Министерство 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: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ А. А. Иванов

Направление подготовки: 02.03.03 Математическое обеспечение и администрирование информационных систем

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

Краснодар

2024

**Цель работы**

Составить программу, которая будет моделировать взаимодействия между графом модулей системы и графом агентов, которые выполняют данные модули.

**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 параллельн 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**

Для написания программы был выбран язык Python ввиду его простой реализации ООП.

В ходе решения будет использоваться объектно-ориентированный подход, поскольку данную задачу можно разбить на классы для упрощения логики работы.

Было создано 6 файлов: task.txt (текстовое описание задачи), Module.py (класс, определяющий реализацию модуля), Graph\_of\_modules.py (класс, определяющий реализацию графа модулей), Agent.py (класс, определяющий реализацию агента), Graph\_of\_agents.py (класс, определяющий реализацию графа агентов), main.py (основной файл, в котором происходит моделирование взаимодействий агентов и модулей).

Описание файлов:

Module.py: данный класс создаёт объект-модуль, а также определяет основной функционал (check\_availability() – проверка доступности модуля для выполнения; appointment\_agent() – назначение агента модулю; check\_execution() – проверка завершённости модуля; load\_recovery() – восстановление нагрузки модуля после поломки агента);

Graph\_of\_modules.py: данный класс создаёт объект-граф модулей, а также определяет основной функционал (creating\_dependencies() – создание зависимостей между модулями в графе (благодаря зависимостям граф становится ориентированным));

Agent.py: данный класс создаёт объект-агента, а также определяет основной функционал (attempt\_get\_module() – попытка взять модуль на выполнение; execution\_step() – описывает итерационный шаг выполнения модуля (уменьшения его текущей нагрузки); check\_breakdown() – проверка поломки агента перед итерационным шагом);

Graph\_of\_agents.py: данный класс создаёт объект-граф агентов, а также определяет основной функционал (creating\_neighborhood() – создание соседства между агентами в графе (благодаря соседству граф становиться неориентированным));

main.py: данный файл определяет набор функций, которые создают необходимые графы, задают связи между объектами и запускают основной цикл симуляции.

**Код программы**

import random as rnd

class Module:

    def \_\_init\_\_(*self*, *id*: int, *a*: float, *b*: float) -> None:

        """

        Метод-конструктор

        Поля:

            id (int): Id модуля

            load (float): Общая нагрузка модуля

            cur\_load (float): Текущая нагрузка модуля

            dependencies (list): Зависимости модуля (другие модули, которые нужно выполнить, чтобы получить доступ к этому модулю)

            availability (bool): Доступность выполнения модуля (доступен/не доступен)

            cur\_agent (None/Agent): Текущий агент модуля

            completed (bool): Завершённость модуля (выполнен/не выполнен)

        Args:

            id (int): Id модуля

            a (float): Нижняя граница нагрузки

            b (float): Верхняя граница нагрузки

        Returns:

            None

        """

*self*.id = *id*

*self*.load = round(rnd.uniform(*a*, *b*), 1)

*self*.cur\_load = *self*.load

*self*.dependencies = []

*self*.availability = False

*self*.cur\_agent = None

*self*.completed = False

    def check\_availability(*self*) -> bool:

        """

        Метод, который проверяет возможность выполнения модуля

        Модуль считается доступным, если он: не выполнен; не имеет текущего агента; является стартовым или выполнены все его зависимости

        Args:

            None

        Returns:

            res (bool): Результат проверки

        """

        if *self*.completed == False:

            if *self*.cur\_agent == None:

                if len(*self*.dependencies) == 0:

*self*.availability = True

                    print(f"Модуль id = {*self*.id} можно взять на выполнение (стартовый)")

                    return True

                else:

                    for dep in *self*.dependencies:

                        if dep.completed == False:

                            print(f"Модуль id = {*self*.id} недоступен для выполнения пока не будут выполнены все его зависимости")

                            return False

                    else:

*self*.availability = True

                        print(f"Модуль id = {*self*.id} можно взять на выполнение (все зависимости выполнены)")

                        return True

            else:

                print(f"Модуль id = {*self*.id} занят агентом id = {*self*.cur\_agent.id}")

                return False

        else:

            print(f"Модуль id = {*self*.id} уже выполнен")

            return False

    def appointment\_agent(*self*, *agent*) -> None:

        """

        Метод, который назначает модулю агента для выполнения

        Модулю можно назначить агента, если он: не выполнен; не имеет текущего агента; является доступным

        Args:

            agent (Agent): Агент, который пробует назначить себе модуль

        Returns:

            None

        """

        if *self*.completed == False:

            if *self*.cur\_agent == None:

                if *self*.availability == True:

*self*.cur\_agent = *agent*

                    print(f"Модулю id = {*self*.id} был назначен агент id = {*self*.cur\_agent.id}")

                else:

                    print(f"Модулю id = {*self*.id} нельзя назначить агента id = {*agent*.id}, т.к. модуль недоступен для выполнения")

            else:

                print(f"Модулю id = {*self*.id} нельзя назначить агента id = {*agent*.id}, т.к. модуль занят агентом id = {*self*.cur\_agent.id}")

        else:

            print(f"Модулю id = {*self*.id} нельзя назначить агента id = {*agent*.id}, т.к. модуль выполнен")

    def check\_execution(*self*) -> bool:

        """

        Метод, который проверяет завершённость модуля

        Модуль считается завершённым, если его текущая нагрузка <= 0

        Args:

            None

        Returns:

            res (bool): Результат проверки

        """

        if *self*.completed == False:

            if *self*.cur\_agent != None:

                if *self*.availability == True:

                    if *self*.cur\_load <= 0:

*self*.completed = True

                        print(f"Модуль id = {*self*.id} был выполнен агентом id = {*self*.cur\_agent.id}")

*self*.cur\_agent = None

                        return True

                    else:

                        print(f"Модуль id = {*self*.id} пока не выполнен (текущая нагрузка = {*self*.cur\_load}) агентом id = {*self*.cur\_agent.id}")

                        return False

                else:

                    print(f"Модуль id = {*self*.id} не нужно проверять, т.к. он недоступен для выполнения")

                    return False

            else:

                print(f"Модуль id = {*self*.id} не нужно проверять, т.к. у него нет агента")

                return False

        else:

            print(f"Модуль id = {*self*.id} не нужно проверять, т.к. он выполнен")

            return False

    def load\_recovery(*self*) -> bool:

        """

        Метод, который восстанавливает агенту нагрузку в случае поломки агента, который модуль выполнял

        Нагрузка восстанавливается, если модуль: не выполнен; у него есть агента; модуль доступен

        Args:

            None

        Returns:

            res (bool): Результат восстановления нагрузки

        """

        if *self*.completed == False:

            if *self*.cur\_agent != None:

                if *self*.availability == True:

*self*.cur\_load = *self*.load

                    print(f"Модуль id = {*self*.id} восстановил нагрузку из-за поломки своего агента (id = {*self*.cur\_agent.id})")

*self*.cur\_agent = None

                    return True

                else:

                    print(f"Модулю id = {*self*.id} не нужно восстанавливать нагрузку, т.к. он недоступен для выполнения")

                    return False

            else:

                print(f"Модулю id = {*self*.id} не нужно восстанавливать нагрузку, т.к. у него нет агента (модуль не выполняется)")

                return False

        else:

            print(f"Модулю id = {*self*.id} не нужно восстанавливать нагрузку, т.к. он выполнен")

            return False

from Module import \*

class Graph\_of\_modules:

    def \_\_init\_\_(*self*, *m*: int, *a*: float, *b*: float) -> None:

        """

        Метод-конструктор

        Создаёт m модулей

        Поля:

            list\_of\_modules (list): Список всех созданных модулей

        Args:

            m (int): Кол-во модулей

            a (float): Нижняя граница нагрузки модуля

            b (float): Верхняя граница нагрузки модуля

        Returns:

            None

        """

*self*.list\_of\_modules = [Module(i + 1, *a*, *b*) for i in range(*m*)]

    def creating\_dependencies(*self*, *module\_main*: Module, *module\_dependent*: Module) -> None:

        """

        Метод, который устанавливает зависимость между парой модулей

        Args:

            module\_main (Module): Главный модуль

            module\_dependent (Module): Зависимый модуль

        Returns:

            None

        """

        if *module\_dependent* not in *module\_main*.dependencies and *module\_main* not in *module\_dependent*.dependencies:

*module\_dependent*.dependencies.append(*module\_main*)

            print(f"Модуль id = {*module\_dependent*.id} зависим от модуля id = {*module\_main*.id}")

        else:

            print(f"Не удалось установить зависимость между модулями id = {*module\_main*.id} и id = {*module\_dependent*.id}")

import random as rnd

class Agent:

    def \_\_init\_\_(*self*, *id*: int, *p*: float) -> None:

        """

        Метод-конструктор

        Поля:

            id (int): Id агента

            neighbors (list): Список соседей агента, которым он может передать модуль

            cur\_module (None/Module): Текущий модуль агента

            execution (bool): Возможность агента выполнять вычисления модуля

            amount\_of\_execution (float): Величина, на которую агент уменьшает текущую нагрузку модуля каждую итерацию выполнения

        Args:

            id (int): Id агента

            p (float): Величина выполнения агента

        Returns:

            None

        """

*self*.id = *id*

*self*.neighbors = []

*self*.cur\_module = None

*self*.execution = True

*self*.amount\_of\_execution = *p*

    def attempt\_get\_module(*self*, *module*) -> None:

        """

        Метод, который пробует взять модуль на выполнение

        Агент может взять модуль на выполнение, если он: не сломан; не имеет модуль; модуль доступен

        Args:

            module (Module): Модуль, который агент пробует взять на выполнение

        Returns:

            None

        """

        if *self*.execution == True:

            if *self*.cur\_module == None:

                check = *module*.check\_availability()

                if check == True:

*self*.cur\_module = *module*

                    print(f"Агент id = {*self*.id} взял модуль id = {*self*.cur\_module.id} на выполнение")

*module*.appointment\_agent(*self*)

                else:

                    print(f"Агент id = {*self*.id} не может взять модуль id = {*module*.id} на выполнение")

            else:

                print(f"Агент id = {*self*.id} не может взять модуль id = {*module*.id}, т.к. у агента уже есть модуль id = {*self*.cur\_module.id}")

        else:

            print(f"Агент id = {*self*.id} не может взять модуль id = {*module*.id}, т.к. агент сломался")

    def execution\_step(*self*) -> None:

        """

        Метод, который выполняет шаг выполнения модуля агентом

        Шаг выполняется, если агент: не сломан; имеет модуль

        Args:

            None

        Returns:

            None

        """

        if *self*.execution == True:

            if *self*.cur\_module != None:

                check = *self*.cur\_module.check\_execution()

                if check == False:

*self*.cur\_module.cur\_load -= *self*.amount\_of\_execution

                    print(f"Агент id = {*self*.id} уменьшил нагрузку своего модуля (id = {*self*.cur\_module.id}) на {*self*.amount\_of\_execution}")

                    check = *self*.cur\_module.check\_execution()

                    if check == True:

                        print(f"Агент id = {*self*.id} выполнил модуль id = {*self*.cur\_module.id}")

*self*.cur\_module = None

                else:

                    print(f"Агент id = {*self*.id} выполнил модуль id = {*self*.cur\_module.id}")

*self*.cur\_module = None

            else:

                print(f"Агент id = {*self*.id} не может выполнять модуль, т.к. агент не имеет модуля")

        else:

            print(f"Агент id = {*self*.id} не может выполнять модуль, т.к. агент сломался")

    def check\_breakdown(*self*) -> None:

        """

        Метод, который на итерационном шаге проверяет, не сломался ли агент

        Если агент сломался, то его модуль восстанавливает нагрузку, а агент пытается передать этот модуль соседу

        Args:

            None

        Returns:

            None

        """

        b = rnd.randint(1, 100)

        if b <= 5:

            if *self*.execution == True:

                if *self*.cur\_module != None:

                    check = *self*.cur\_module.load\_recovery()

                    if check == True:

*self*.execution = False

                        module = *self*.cur\_module

*self*.cur\_module = None

                        if len(*self*.neighbors) >= 1:

                            for neig in *self*.neighbors:

                                if neig.execution == True and neig.cur\_module == None:

                                    neig.cur\_module = module

                                    print(f"Агент id = {*self*.id} сломался и передал свой модуль id = {module.id} агенту-соседу id = {neig.id}")

                                    module.appointment\_agent(neig)

                                    return

                                else:

                                    print(f"Агент id = {*self*.id} сломался, и не может передать свой модуль id = {module.id} агенту-соседу id = {neig.id}")

                        else:

                            print(f"Агент id = {*self*.id} не имеет соседей... ОШИБКА")

                            raise ValueError

                    else:

                        print(f"Агент id = {*self*.id} сломался, а его модуль id = {module.id} не восстановил нагрузку (0\_0)")

                else:

                    print(f"Агент id = {*self*.id} не может сломаться без модуля... ОШИБКА")

                    raise ValueError

            else:

                print(f"Агент id = {*self*.id} не может сломаться второй раз... ОШИБКА")

                raise ValueError

        else:

*self*.execution\_step()

from Agent import \*

class Graph\_of\_agents:

    def \_\_init\_\_(*self*, *n*: int, *p*: float) -> None:

        """

        Метод-конструктор

        Создаёт n агентов

        Поля:

            list\_of\_agents (list): Список всех агентов

        Args:

            n (int): Кол-во агентов

            p (float): Величина выполнения агента

        Returns:

            None

        """

*self*.list\_of\_agents = [Agent(i + 1, *p*) for i in range(*n*)]

    def creating\_neighborhood(*self*, *agent\_1*: Agent, *agent\_2*: Agent) -> None:

        """

        Метод, который устанавливает соседство между парой агентов

        Args:

            agent\_1 (Agent): Первый агент

            agent\_2 (Agent): Второй агент

        Returns:

            None

        """

        if *agent\_1* not in *agent\_2*.neighbors and *agent\_2* not in *agent\_1*.neighbors:

*agent\_1*.neighbors.append(*agent\_2*)

*agent\_2*.neighbors.append(*agent\_1*)

            print(f"Агенты id = {*agent\_1*.id} и id = {*agent\_2*.id} соседи")

        else:

            print(f"Не удалось установить соседство между агентами id = {*agent\_1*.id} и id = {*agent\_2*.id}")

    def check\_agents\_breakdown(*self*) -> bool:

        """

        Метод, который проверяет всех агентов на предмет общей поломки

        Args:

            None

        Returns:

            res (bool): Результат проверки

        """

        count = 0

        for agent in *self*.list\_of\_agents:

            if agent.execution == False:

                count += 1

        if count == len(*self*.list\_of\_agents):

            print(f"Все агенты сломались, дальнейшее выполнение программы невозможно!")

            return False

        else:

            return True

from Module import \*

from Agent import \*

from Graph\_of\_modules import \*

from Graph\_of\_agents import \*

import typing as tp

import time

def create\_graph\_of\_agents(*n*: int, *p*: float) -> tp.Tuple[Graph\_of\_agents, list]:

    """

    Функция, которая создаёт граф агентов

    Args:

        n (int): Кол-во агентов

        p (float): Величина выполнения агентов

    Returns:

        graph\_of\_agents (Graph\_of\_agents): Граф агентов

        list\_of\_agents (list): Список всех агентов

    """

    graph\_of\_agents = Graph\_of\_agents(*n*, *p*)

    list\_of\_agents = graph\_of\_agents.list\_of\_agents

    return graph\_of\_agents, list\_of\_agents

def create\_graph\_of\_modules(*m*: int, *a*: float, *b*: float) -> tp.Tuple[Graph\_of\_modules, list]:

    """

    Функция, которая создаёт граф модулей

    Args:

        m (int): Кол-во модулей

        a (float): Нижняя граница нагрузки модуля

        b (float): Верхняя граница нагрузки модуля

    Returns:

        graph\_of\_modules (Graph\_of\_modules): Граф модулей

        list\_of\_modules (list): Список всех модулей

    """

    graph\_of\_modules = Graph\_of\_modules(*m*, *a*, *b*)

    list\_of\_modules = graph\_of\_modules.list\_of\_modules

    return graph\_of\_modules, list\_of\_modules

def create\_modules\_dependencies\_v1(*graph\_of\_modules*: Graph\_of\_modules, *list\_of\_modules*: list) -> None:

    """

    Функция, которая задаёт связи для ориентированного графа модулей

    Args:

        graph\_of\_modules (Graph\_of\_modules): Граф модулей

        list\_of\_modules (list): Список всех модулей

    Returns:

        None

    """

*graph\_of\_modules*.creating\_dependencies(*list\_of\_modules*[0], *list\_of\_modules*[1])

*graph\_of\_modules*.creating\_dependencies(*list\_of\_modules*[0], *list\_of\_modules*[2])

*graph\_of\_modules*.creating\_dependencies(*list\_of\_modules*[1], *list\_of\_modules*[2])

*graph\_of\_modules*.creating\_dependencies(*list\_of\_modules*[1], *list\_of\_modules*[3])

*graph\_of\_modules*.creating\_dependencies(*list\_of\_modules*[2], *list\_of\_modules*[3])

*graph\_of\_modules*.creating\_dependencies(*list\_of\_modules*[3], *list\_of\_modules*[4])

def create\_modules\_dependencies\_v2(*graph\_of\_modules*: Graph\_of\_modules, *list\_of\_modules*: list) -> None:

    """

    Функция, которая задаёт связи для ориентированного графа модулей

    Args:

        graph\_of\_modules (Graph\_of\_modules): Граф модулей

        list\_of\_modules (list): Список всех модулей

    Returns:

        None

    """

*graph\_of\_modules*.creating\_dependencies(*list\_of\_modules*[0], *list\_of\_modules*[1])

*graph\_of\_modules*.creating\_dependencies(*list\_of\_modules*[0], *list\_of\_modules*[2])

*graph\_of\_modules*.creating\_dependencies(*list\_of\_modules*[1], *list\_of\_modules*[3])

*graph\_of\_modules*.creating\_dependencies(*list\_of\_modules*[2], *list\_of\_modules*[4])

def create\_modules\_dependencies\_v3(*graph\_of\_modules*: Graph\_of\_modules, *list\_of\_modules*: list) -> None:

    """

    Функция, которая задаёт связи для ориентированного графа модулей

    Args:

        graph\_of\_modules (Graph\_of\_modules): Граф модулей

        list\_of\_modules (list): Список всех модулей

    Returns:

        None

    """

*graph\_of\_modules*.creating\_dependencies(*list\_of\_modules*[0], *list\_of\_modules*[3])

*graph\_of\_modules*.creating\_dependencies(*list\_of\_modules*[1], *list\_of\_modules*[3])

*graph\_of\_modules*.creating\_dependencies(*list\_of\_modules*[2], *list\_of\_modules*[3])

*graph\_of\_modules*.creating\_dependencies(*list\_of\_modules*[3], *list\_of\_modules*[4])

*graph\_of\_modules*.creating\_dependencies(*list\_of\_modules*[3], *list\_of\_modules*[5])

*graph\_of\_modules*.creating\_dependencies(*list\_of\_modules*[3], *list\_of\_modules*[6])

def create\_agents\_neighborhood\_v1(*graph\_of\_agents*: Graph\_of\_agents, *list\_of\_agents*: list) -> None:

    """

    Функция, которая задаёт связи для неориентированного графа агентов

    Args:

        graph\_of\_agents (Graph\_of\_agents): Граф агентов

        list\_of\_agents (list): Список всех агентов

    Returns:

        None

    """

*graph\_of\_agents*.creating\_neighborhood(*list\_of\_agents*[0], *list\_of\_agents*[1])

*graph\_of\_agents*.creating\_neighborhood(*list\_of\_agents*[1], *list\_of\_agents*[2])

def create\_agents\_neighborhood\_v2(*graph\_of\_agents*: Graph\_of\_agents, *list\_of\_agents*: list) -> None:

    """

    Функция, которая задаёт связи для неориентированного графа агентов

    Args:

        graph\_of\_agents (Graph\_of\_agents): Граф агентов

        list\_of\_agents (list): Список всех агентов

    Returns:

        None

    """

*graph\_of\_agents*.creating\_neighborhood(*list\_of\_agents*[0], *list\_of\_agents*[1])

*graph\_of\_agents*.creating\_neighborhood(*list\_of\_agents*[1], *list\_of\_agents*[2])

*graph\_of\_agents*.creating\_neighborhood(*list\_of\_agents*[2], *list\_of\_agents*[0])

def base\_cycle(*list\_of\_modules*: list, *list\_of\_agents*: list, *graph\_of\_agents*: Graph\_of\_agents) -> None:

    """

    Функция, которая выполняет основной цикл работы системы

    Args:

        list\_of\_modules (list): Список всех модулей

        list\_of\_agents (list): Список всех агентов

        graph\_of\_agents (Graph\_of\_agents): Граф агентов

    Returns:

        None

    """

    print("Начало выполнения программы")

    i = 0

    count = 0

    for module in *list\_of\_modules*:

        if module.completed == False:

            count += 1

    while count > 0:

        i += 1

        for agent in *list\_of\_agents*:

            for module in *list\_of\_modules*:

                if agent.cur\_module == None:

                    agent.attempt\_get\_module(module)

            if agent.cur\_module != None:

                agent.check\_breakdown()

        check = *graph\_of\_agents*.check\_agents\_breakdown()

        if check == False:

            return

        print(f"Текущая итерация = {i}")

        count = 0

        for module in *list\_of\_modules*:

            if module.completed == False:

                count += 1

        time.sleep(0.5)

    print(f"Агенты успешно выполнили все модули за {i} итераций")

    print()

def main\_cycle(*m*, *a*, *b*, *n*, *p*):

    """

    Функция, которая запускает основной цикл работы программы

    Цикл работает, пока все модули не будут выполнены

    Args:

        list\_of\_modules (list): Список всех модулей

        list\_of\_agents (list): Список всех агентов

    Returns:

        None

    """

    graph\_of\_modules, list\_of\_modules = create\_graph\_of\_modules(*m*, *a*, *b*)

    create\_modules\_dependencies\_v1(graph\_of\_modules, list\_of\_modules)

    graph\_of\_agents, list\_of\_agents = create\_graph\_of\_agents(*n*, *p*)

    create\_agents\_neighborhood\_v1(graph\_of\_agents, list\_of\_agents)

    base\_cycle(list\_of\_modules, list\_of\_agents, graph\_of\_agents)

    graph\_of\_modules, list\_of\_modules = create\_graph\_of\_modules(5, 1.0, 7.1)

    create\_modules\_dependencies\_v2(graph\_of\_modules, list\_of\_modules)

    graph\_of\_agents, list\_of\_agents = create\_graph\_of\_agents(3, 1.5)

    create\_agents\_neighborhood\_v2(graph\_of\_agents, list\_of\_agents)

    base\_cycle(list\_of\_modules, list\_of\_agents, graph\_of\_agents)

    graph\_of\_modules, list\_of\_modules = create\_graph\_of\_modules(7, 5.0, 10.1)

    create\_modules\_dependencies\_v3(graph\_of\_modules, list\_of\_modules)

    graph\_of\_agents, list\_of\_agents = create\_graph\_of\_agents(3, 2.0)

    create\_agents\_neighborhood\_v2(graph\_of\_agents, list\_of\_agents)

    base\_cycle(list\_of\_modules, list\_of\_agents, graph\_of\_agents)

m = 5

a = 1.0

b = 5.1

n = 3

p = 1.0

if \_\_name\_\_ == "\_\_main\_\_":

    main\_cycle(m, a, b, n, p)

Вышеописанный код реализует симуляцию для следующих графов:

А) модулей (Рисунок 1, Рисунок 2, Рисунок 3):

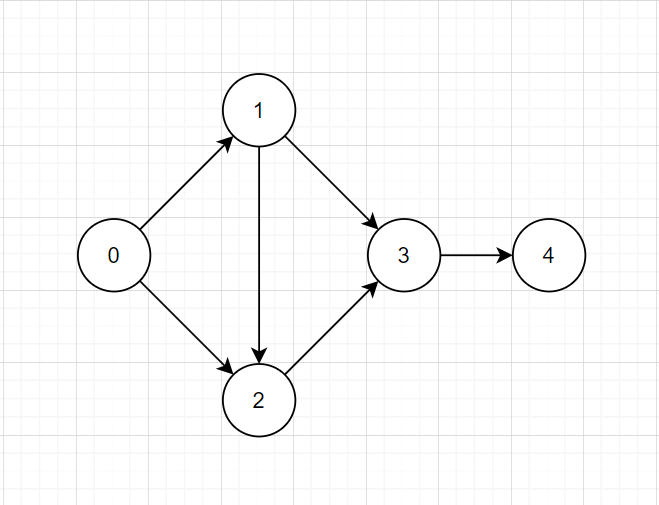


Рисунок 1 – первый граф модулей

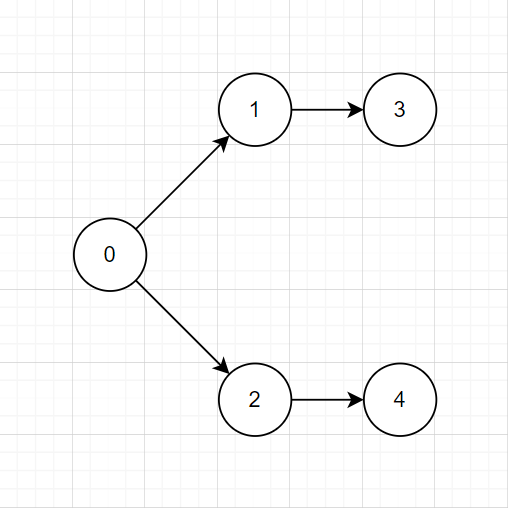


Рисунок 2 – второй граф модулей

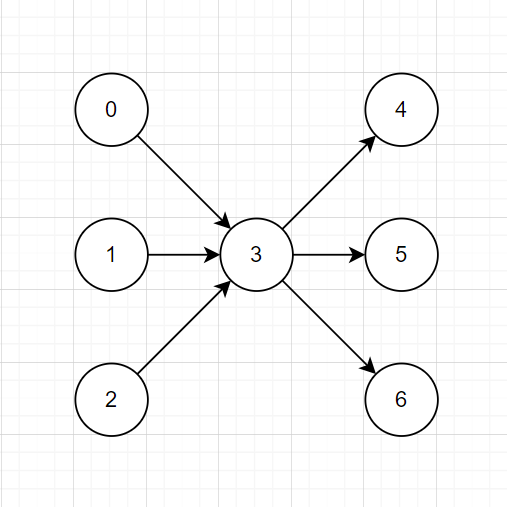


Рисунок 3 – третий граф модулей

Б) агентов (Рисунок 4, Рисунок 5):

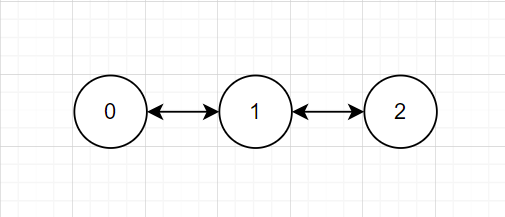


Рисунок 4 – первый граф агентов

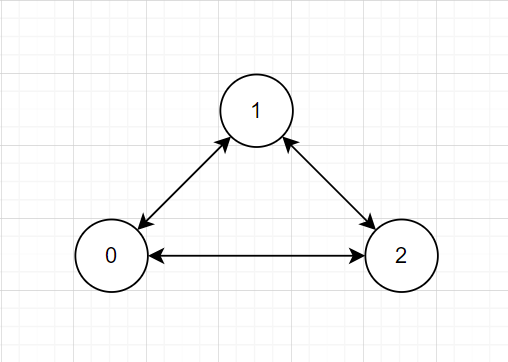
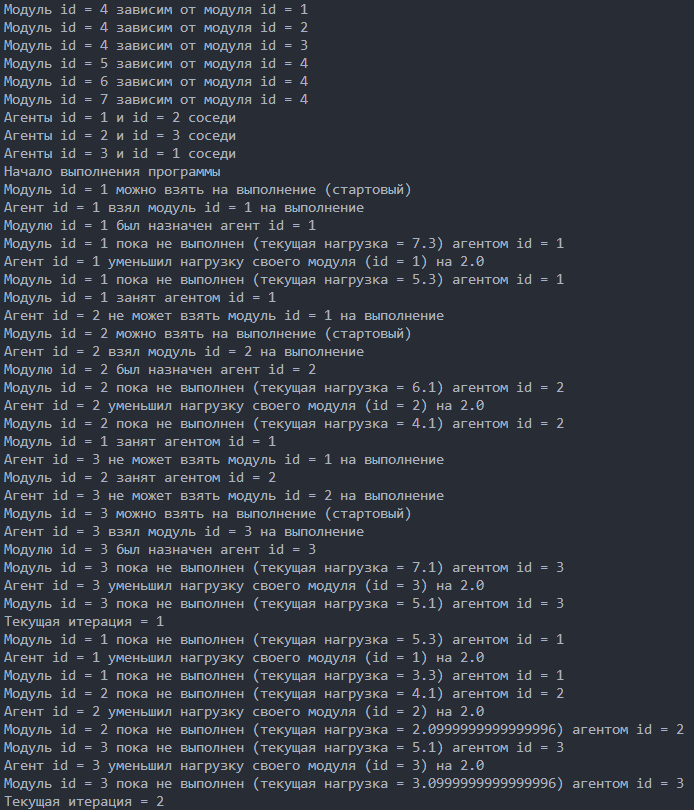
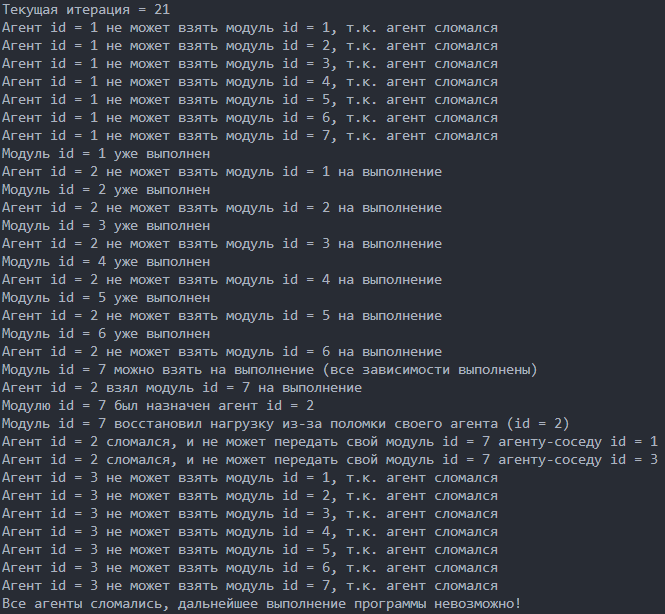


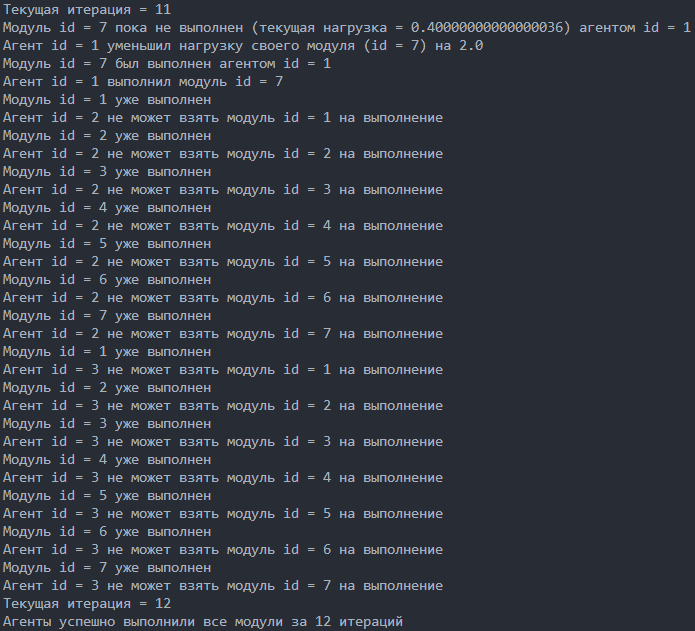
Рисунок 5 – второй граф агентов

Иначе говоря, вышеописанный код запускает симуляции для (А1, Б1), (А2, Б2), (А3, Б2).

**Вывод программы**

****

****

****