yМинистерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение

высшего образования

**«КУБАНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»**

**(ФГБОУ ВО «КубГУ»)**

**Факультет компьютерных технологий и прикладной математики**

**Кафедра информационных технологий**

**ОТЧЁТ №6**

**Дисциплина: Многоагентное моделирование**

Работу выполнил: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_М. В. Сидоренко

Направление подготовки: 02.03.03 Математическое обеспечение и администрирование информационных систем

Преподаватель: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_А. А. Миков

Краснодар

2024

**Цель работы**

Составить программу, которая будет моделировать «Эпидемию»: на карте создаются здоровые агенты, через время некоторые из них заболевают, превращаются в зомби и начинают заражать других агентов.

**Описание задачи**

***Эпидемия***

**Контекст**

Есть n агентов, случайно расположенных в квадрате 100х100 (это значит, что у каждого i агента есть координаты позиции на плоскости (x\_i, y\_i)). Каждый агент имеет определённое состояние: здоровый, заражённый, зомби, выздоровевший.

У каждого агента есть некоторое направление и скорость движения (v\_xi, v\_yi), согласно которым он движется случайный период времени (t\_move), по истечению которого для него генерируется новое направление движения.

В начале моделирования все агенты на поле являются здоровыми. Спустя некоторый период длительности t\_init среди них случайным образом m агентов становятся заражёнными, и у них начинается инкубационный период. Во время этого периода скорость агента снижена по сравнению со стандартной.

По истечению инкубационного периода, заражённый агент становится зомби. Целью зомби является заражение здоровых агентов. В свою очередь у здоровых агентов целью является не быть заражёнными.

На каждой итерации цикла, зомби с 1% шансом может стать выздоровевшим. Выздоровевший агент не является целью для зомби, однако если он попадёт в радиус их действия, то с 25% шансом он обращается снова, миную стадию заражения.

Все агенты зрячие, причём видят они сектор окружности, расположенный согласно вектору их движения, имеющий угол alpha и радиус r. Соответственно они могут идентифицировать других агентов, если те попали в их сектор видимости, о других они не знают (ни их положения, ни их состояния).

**Характеристики состояний**

**Здоровый**

Скорость (сколько может пройти за итерацию): v\_h – при обычном движении; v\_h \* 1.25 – при бегстве от зомби.

Сектор видимости: угол – alpha (случайная для каждого агента величина в диапазоне от 90 до 150 градусов); радиус – r\_h.

Имеет 2 состояния: случайное блуждание в области; бегство от зомби.

Если агент видит зомби в своём секторе, то сектор делится надвое: если зомби в правом секторе, то его направление движения меняется влево так, чтобы указанный зомби (или несколько) больше не попадал в сектор его видимости; если зомби в левом секторе, то его направление движения меняется вправо так, чтобы указанный зомби (или несколько) больше не попадал в сектор его видимости; если зомби есть в обоих секторах, то агент поворачивается на 180 градусов и движется в обратном направлении.

Не избегает заражённых.

**Заражённый**

Скорость снижена по сравнению со стандартной на 10%. Сектор видимости аналогичен здоровому. Не имеет цели и не является целью для зомби.

Инкубационный период: случайное значение из диапазона [t\_inc\_min, t\_inc\_max] (согласно равномерному распределению).

**Зомби**

Скорость снижена по сравнению со стандартной на 15%.

Сектор видимости: угол – alpha \* 0.75 (то есть уменьшается на 25% по сравнению со стандартной видимостью конкретного агента); радиус – r \* 1.1 (радиус обзора увеличен на 10% по сравнению со стандартной видимостью конкретного агента).

Сектор действия (сектор, при попадании в которой здоровый агент становится заражённым), по всем характеристикам меньше сектора видимости на 7%.

Когда зомби видит здорового агента, он начинает двигаться в его сторону. При попадании в зону видимости нескольких здоровых агентов, движение осуществляется к ближайшему из них.

На каждой итерации с небольшим шансом может стать выздоровевшим.

**Выздоровевший**

Имеет стандартные характеристики (как у здорового). Не является целью для зомби, не убегает от них. В случае, если агент попадает в зону действия зомби, он с 25% вероятностью снова становится зомби.

**Процесс моделирования**

В нулевой момент времени в области равномерно распределяются агенты, все они имеют статус здоровый. Они передвигаются случайным образом, с некоторой заданной скоростью в случайном направлении (на каждом шаге к их позиции добавляется определённый вектор, так, чтобы их перемещение было по направлению движения, но величина смещения была не больше значения скорости). Если при смещении на следующем шаге, позиция агента окажется за пределами области, то он должен изменить своё направление движения согласно принципу отражения. Движение осуществляется на протяжении t\_move итераций (случайное значение из диапазона [t\_move\_min, t\_move\_max]), после чего генерируется новое направление и время движения.

По истечении времени t\_init m агентов становятся заражёнными, для каждого агента генерируется своё время инкубационного периода из диапазона [t\_inc\_min, t\_inc\_max] (согласно равномерному распределению).

Когда инкубационный период у агента заканчивается, он становится зомби, его целью становится преследование и заражение здоровых, здоровые же, в свою очередь, могут его идентифицировать и спасаться.

Моделирование заканчивается, когда в области не остаётся агентов в состоянии здоровый, или по истечению времени моделирования T.

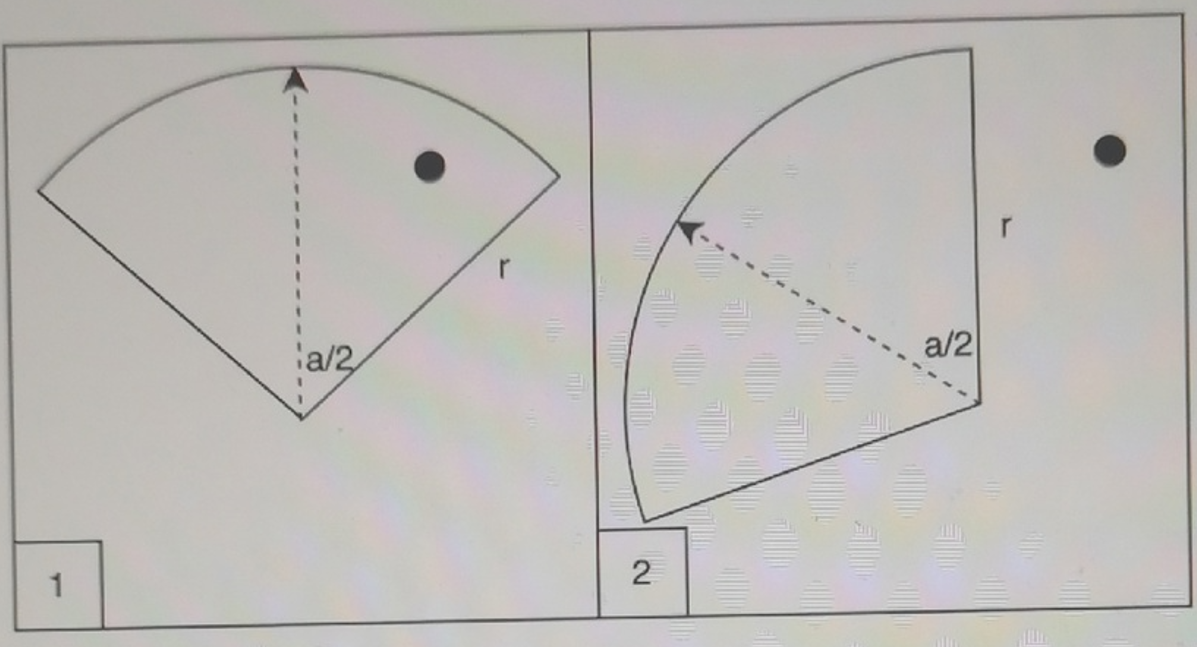
**Задача**

Для нескольких разных наборов значений n и m (не менее 4 значений для каждой характеристики), составить таблицу, в которой будет указано среднее время обращения в зомби всей популяции (в области не остаётся здоровых агентов). Для получения среднего времени для каждой конфигурации провести 1000 экспериментов.

Рекомендации и ремарки

1. Для реализации изменяемого поведения агентов можно использовать паттерн проектирования «Машина состояний».

2. Пример изменения направления движения у здоровых агентов при попадании зомби в правый сектор видимости:



3. Функции для работы с геометрией и математикой в целом можно вынести в отдельный класс/файл. Разумно добавить набор тестов для каждой функции, чтобы быстро убедиться в правильности её работы.

**Описание решения**

1. **Класс Agent**:
   * Представляет агента в симуляции, имея координаты (x, y), скорость, направление и текущее состояние (здоровый, зараженный, зомби или выздоровевший).
   * Обновляет свое состояние и перемещение на каждом шаге симуляции.
   * Реализует логику движения, включая случайное блуждание и проверку границ области (1000x1000).
2. **Классы состояний (State)**:
   * Каждый агент может находиться в одном из нескольких состояний, каждое из которых реализует свою логику поведения:
     + **HealthyState**: Здоровый агент, который может бежать от зомби, если видит их в своем секторе видимости.
     + **InfectedState**: Зараженный агент, который имеет сниженное движение и инкубационный период перед превращением в зомби.
     + **ZombieState**: Зомби, который преследует здоровых агентов и может заразить их, а также с небольшим шансом может стать выздоровевшим.
     + **RecoveredState**: Выздоровевший агент, который больше не является целью для зомби, но может снова стать зомби при попадании в зону действия.
3. **Класс Simulation**:
   * Управляет всей симуляцией, включая создание агентов, их состояние и взаимодействия.
   * Инициализирует агентов в здоровом состоянии и через определенное время (t\_init) случайным образом заражает m агентов.
   * Отслеживает состояние агентов на каждом шаге симуляции и завершает моделирование, когда не остается здоровых агентов или истекает заданное время (T).
4. **Класс SimulationPanel**:
   * Отвечает за визуализацию симуляции, отрисовывая агентов на экране в зависимости от их состояния.
5. **Класс Main**:
   * Запускает симуляцию, создавая окно приложения и обновляя состояние симуляции с помощью таймера, который вызывает обновление и перерисовку панели.

**Логика работы симуляции:**

* В начале симуляции все агенты случайным образом размещаются на поле и начинают двигаться в случайных направлениях.
* По истечении времени t\_init, среди агентов случайным образом выбираются m зараженных, которые получают инкубационный период.
* Зараженные агенты становятся зомби по окончании инкубационного периода и начинают преследовать здоровых агентов.
* Здоровые агенты, видя зомби в своем секторе видимости, изменяют направление движения, чтобы избежать заражения.
* Зомби могут заразить здоровых агентов, а выздоровевшие агенты могут снова стать зомби с определенной вероятностью.
* Симуляция продолжается до тех пор, пока не останется здоровых агентов или не истечет общее время моделирования.

**Итоги:**

Решение эффективно моделирует динамику взаимодействия между различными состояниями агентов, используя принципы объектно-ориентированного программирования и паттерн "Машина состояний". Это позволяет легко расширять функциональность и добавлять новые состояния или правила поведения агентов в будущем.

**Код программы**

package org.example;  
  
import java.util.Random;  
  
class Agent {  
 private double x;  
 private double y;  
 private double baseSpeed;  
 private double speed;  
 private double direction;  
 private State state;  
 private Simulation simulation;  
  
 public Agent(double x, double y, Class<? extends State> stateClass, double baseSpeed, Simulation simulation) {  
 this.x = x;  
 this.y = y;  
 this.baseSpeed = baseSpeed;  
 this.speed = baseSpeed;  
 this.simulation = simulation;  
 this.direction = new Random().nextDouble() \* 2 \* Math.*PI*;  
 try {  
 this.state = stateClass.getConstructor(Agent.class).newInstance(this);  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
 }  
  
 public void update() {  
 state.update();  
 move();  
 }  
  
 public void move() {  
 if (state instanceof HealthyState || state instanceof InfectedState) {  
 moveRandomly();  
 }  
 }  
  
 public void moveRandomly() {  
 direction += (new Random().nextDouble() \* 0.2 - 0.1);  
 double dx = Math.*cos*(direction) \* speed;  
 double dy = Math.*sin*(direction) \* speed;  
 x += dx;  
 y += dy;  
 checkBoundaries();  
 }  
  
 public void checkBoundaries() {  
 if (x < 0 || x > 1000) {  
 direction = Math.*PI* - direction;  
 }  
 if (y < 0 || y > 1000) {  
 direction = -direction;  
 }  
 x = Math.*max*(0, Math.*min*(1000, x));  
 y = Math.*max*(0, Math.*min*(1000, y));  
 }  
  
 // Getters and setters  
 public double getX() { return x; }  
 public void setX(double x) { this.x = x; }  
 public double getY() { return y; }  
 public void setY(double y) { this.y = y; }  
 public double getBaseSpeed() { return baseSpeed; }  
 public double getSpeed() { return speed; }  
 public void setSpeed(double speed) { this.speed = speed; }  
 public double getDirection() { return direction; }  
 public void setDirection(double direction) { this.direction = direction; }  
 public State getState() { return state; }  
 public void setState(State state) { this.state = state; }  
 public Simulation getSimulation() { return simulation; }  
 public double distanceTo(Agent other) {  
 return Math.*sqrt*(Math.*pow*(x - other.getX(), 2) + Math.*pow*(y - other.getY(), 2));  
 }  
}

package org.example;  
  
abstract class State {  
 protected Agent agent;  
  
 public State(Agent agent) {  
 this.agent = agent;  
 }  
  
 public abstract void update();  
}

package org.example;  
  
class HealthyState extends State {  
 public HealthyState(Agent agent) {  
 super(agent);  
 agent.setSpeed(agent.getBaseSpeed());  
 }  
  
 @Override  
 public void update() {  
 for (Agent other : agent.getSimulation().getAgents()) {  
 if (other.getState() instanceof ZombieState && other.distanceTo(agent) < agent.getSimulation().getInfectionRadius()) {  
 agent.setSpeed(agent.getBaseSpeed() \* 1.25);  
 return;  
 }  
 }  
 agent.setSpeed(agent.getBaseSpeed());  
 }  
}

package org.example;  
  
class InfectedState extends State {  
 private int incubationPeriod;  
  
 public InfectedState(Agent agent, int incubationPeriod) {  
 super(agent);  
 this.incubationPeriod = incubationPeriod;  
 agent.setSpeed(agent.getBaseSpeed() \* 0.95);  
 }  
  
 @Override  
 public void update() {  
 incubationPeriod--;  
 if (incubationPeriod <= 0) {  
 agent.setState(new ZombieState(agent));  
 }  
 }  
}

package org.example;  
  
import java.util.Random;  
  
class ZombieState extends State {  
 private int viewRadius = 15;  
 private double viewAngle = Math.*toRadians*(90);  
  
 public ZombieState(Agent agent) {  
 super(agent);  
 agent.setSpeed(agent.getBaseSpeed() \* 0.9);  
 }  
  
 @Override  
 public void update() {  
 if (new Random().nextDouble() < 0.001) {  
 agent.setState(new RecoveredState(agent));  
 return;  
 }  
  
 Agent target = findVisibleHealthyAgent();  
 if (target != null) {  
 moveToward(target);  
 if (agent.distanceTo(target) < agent.getSimulation().getInfectionRadius()) {  
 int incubationPeriod = new Random().nextInt(agent.getSimulation().getTIncMax() - agent.getSimulation().getTIncMin() + 1) + agent.getSimulation().getTIncMin();  
 target.setState(new InfectedState(target, incubationPeriod));  
 }  
 } else {  
 agent.moveRandomly();  
 }  
 }  
  
 private Agent findVisibleHealthyAgent() {  
 Agent closestAgent = null;  
 double minDistance = Double.*MAX\_VALUE*;  
  
 for (Agent other : agent.getSimulation().getAgents()) {  
 if (other.getState() instanceof HealthyState) {  
 double distance = agent.distanceTo(other);  
 if (distance < viewRadius && inViewAngle(other)) {  
 if (distance < minDistance) {  
 minDistance = distance;  
 closestAgent = other;  
 }  
 }  
 }  
 }  
 return closestAgent;  
 }  
  
 private boolean inViewAngle(Agent otherAgent) {  
 double dx = otherAgent.getX() - agent.getX();  
 double dy = otherAgent.getY() - agent.getY();  
 double angleToAgent = Math.*atan2*(dy, dx);  
 double relativeAngle = (angleToAgent - agent.getDirection()) % (2 \* Math.*PI*);  
 return Math.*abs*(relativeAngle) < viewAngle / 2;  
 }  
  
 private void moveToward(Agent targetAgent) {  
 double dx = targetAgent.getX() - agent.getX();  
 double dy = targetAgent.getY() - agent.getY();  
 double distance = Math.*sqrt*(dx \* dx + dy \* dy);  
 if (distance > 0) {  
 agent.setX(agent.getX() + (dx / distance) \* agent.getSpeed());  
 agent.setY(agent.getY() + (dy / distance) \* agent.getSpeed());  
 agent.setDirection(Math.*atan2*(dy, dx));  
 }  
 }  
}

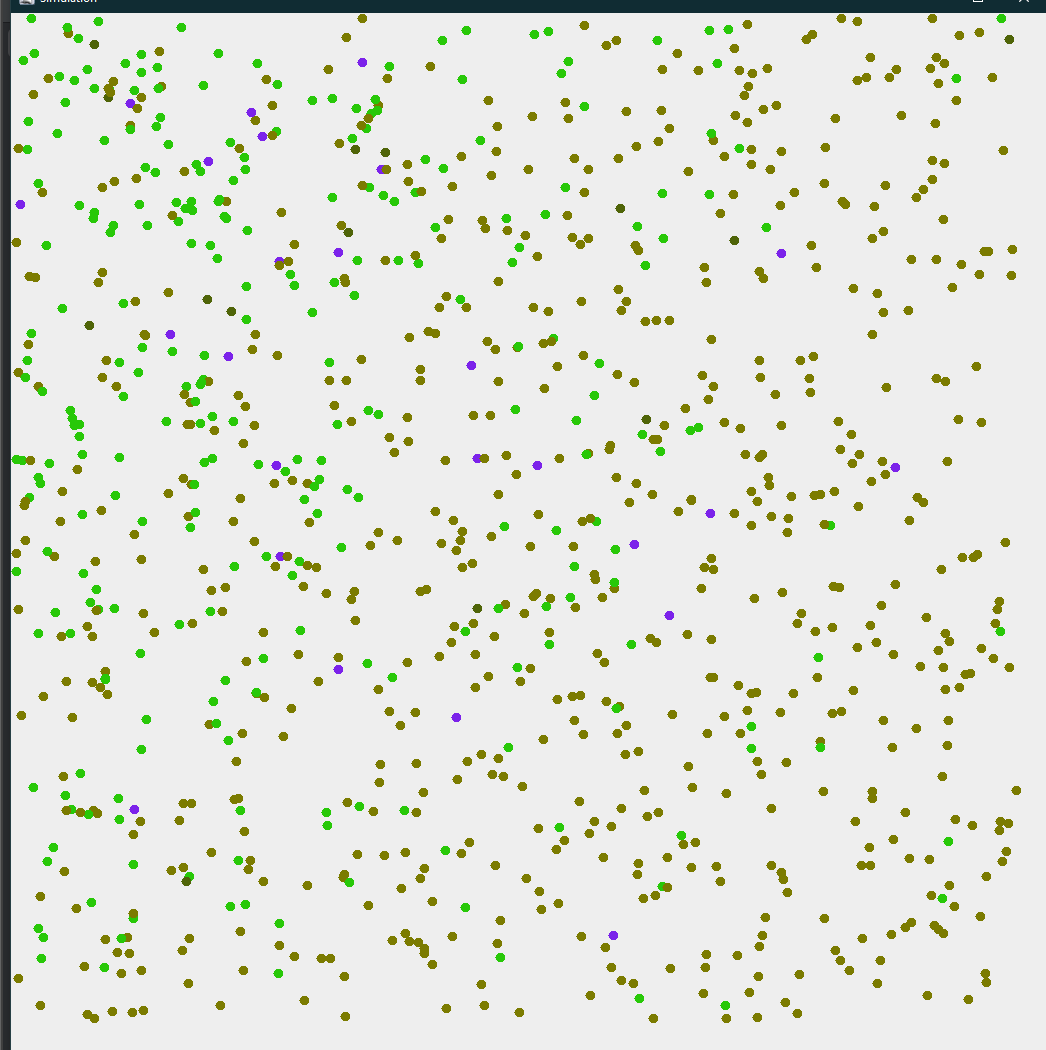
package org.example;  
  
import java.util.Random;  
  
class RecoveredState extends State {  
 public RecoveredState(Agent agent) {  
 super(agent);  
 agent.setSpeed(agent.getBaseSpeed());  
 }  
  
 @Override  
 public void update() {  
 agent.moveRandomly();  
 for (Agent other : agent.getSimulation().getAgents()) {  
 if (other.getState() instanceof ZombieState && other.distanceTo(agent) < agent.getSimulation().getInfectionRadius()) {  
 if (new Random().nextDouble() < 0.64) {  
 agent.setState(new ZombieState(agent));  
 break;  
 }  
 }  
 }  
 }  
}

package org.example;  
  
import java.util.ArrayList;  
import java.util.List;  
import java.util.Random;  
  
class Simulation {  
 private int n;  
 private int m;  
 private int tInit;  
 private int tIncMin;  
 private int tIncMax;  
 private int T;  
 private double infectionRadius;  
 private List<Agent> agents;  
 private int time;  
  
 public Simulation(int n, int m, int tInit, int tIncMin, int tIncMax, int T, double infectionRadius) {  
 this.n = n;  
 this.m = m;  
 this.tInit = tInit;  
 this.tIncMin = tIncMin;  
 this.tIncMax = tIncMax;  
 this.T = T;  
 this.infectionRadius = infectionRadius;  
 this.agents = new ArrayList<>();  
 this.time = 0;  
 initAgents();  
 }  
  
 private void initAgents() {  
 for (int i = 0; i < n; i++) {  
 double x = new Random().nextDouble() \* 1000;  
 double y = new Random().nextDouble() \* 1000;  
 agents.add(new Agent(x, y, HealthyState.class, 5.0, this));  
 }  
 }  
  
 public void infectInitialAgents() {  
 List<Agent> infectedAgents = new ArrayList<>(agents);  
 java.util.Collections.*shuffle*(infectedAgents);  
 for (int i = 0; i < m; i++) {  
 Agent agent = infectedAgents.get(i);  
 int incubationPeriod = new Random().nextInt(tIncMax - tIncMin + 1) + tIncMin;  
 agent.setState(new InfectedState(agent, incubationPeriod));  
 }  
 }  
  
 public void runStep() {  
 if (time == tInit) {  
 infectInitialAgents();  
 }  
 for (Agent agent : agents) {  
 agent.update();  
 }  
 time++;  
 }  
  
 public int[] countStates() {  
 int healthyCount = 0;  
 int infectedCount = 0;  
 int zombieCount = 0;  
 int recoveredCount = 0;  
  
 for (Agent agent : agents) {  
 if (agent.getState() instanceof HealthyState) healthyCount++;  
 if (agent.getState() instanceof InfectedState) infectedCount++;  
 if (agent.getState() instanceof ZombieState) zombieCount++;  
 if (agent.getState() instanceof RecoveredState) recoveredCount++;  
 }  
  
 return new int[]{healthyCount, infectedCount, zombieCount, recoveredCount};  
 }  
  
 public boolean isFinished() {  
 int[] counts = countStates();  
 int healthyCount = counts[0];  
 int infectedCount = counts[1];  
 int zombieCount = counts[2];  
 int recoveredCount = counts[3];  
  
 if (time >= T) return true;  
 if (zombieCount == n || recoveredCount == n) return true;  
 if (infectedCount > 0) return false;  
  
 for (Agent agent : agents) {  
 if (agent.getState() instanceof InfectedState) {  
 return false;  
 }  
 }  
  
 return zombieCount == n || recoveredCount == n;  
 }  
  
 // Getters  
 public List<Agent> getAgents() { return agents; }  
 public double getInfectionRadius() { return infectionRadius; }  
 public int getTIncMin() { return tIncMin; }  
 public int getTIncMax() { return tIncMax; }  
}

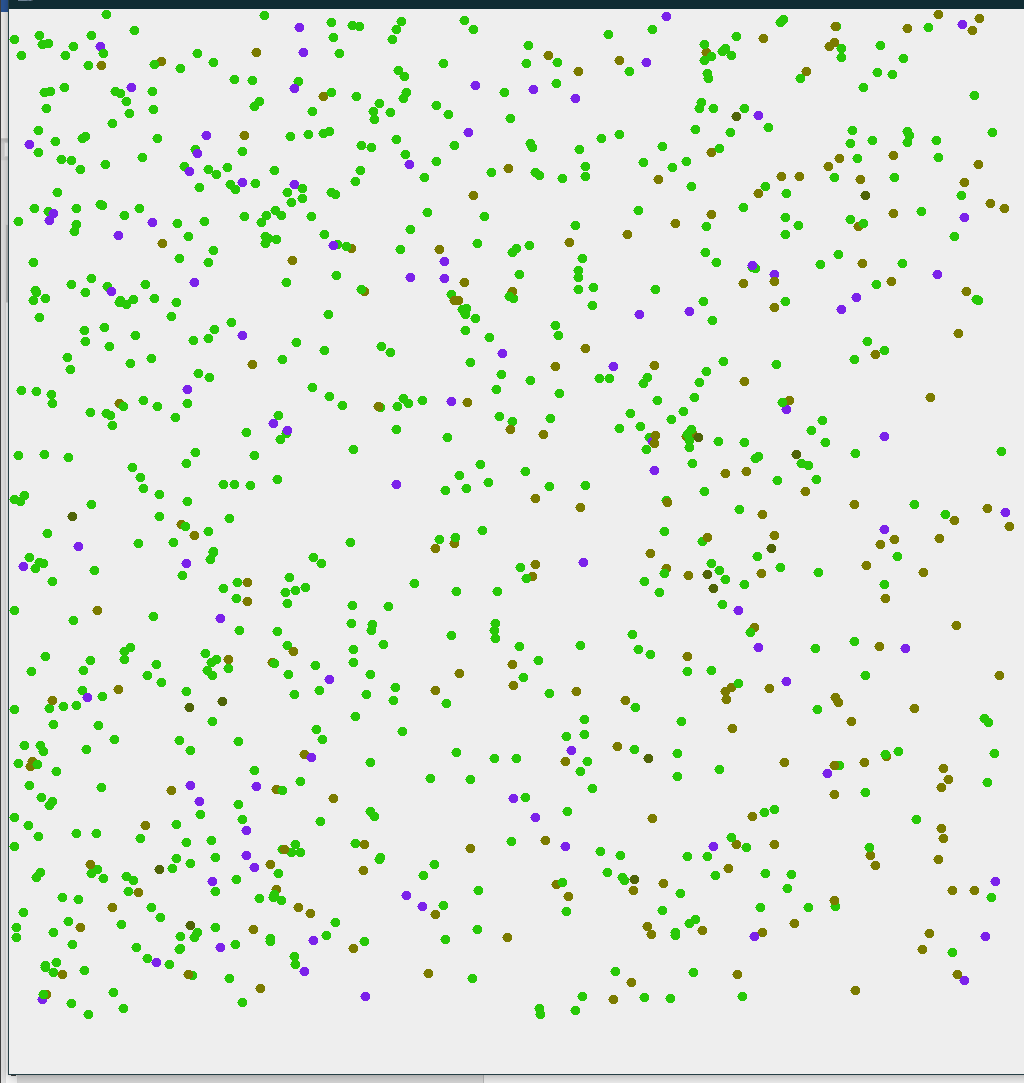
package org.example;  
  
import javax.swing.\*;  
import java.awt.\*;  
  
class SimulationPanel extends JPanel {  
 private Simulation simulation;  
  
 public SimulationPanel(Simulation simulation) {  
 this.simulation = simulation;  
 }  
  
 @Override  
 protected void paintComponent(Graphics g) {  
 super.paintComponent(g);  
 Graphics2D g2d = (Graphics2D) g;  
  
 for (Agent agent : simulation.getAgents()) {  
 if (agent.getState() instanceof HealthyState) {  
 g2d.setColor(new Color(123, 123,0));  
 } else if (agent.getState() instanceof InfectedState) {  
 g2d.setColor(new Color(79, 100,8));  
 } else if (agent.getState() instanceof ZombieState) {  
 g2d.setColor( new Color(40, 200,8));  
 } else if (agent.getState() instanceof RecoveredState) {  
 g2d.setColor(new Color(123, 34,234));  
 }  
  
 g2d.fillOval((int) agent.getX(), (int) agent.getY(), 10, 10);  
 }  
 }  
}

package org.example;  
  
import javax.swing.\*;  
import java.awt.event.ActionEvent;  
import java.awt.event.ActionListener;  
  
  
public class Main {  
 public static void main(String[] args) {  
 Simulation simulation = new Simulation(1000, 5, 10, 5, 10, 1000, 1.5);  
 SimulationPanel panel = new SimulationPanel(simulation);  
 JFrame frame = new JFrame("Simulation");  
 frame.setDefaultCloseOperation(JFrame.*EXIT\_ON\_CLOSE*);  
 frame.setSize(1000, 1000);  
 frame.add(panel);  
 frame.setVisible(true);  
  
 Timer timer = new Timer(100, new ActionListener() {  
 @Override  
 public void actionPerformed(ActionEvent e) {  
 if (!simulation.isFinished()) {  
 simulation.runStep();  
 panel.repaint();  
 } else {  
 ((Timer) e.getSource()).stop();  
 }  
 }  
 });  
 timer.start();  
 }  
}

**Примеры вывода**



Болотные – здоровые люди, зелёные – зомби, фиолетовые – выздоровевшие.



**Результирующая таблица**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Кол-во агентов и заражённых | 1000, 5 | 1000, 50 | 10000, 50 | 100, 25 |
| Среднее время окончания симуляции | 1000 | 1000 | 570 | 1000 |

**Вывод**

Распространение эпидемии происходит быстрее при большем колличестве агентов, что приводик к их более часттым контактам друг с другом.