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 №5**

**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

**Описание задачи**

**Динамическая балансировка нагрузки агентов**

**Модель нагрузки**

Имеется приложение, первоначально состоящее из "m" программных модулей, связанных между собой по управлению.

При выполнении приложения каждому модулю соответствует логический процесс (LP) на компьютере.

Логические процессы работают как сопрограммы - параллельно-последовательно.

Каждый LP\_i стартует в момент времени t\_i, последовательно выполняет инструкции модуля.

LP является нагрузкой для вычислительного ресурса (компьютера).

Величину нагрузки будем задавать вещественным числом.

Работа приложения заканчивается, когда заканчивает работу последний LP.

Структура приложения задана ориентированным нагруженным графом, в котором множество вершин соответствует множеству модулей.

Дуга в графе, идущая из вершины "a" в вершину "b", означает возможную передачу управления от модуля "a" модулю "b".

Часть вершин (множество S) имеет степень по входу, равную нулю.

Такие вершины являются начальными.

Часть вершин имеет степень по выходу, равную нулю. Каждой вершине приписана величина нагрузки.

**Модель многоагентной системы (МАС)**

Имеется "n" агентов.

Агенты обладают вычислительными ресурсами и взаимодействуют со средой - нагрузкой.

Агент может наблюдать среду, т.е. имеющиеся модули и их характеристики.

Агент может выполнять действия: превращать модуль в логический процесс на своём вычислительном ресурсе (и исполнять его - начинать и завершать); передавать модуль другому агенту; получать модуль от другого агента.

В любой момент времени агент может исполнять не более одного процесса.

Многоагентная система обладает структурой, изображаемой неориентированным графом: вершины соответствуют агентам; множество рёбер определяет пары агентов, имеющие возможность взаимодействовать между собой.

Распределение нагрузки - это отображение графа приложения на граф многоагентной системы, при котором один или несколько модулей передаётся одному агенту и дуги между модулями отображаются на ребра графа МАС.

Исполнение приложения многоагентной системой - процесс, начинающийся стартом в момент времени t = 0 логических процессов, соответствующих модулям из множества S, и заканчивающийся завершением (неодновременным) логических процессов, соответствующих вершинам графа нагрузки со степенью 0 по выходу.

Промежуточные LP стартуют сразу после завершения тех LP, с которыми их вершины связаны дугами (в отношении "предыдущий - последующий"), либо после назначения их на узел МАС.

Исполнение приложения начинается после завершения начального Распределения нагрузки.

Во время исполнения любого LP с вероятностью p = 0.05 может произойти отказ оборудования на узле, что потребует переноса его на другой узел (вычислять узел придётся с заново).

Динамическое перераспределение нагрузки - процесс, протекающий параллельно процессу исполнения, начинающийся после принятия одним из агентов решения о необходимости перераспределения нагрузки, и заканчивающийся до завершения приложения.

Динамическое перераспределение нагрузки состоит из последовательных и, возможно, параллельных (в разных локальных частях графа МАС) действий по изменению отображения на граф МАС.

Перераспределение возможно только в момент завершения какого-либо LP.

Цель распределения и динамического перераспределения нагрузки - уменьшение (если возможно - минимизация) времени работы приложения

**Задание**

1. Разработать алгоритм распределения и динамического перераспределения нагрузки в многоагентной системе. Входными (изменяемыми) данными являются граф нагрузки и граф МАС.

2. Реализовать модель нагрузки, модель многоагентной системы.

3. Реализовать алгоритм распределения и динамического перераспределения нагрузки в многоагентной системе для совместной работы с моделями нагрузки в МАС.

4. Продемонстрировать работу алгоритма на 3-4 тестовых примерах с различными моделями нагрузки и МАС (графы нагрузки и МАС и поэтапное изменение нагрузки агентов модулями приложения). Для каждого теста определить общее время выполнения приложения.

**Замечания**

Агент передаёт модуль на исполнение другому агенту (связанному) в том случае, если у текущего агента произошёл сбой (p = 0.05)

Если произошёл сбой - агент перестаёт работать до конца выполнения всей программы

Граф модели нагрузки и модели многоагентной системы можно задать в виде набора правил для объектов, т.е. при создании агента 1 можно,

например, в качестве его соседей указать агента 2, агенту 2 - агентов 1, 3, 5, и т.д.

Программу не нужно делать асинхронной

**Описание решения**

1. **Инициализация графов агентов и модулей**
   * Создаются классы GraphOfAgents и GraphOfModules для представления графов агентов и модулей, соответственно.
   * Методы createGraphOfAgents и createGraphOfModules инициализируют эти графы с заданным количеством агентов и модулей, а также их параметрами.
2. **Формирование зависимостей между модулями**
   * В методах createModulesDependenciesV1 и createModulesDependenciesV2 определяются зависимости между модулями. Эти зависимости означают, что один модуль должен быть завершен до начала выполнения другого.
3. **Агент и модуль**
   * Agent представляет агента с уникальным идентификатором, списком соседей, и текущим обрабатываемым модулем.
   * Module представляет модуль с уникальным идентификатором, начальной и текущей нагрузкой, списком зависимых модулей, текущим агентом и статусом завершенности.
4. **Назначение и выполнение модулей агентами**
   * Агенты могут пытаться взять модуль для выполнения, если тот доступен и агент не сломался.
   * Каждый шаг выполнения агенты уменьшают нагрузку текущего модуля и проверяют, завершен ли модуль.
   * Модули могут восстанавливаться, если агент сломался.
5. **Обработка поломок агентов**
   * В методе checkBreakdown агенты проверяют вероятность поломки.
   * В случае поломки агенты передают свои текущие модули соседям.
   * Если модуль был передан соседу, то продолжение выполнения происходит на новом агенте.
6. **Проверка всех агентов на состояние**
   * Метод checkAgentsBreakdown проверяет, сломались ли все агенты. Если да, выполнение программы невозможно.
7. **Основной метод**
   * В основном методе создаются графы агентов и модулей, назначаются

**Код программы**

package org.example;  
  
import java.util.ArrayList;  
import java.util.List;  
import java.util.Random;  
  
public class Agent {  
 private int id;  
 private List<Agent> neighbors;  
 private Module curModule;  
 private boolean execution;  
 private float amountOfExecution;  
 private static final float *BREAKDOWN\_PROBABILITY* = 0.05f;  
  
 public Agent(int id, float p) {  
 this.id = id;  
 this.neighbors = new ArrayList<>();  
 this.curModule = null;  
 this.execution = true;  
 this.amountOfExecution = p;  
 }  
  
 public void attemptGetModule(Module module) {  
 if (execution) {  
 if (curModule == null) {  
 if (module.checkAvailability()) {  
 curModule = module;  
 System.*out*.println("Агент id = " + id + " взял модуль id = " + curModule.getId() + " на выполнение");  
 module.appointmentAgent(this);  
 } else {  
 System.*out*.println("Агент id = " + id + " не может взять модуль id = " + module.getId() + " на выполнение");  
 }  
 } else {  
 System.*out*.println("Агент id = " + id + " не может взять модуль id = " + module.getId() + ", т.к. у агента уже есть модуль id = " + curModule.getId());  
 }  
 } else {  
 System.*out*.println("Агент id = " + id + " не может взять модуль id = " + module.getId() + ", т.к. агент сломался");  
 }  
 }  
  
 public void executionStep() {  
 if (execution) {  
 if (curModule != null) {  
 if (!curModule.checkExecution()) {  
 curModule.reduceLoad(amountOfExecution);  
 System.*out*.println("Агент id = " + id + " уменьшил нагрузку своего модуля (id = " + curModule.getId() + ") на " + amountOfExecution);  
 if (curModule.checkExecution()) {  
 System.*out*.println("Агент id = " + id + " выполнил модуль id = " + curModule.getId());  
 curModule = null;  
 }  
 } else {  
 System.*out*.println("Агент id = " + id + " выполнил модуль id = " + curModule.getId());  
 curModule = null;  
 }  
 } else {  
 System.*out*.println("Агент id = " + id + " не может выполнять модуль, т.к. агент не имеет модуля");  
 }  
 } else {  
 System.*out*.println("Агент id = " + id + " не может выполнять модуль, т.к. агент сломался");  
 }  
 }  
  
 public void checkBreakdown() {  
 Random rnd = new Random();  
 int b = rnd.nextInt(100) + 1; // Random number between 1 and 100  
  
 if (b <= 5) {  
 if (execution) {  
 if (curModule != null) {  
 Module module = null;  
 if (curModule.loadRecovery()) {  
 execution = false;  
 module = curModule;  
 curModule = null;  
  
 if (!neighbors.isEmpty()) {  
 for (Agent neighbor : neighbors) {  
 if (neighbor.execution && neighbor.curModule == null) {  
 neighbor.curModule = module;  
 System.*out*.println("Агент id = " + id + " сломался и передал свой модуль id = " + module.getId() + " агенту-соседу id = " + neighbor.id);  
 module.appointmentAgent(neighbor);  
 return;  
 } else {  
 System.*out*.println("Агент id = " + id + " сломался, и не может передать свой модуль id = " + module.getId() + " агенту-соседу id = " + neighbor.id);  
 }  
 }  
 } else {  
 System.*out*.println("Агент id = " + id + " не имеет соседей... ОШИБКА");  
 throw new RuntimeException("Agent has no neighbors");  
 }  
 } else {  
 System.*out*.println("Агент id = " + id + " сломался, а его модуль id = " + module.getId() + " не восстановил нагрузку (0\_0)");  
 }  
 } else {  
 System.*out*.println("Агент id = " + id + " не может сломаться без модуля... ОШИБКА");  
 throw new RuntimeException("Agent cannot break down without a module");  
 }  
 } else {  
 System.*out*.println("Агент id = " + id + " не может сломаться второй раз... ОШИБКА");  
 throw new RuntimeException("Agent cannot break down again");  
 }  
 } else {  
 executionStep();  
 }  
 }  
  
 public void addNeighbor(Agent neighbor) {  
 this.neighbors.add(neighbor);  
 }  
  
 // Getters  
 public int getId() {  
 return id;  
 }  
  
 public List<Agent> getNeighbors() {  
 return neighbors;  
 }  
  
 public boolean isExecution() {  
 return execution;  
 }  
  
 public Module getCurModule() {  
 return curModule;  
 }  
}

package org.example;  
  
import java.util.ArrayList;  
import java.util.List;  
import java.util.Random;  
  
public class Module {  
 private int id;  
 private float load;  
 private float curLoad;  
 private List<Module> dependencies;  
 private boolean availability;  
 private Agent curAgent;  
 private boolean completed;  
  
 public Module(int id, float a, float b) {  
 this.id = id;  
 this.load = Math.*round*(new Random().nextFloat() \* (b - a) + a \* 10.0f) / 10.0f; // Generate random load in range [a, b]  
 this.curLoad = this.load;  
 this.dependencies = new ArrayList<>();  
 this.availability = false;  
 this.curAgent = null;  
 this.completed = false;  
 }  
  
 public boolean checkAvailability() {  
 if (!completed) {  
 if (curAgent == null) {  
 if (dependencies.isEmpty()) {  
 availability = true;  
 System.*out*.println("Модуль id = " + id + " можно взять на выполнение (стартовый)");  
 return true;  
 } else {  
 for (Module dep : dependencies) {  
 if (!dep.isCompleted()) {  
 System.*out*.println("Модуль id = " + id + " недоступен для выполнения пока не будут выполнены все его зависимости");  
 return false;  
 }  
 }  
 availability = true;  
 System.*out*.println("Модуль id = " + id + " можно взять на выполнение (все зависимости выполнены)");  
 return true;  
 }  
 } else {  
 System.*out*.println("Модуль id = " + id + " занят агентом id = " + curAgent.getId());  
 return false;  
 }  
 } else {  
 System.*out*.println("Модуль id = " + id + " уже выполнен");  
 return false;  
 }  
 }  
  
 public void appointmentAgent(Agent agent) {  
 if (!completed) {  
 if (curAgent == null) {  
 if (availability) {  
 curAgent = agent;  
 System.*out*.println("Модулю id = " + id + " был назначен агент id = " + curAgent.getId());  
 } else {  
 System.*out*.println("Модулю id = " + id + " нельзя назначить агента id = " + agent.getId() + ", т.к. модуль недоступен для выполнения");  
 }  
 } else {  
 System.*out*.println("Модулю id = " + id + " нельзя назначить агента id = " + agent.getId() + ", т.к. модуль занят агентом id = " + curAgent.getId());  
 }  
 } else {  
 System.*out*.println("Модулю id = " + id + " нельзя назначить агента id = " + agent.getId() + ", т.к. модуль выполнен");  
 }  
 }  
  
 public boolean checkExecution() {  
 if (!completed) {  
 if (curAgent != null) {  
 if (availability) {  
 if (curLoad <= 0) {  
 completed = true;  
 System.*out*.println("Модуль id = " + id + " был выполнен агентом id = " + curAgent.getId());  
 curAgent = null;  
 return true;  
 } else {  
 System.*out*.println("Модуль id = " + id + " пока не выполнен (текущая нагрузка = " + curLoad + ") агентом id = " + curAgent.getId());  
 return false;  
 }  
 } else {  
 System.*out*.println("Модуль id = " + id + " не нужно проверять, т.к. он недоступен для выполнения");  
 return false;  
 }  
 } else {  
 System.*out*.println("Модуль id = " + id + " не нужно проверять, т.к. у него нет агента");  
 return false;  
 }  
 } else {  
 System.*out*.println("Модуль id = " + id + " не нужно проверять, т.к. он выполнен");  
 return false;  
 }  
 }  
  
 public boolean loadRecovery() {  
 if (!completed) {  
 if (curAgent != null) {  
 if (availability) {  
 curLoad = load;  
 System.*out*.println("Модуль id = " + id + " восстановил нагрузку из-за поломки своего агента (id = " + curAgent.getId() + ")");  
 curAgent = null;  
 return true;  
 } else {  
 System.*out*.println("Модулю id = " + id + " не нужно восстанавливать нагрузку, т.к. он недоступен для выполнения");  
 return false;  
 }  
 } else {  
 System.*out*.println("Модулю id = " + id + " не нужно восстанавливать нагрузку, т.к. у него нет агента (модуль не выполняется)");  
 return false;  
 }  
 } else {  
 System.*out*.println("Модулю id = " + id + " не нужно восстанавливать нагрузку, т.к. он выполнен");  
 return false;  
 }  
 }  
  
 public void addDependency(Module module) {  
 this.dependencies.add(module);  
 }  
  
 public int getId() {  
 return id;  
 }  
  
 public List<Module> getDependencies() {  
 return dependencies;  
 }  
  
 public boolean isCompleted() {  
 return completed;  
 }  
  
 public boolean isAvailability() {  
 return availability;  
 }  
  
 public float getCurLoad() {  
 return curLoad;  
 }  
  
 public void reduceLoad(float amount) {  
 this.curLoad -= amount;  
 }  
}

package org.example;  
  
import java.util.ArrayList;  
import java.util.List;  
  
public class GraphOfAgents {  
 private List<Agent> listOfAgents;  
  
 public GraphOfAgents(int n, float p) {  
 this.listOfAgents = new ArrayList<>();  
 for (int i = 0; i < n; i++) {  
 listOfAgents.add(new Agent(i + 1, p));  
 }  
 }  
  
 public void creatingNeighborhood(Agent agent1, Agent agent2) {  
 if (!agent1.getNeighbors().contains(agent2) && !agent2.getNeighbors().contains(agent1)) {  
 agent1.addNeighbor(agent2);  
 agent2.addNeighbor(agent1);  
 System.*out*.println("Агенты id = " + agent1.getId() + " и id = " + agent2.getId() + " соседи");  
 } else {  
 System.*out*.println("Не удалось установить соседство между агентами id = " + agent1.getId() + " и id = " + agent2.getId());  
 }  
 }  
  
 public boolean checkAgentsBreakdown() {  
 int count = 0;  
 for (Agent agent : listOfAgents) {  
 if (!agent.isExecution()) {  
 count++;  
 }  
 }  
 if (count == listOfAgents.size()) {  
 System.*out*.println("Все агенты сломались, дальнейшее выполнение программы невозможно!");  
 return false;  
 } else {  
 return true;  
 }  
 }  
  
 // Getter for listOfAgents  
 public List<Agent> getListOfAgents() {  
 return listOfAgents;  
 }  
}

package org.example;  
  
import java.util.ArrayList;  
import java.util.List;  
  
public class GraphOfModules {  
 private List<Module> listOfModules;  
  
 public GraphOfModules(int m, float a, float b) {  
 this.listOfModules = new ArrayList<>();  
 for (int i = 0; i < m; i++) {  
 listOfModules.add(new Module(i + 1, a, b));  
 }  
 }  
  
 public void creatingDependencies(Module moduleMain, Module moduleDependent) {  
 if (!moduleDependent.getDependencies().contains(moduleMain) && !moduleMain.getDependencies().contains(moduleDependent)) {  
 moduleDependent.addDependency(moduleMain);  
 System.*out*.println("Модуль id = " + moduleDependent.getId() + " зависим от модуля id = " + moduleMain.getId());  
 } else {  
 System.*out*.println("Не удалось установить зависимость между модулями id = " + moduleMain.getId() + " и id = " + moduleDependent.getId());  
 }  
 }  
  
 // Getter for listOfModules  
 public List<Module> getListOfModules() {  
 return listOfModules;  
 }  
}

package org.example;  
  
class Tuple<X, Y> {  
 private final X first;  
 private final Y second;  
  
 public Tuple(X first, Y second) {  
 this.first = first;  
 this.second = second;  
 }  
  
 public X getFirst() {  
 return first;  
 }  
  
 public Y getSecond() {  
 return second;  
 }  
}

package org.example;  
  
import java.util.List;  
  
  
public class Main {  
  
 public static Tuple<GraphOfAgents, List<Agent>> createGraphOfAgents(int n, double p) {  
 GraphOfAgents GraphOfAgents = new GraphOfAgents(n, (float) p);  
 List<Agent> list\_of\_agents = GraphOfAgents.getListOfAgents();  
 return new Tuple<>(GraphOfAgents, list\_of\_agents);  
 }  
  
 public static Tuple<GraphOfModules, List<Module>> createGraphOfModules(int m, double a, double b) {  
 GraphOfModules GraphOfModules = new GraphOfModules(m, (float) a, (float) b);  
 List<Module> list\_of\_modules = GraphOfModules.getListOfModules();  
 return new Tuple<>(GraphOfModules, list\_of\_modules);  
 }  
  
 public static void createModulesDependenciesV1(GraphOfModules GraphOfModules, List<Module> list\_of\_modules) {  
 GraphOfModules.creatingDependencies(list\_of\_modules.get(0), list\_of\_modules.get(1));  
 GraphOfModules.creatingDependencies(list\_of\_modules.get(0), list\_of\_modules.get(2));  
 GraphOfModules.creatingDependencies(list\_of\_modules.get(1), list\_of\_modules.get(2));  
 GraphOfModules.creatingDependencies(list\_of\_modules.get(1), list\_of\_modules.get(3));  
 GraphOfModules.creatingDependencies(list\_of\_modules.get(2), list\_of\_modules.get(3));  
 GraphOfModules.creatingDependencies(list\_of\_modules.get(3), list\_of\_modules.get(4));  
 }  
  
 public static void createModulesDependenciesV2(GraphOfModules GraphOfModules, List<Module> list\_of\_modules) {  
 GraphOfModules.creatingDependencies(list\_of\_modules.get(0), list\_of\_modules.get(1));  
 GraphOfModules.creatingDependencies(list\_of\_modules.get(0), list\_of\_modules.get(2));  
 GraphOfModules.creatingDependencies(list\_of\_modules.get(1), list\_of\_modules.get(3));  
 GraphOfModules.creatingDependencies(list\_of\_modules.get(2), list\_of\_modules.get(4));  
 }  
  
 public static void createModulesDependenciesV3(GraphOfModules GraphOfModules, List<Module> list\_of\_modules) {  
 GraphOfModules.creatingDependencies(list\_of\_modules.get(0), list\_of\_modules.get(3));  
 GraphOfModules.creatingDependencies(list\_of\_modules.get(1), list\_of\_modules.get(3));  
 GraphOfModules.creatingDependencies(list\_of\_modules.get(2), list\_of\_modules.get(3));  
 GraphOfModules.creatingDependencies(list\_of\_modules.get(3), list\_of\_modules.get(4));  
 GraphOfModules.creatingDependencies(list\_of\_modules.get(3), list\_of\_modules.get(5));  
 GraphOfModules.creatingDependencies(list\_of\_modules.get(3), list\_of\_modules.get(6));  
 }  
  
 public static void createAgentsNeighborhoodV1(GraphOfAgents GraphOfAgents, List<Agent> list\_of\_agents) {  
 GraphOfAgents.creatingNeighborhood(list\_of\_agents.get(0), list\_of\_agents.get(1));  
 GraphOfAgents.creatingNeighborhood(list\_of\_agents.get(1), list\_of\_agents.get(2));  
 }  
  
 public static void createAgentsNeighborhoodV2(GraphOfAgents GraphOfAgents, List<Agent> list\_of\_agents) {  
 GraphOfAgents.creatingNeighborhood(list\_of\_agents.get(0), list\_of\_agents.get(1));  
 GraphOfAgents.creatingNeighborhood(list\_of\_agents.get(1), list\_of\_agents.get(2));  
 GraphOfAgents.creatingNeighborhood(list\_of\_agents.get(2), list\_of\_agents.get(0));  
 }  
  
 public static void baseCycle(List<Module> list\_of\_modules, List<Agent> list\_of\_agents, GraphOfAgents GraphOfAgents) {  
 System.*out*.println("Начало выполнения программы");  
  
 int i = 0;  
  
 int count = 0;  
 for (Module module : list\_of\_modules) {  
 if (!module.isCompleted()) {  
 count++;  
 }  
 }  
  
 while (count > 0) {  
 i++;  
  
 for (Agent agent : list\_of\_agents) {  
 for (Module module : list\_of\_modules) {  
 if (agent.getCurModule() == null) {  
 agent.attemptGetModule(module);  
 }  
 }  
 if (agent.getCurModule() != null) {  
 agent.checkBreakdown();  
 }  
 }  
  
 boolean check = GraphOfAgents.checkAgentsBreakdown();  
 if (!check) {  
 return;  
 }  
  
 System.*out*.println("Текущая итерация = " + i);  
  
 count = 0;  
 for (Module module : list\_of\_modules) {  
 if (!module.isCompleted()) {  
 count++;  
 }  
 }  
  
 try {  
 Thread.*sleep*(500);  
 } catch (InterruptedException e) {  
 Thread.*currentThread*().interrupt();  
 }  
 }  
  
 System.*out*.println("Агенты успешно выполнили все модули за " + i + " итераций");  
 }  
  
 public static void mainCycle(int m, double a, double b, int n, double p) {  
 Tuple<GraphOfModules, List<Module>> modulesInfo = *createGraphOfModules*(m, a, b);  
 *createModulesDependenciesV1*(modulesInfo.getFirst(), modulesInfo.getSecond());  
 Tuple<GraphOfAgents, List<Agent>> agentsInfo = *createGraphOfAgents*(n, p);  
 *createAgentsNeighborhoodV1*(agentsInfo.getFirst(), agentsInfo.getSecond());  
  
 *baseCycle*(modulesInfo.getSecond(), agentsInfo.getSecond(), agentsInfo.getFirst());  
  
 modulesInfo = *createGraphOfModules*(5, 1.0, 7.1);  
 *createModulesDependenciesV2*(modulesInfo.getFirst(), modulesInfo.getSecond());  
 agentsInfo = *createGraphOfAgents*(3, 1.5);  
 *createAgentsNeighborhoodV2*(agentsInfo.getFirst(), agentsInfo.getSecond());  
  
 *baseCycle*(modulesInfo.getSecond(), agentsInfo.getSecond(), agentsInfo.getFirst());  
  
 modulesInfo = *createGraphOfModules*(7, 5.0, 10.1);  
 *createModulesDependenciesV3*(modulesInfo.getFirst(), modulesInfo.getSecond());  
 agentsInfo = *createGraphOfAgents*(3, 2.0);  
 *createAgentsNeighborhoodV2*(agentsInfo.getFirst(), agentsInfo.getSecond());  
  
 *baseCycle*(modulesInfo.getSecond(), agentsInfo.getSecond(), agentsInfo.getFirst());  
 }  
  
 public static void main(String[] args) {  
 int m = 5;  
 double a = 1.0;  
 double b = 5.1;  
 int n = 3;  
 double p = 1.0;  
  
 *mainCycle*(m, a, b, n, p);  
 }  
}

**Примеры вывода**







