain infectés àcette case.*/
public int calculInfected(int ligne, int colonne){
    List<Humain> list = getHumains(ligne,colonne);
    int infected = 0;
    for (int i = 0; i < list.size(); i++){
        Humain humain = list.get(i);
        if (humain.GetStatut() == 'I'){
            infected+=1;
        }
    }
    return infected;
}

/* infectedAround, méthode qui prend en paramètre un entier
    ligne et un entier colonne, et qui renvoie le nombre
 * d'humain infectés àcette case, et les 8 cases autour.*/
public int infectedAround(int ligne, int colonne){
    int infected = 0;

    for (int i = ligne-1; i <= ligne+1; i++){
        for (int j = colonne-1; j <= colonne+1; j++){
            int ni = HorsTab(i, grille.length);
            int nj = HorsTab(j, grille.length);
            infected += calculInfected(HorsTab(i,
                grille.length),HorsTab(j, grille.length));
        }
    }
    return infected;
}

public double probability(Humain h, int ligne, int colonne){
    return 1-Math.exp(-0.5*infectedAround(ligne, colonne));
```

Puis ensuite les dernières méthodes créées servent à mettre en place la simulation :

checkEtat Regarde l'état d'un humain et met en place la conduite défini dans le sujet (eg. pour un infécté, on regarde d'abord si le temps depuis lequel il est dans cet état, est supérieur au temps pendant lequel il doit y rester - défini aléatoirement - puis si c'est le cas, alors on lui change son statut pour le suivant, sinon, on incrémente le temps passé dans ce statut de +1)

deplacementHumains Déplace tout les humain sur une autre case quelle qu'elle soit et de manière aléatoire.

simulation Lance la simulation et écrit au fur et à mesure, le nombre d'humain dans chaque état dans un fichier CSV. Pour cela, elle crée d'abord 19 980 hu-

mains dans l'état Susceptible et 20 dans l'état Infecté. Ensuite, elle lance une boucle de 730 itérations, qui correspond à 730 jours. A l'intérieur, pour chaque humain, on appel la fonction checkEtat() puis ensuite on incrémente en fonction de son nouvel état, les variables permettant de compter le nombre d'individu dans chaque état. Une fois fait pour tout les individus, on appel la fonction deplacementHumains() puis on inscrit les résultats des variables compteur dans le CSV. ET on recommence 729 fois !

```
public void checkEtat(Humain h, int ligne, int colonne){
   switch (h.GetStatut()) {
      case 'S':
         if(random.negExp(1)<this.probability(h, ligne,
             colonne)){
            h.SetStatut('E');
            h.SetTemps(0);
         }
         break;
      case 'E':
         if(h.GetTemps()>h.GetdE()){
            h.SetStatut('I');
            h.SetTemps(0);
         }
         else{
            h.SetTemps(h.GetTemps()+1);
         }
         break;
      case 'I':
         if(h.GetTemps()>h.GetdI()){
            h.SetStatut('R');
            h.SetTemps(0);
         }
         else{
            h.SetTemps(h.GetTemps()+1);
         }
         break;
      case 'R':
         if(h.GetTemps()>h.GetdR()){
            h.SetStatut('S');
            h.SetTemps(0);
         }
         else{
            h.SetTemps(h.GetTemps()+1);
         }
         break;
   }
}

private void deplacementHumains(){
   for (int i = 0; i < grille.length; i++) {
```

```
        for (int j = 0; j < grille[i].length; j++) {
            List<Humain> humains = new
                ArrayList<>(getHumains(i, j));
            for (Humain h : humains) {
                int newLigne = HorsTab((int)
                    random.negExp(300), grille.length);
                int newColonne = HorsTab((int)
                    random.negExp(300), grille.length);
                grille[HorsTab(newLigne,
                    grille.length)][HorsTab(newColonne,
                    grille.length)].add(h);
                grille[i][j].remove(h);
            }
        }
    }
}

public void simulation(int simu){
    /* Initialisation */
    for (int i = 0; i < 19980; i++) {
        Humain h = new Humain('S');
        addHumain(h);
    }

    for (int i = 0; i < 20; i++) {
        Humain infect = new Humain('I');
        addHumain(infect);
    }

    try (PrintWriter writer = new PrintWriter(new
        FileWriter("simulation_results_" + simu + ".csv",
        true))) {
        writer.println(20 + "," + 19980 + "," + 0 + "," + 0);
    }
    catch (IOException e) {
        System.out.println("Il y a une erreur, aucune é
            criture dans le fichier n'a été effectuée.");
        e.printStackTrace();
    }


    for(int k=0;k<730;k++){
        System.out.println("Jour "+k);
        int nombre_infected = 0;
        int nombre_sain = 0;
        int nombre_exposed = 0;
        int nombre_recovered = 0;

        for (int i = 0; i < 300; i++) {
            for (int j = 0; j < 300; j++) {
```

```
                List<Humain> humains = new
                    ArrayList<>(getHumains(i, j));
                for (Humain h : humains) {
                   checkEtat(h, i, j);
                   if(h.GetStatut()== 'I'){
                      nombre_infected+=1;
                   }
                   else if(h.GetStatut()== 'S'){
                      nombre_sain+=1;
                   }
                   else if(h.GetStatut()== 'E'){
                      nombre_exposed+=1;
                   }
                   else if(h.GetStatut()== 'R'){
                      nombre_recovered+=1;
                   }
                }
             }
          }
          this.deplacementHumains();
          try (PrintWriter writer = new PrintWriter(new
              FileWriter("simulation_results_" + simu + ".csv",
              true))) {
             writer.println(nombre_infected + "," + nombre_sain
                 + "," + nombre_exposed + "," +
                 nombre_recovered);
          }
          catch (IOException e) {
             System.out.println("Il y a une erreur, aucune é
                 criture dans le fichier n'a été effectuée.");
             e.printStackTrace();
          }
       }
    }
```

2.2.3 Main

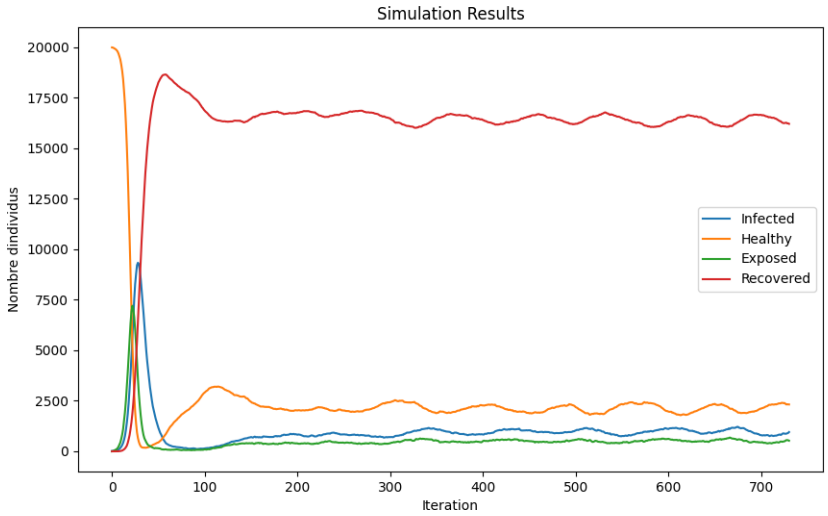
Cette classe, qui se veut la plus simple et courte possible, permet le lancement du programme. Pour cela, elle crée une méthode *main* qui, à travers une boucle for, éxecute 100 fois les 2 commandes suivante : création d'une grille vide puis lancement de la simulation par l'appel de la fonction simulation() qui gère toute la simulation (cf. section 2.2.2).

```
public class Main {
   public static MTRandom random = new MTRandom();
   public static void main(String[] args) {
      for(int i=1;i<=100;i++){
         grilleHumain grille = new grilleHumain(300, 300);
```

```
            grille.simulation(i);
        }
    }
}
```

3 Résultats

En réalisant la simulation 100 fois, on obtient toujours des résultats proche les uns des autres. Voici le graphe obtenu lors de notre 50-ième itération.



On constate - à chaque fois - que le pique d'exposé est atteint entre le 25-ième jour et le 30-ième. Le pique des infectés suit logiquement quelques jours après, autour du 30-ième jour. Puis celui des 'Recovered' encore après. Autour du 55-ième jour. A l'inverse, le nombre de 'Susceptible' chute en même temps que celui de 'Exposed' et 'Infected' augmente, soit autour du 30-ième jour. Il reste ensuite très faible puisque les personnes ayant le status 'Recovered' ne peuvent pas être réinfectés pendant un certain temps.

4 Conclusion

La modélisation de la propagation des maladies dans une population est cruciale pour comprendre les épidémies. A travers ce projet, nous avons développé un modèle multi-agent permettant de simuler la propagation d'une maladie au sein d'une population, en mettant en œuvre des mécanismes tels que l'exposition, l'infection, la récupération et la perte d'immunité. En utilisant une approche basée sur des classes Java et le générateur Mersenne Twister, nous avons pu créer un modèle robuste et flexible. Et à l'aide d'un notebook Jupyter nous avons pu réaliser des graphiques pour étudier les

résultats de nos simulations.

Notre étude a donc révélé des résultats cohérents, avec des simulations répétées montrant des schémas similaires dans la dynamique de la maladie. Le pic d'exposition est généralement observé entre le 25e et le 30e jour, suivi du pic des infectés quelques jours plus tard, et enfin du pic des personnes récupérées. De plus, nous avons constaté une diminution significative du nombre de personnes susceptibles après le pic d'infections, témoignant de l'acquisition d'une immunité partielle au sein de la population.

Ces résultats démontrent l'efficacité de notre modèle pour simuler la propagation des maladies et pour comprendre les tendances épidémiologiques. En combinant des techniques de modélisation multi-agent avec des outils de programmation robustes, notre approche offre un cadre précieux pour étudier et prévoir les épidémies dans des contextes réels.