

МИНОБРНАУКИ РОССИИ  
Федеральное государственное бюджетное образовательное учреждение  
высшего образования  
«САРАТОВСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ  
ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИМЕНИ Н. Г. ЧЕРНЫШЕВСКОГО»  
(СГУ)

Кафедра математической  
кибернетики и компьютерных наук

**ЭВРИСТИЧЕСКИЙ АЛГОРИТМ ПОИСКА ЦЕНТРАЛЬНЫХ ВЕРШИН  
ГРАФОВ**

**КУРСОВАЯ РАБОТА**

студента 3 курса 311 группы  
направления 02.03.02 – Фундаментальная информатика и информационные  
технологии  
факультета КНиИТ  
Власова Андрея Александровича

Научный руководитель  
к. ф.-м. н.

---

С. В. Миронов

Заведующий кафедрой  
к. ф.-м. н.

---

С. В. Миронов

Саратов 2019

## СОДЕРЖАНИЕ

ВВЕДЕНИЕ .....	3
1 Описание задачи .....	4
2 Существующие алгоритмы для решения задачи .....	5
2.1 Точные алгоритмы .....	5
2.1.1 Тривиальный алгоритм .....	5
2.1.2 Алгоритмы, использующие матричное умножение .....	5
2.1.3 Алгоритмы с улучшенной асимптотикой .....	6
2.2 Эвристические алгоритмы .....	6
2.2.1 Генетический алгоритм .....	6
3 Описание созданного генетического алгоритма .....	9
3.1 Инициализация начальной популяции .....	10
3.2 Естественный отбор .....	10
3.3 Процесс кроссинговера .....	10
3.4 Процесс мутации .....	12
4 Реализация алгоритма .....	13
5 Модели случайного графа .....	14
5.1 Модель случайного графа Эрдеша-Ренъи .....	14
5.2 Модель случайного графа Барабаши-Альберт .....	14
5.3 Геометрический случайный граф .....	15
6 Результаты вычислительных экспериментов .....	16
ЗАКЛЮЧЕНИЕ .....	21
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ .....	22
Приложение А Листинг проекта .....	24
Приложение Б СД с отчетом о проделанной работе .....	52

## **ВВЕДЕНИЕ**

В современных компьютерных науках одно из центральных мест занимает математическая модель графа. Графовая модель позволяет описывать целый ряд систем и явлений, которые встречаются в различных предметных областях. Изучение структуры того или иного графа, или же выявление его свойств и особенностей может носить невероятную практическую пользу. В связи с этим работы, посвященные этой теме, вызывают наибольший интерес как со стороны исследователей теоретиков, так и со стороны практиков.

Одной из важных характеристик любого графа можно назвать радиус графа и расположение его центральных вершин. Получив данные параметры, можно судить об общей структуре графа или же о его особенностях. Современные графовые модели могут насчитывать десятки сотен тысяч вершин, поэтому для эффективного решения задач зачастую бывает недостаточно использовать классические алгоритмы, а необходимо реализовать некоторый эвристический подход, одним из которых является генетический алгоритм.

Цель данной работы — предложить эффективный алгоритм поиска центральных вершин в графе, в своей основе реализующий концепции генетического алгоритма, который бы позволил решать задачу за приемлемые временные затраты.

Для достижения данной цели были поставлены следующие задачи:

- проанализировать и реализовать существующие алгоритмы поиска центральных вершин,
- создать генетический алгоритм поиска центральных вершин,
- провести сравнение существующих алгоритмов и созданного и выявить преимущества и недостатки последнего.

## 1 Описание задачи

Формально, рассматриваемую задачу можно описать следующим образом. Пусть задан невзвешенный неориентированный граф  $G = (V, E)$ , где  $V$  — множество вершин,  $E$  — множество ребер.

Также задана функция  $F : V \mapsto \mathbb{R}$ , определенная на множестве вершин и принимающая действительные значения. Оптимизационная задача заключается в том, чтобы найти минимум данной функции, и соответствующую вершину  $v \in V$  такую, что  $F(v) = \min_{x \in V} F(x)$ .

В данной работе в роли функции  $F(v)$  выступает эксцентризитет вершины  $e$  — максимальное из всех расстояний от вершины до другой вершины. Найденный минимум функции  $F(v)$  будет достигаться на вершинах, которые в теории графов называются центральными, а значение минимального эксцентризита — радиусом графа. То есть всю задачу можно описать следующей формулой:

$$R = \min_{x \in V} \max_{y \in V} (d(x, y)),$$

где  $d(x, y)$  — расстояние между вершинами  $x, y$ ;  $R$  — радиус графа.

## **2 Существующие алгоритмы для решения задачи**

Вследствие естественности описанных характеристик, радиус графа и эксцентриситет вершины являются одними из базовых параметров, которые наиболее часто встречаются в прикладных задачах, либо необходимы при исследовании свойств графа [1], [2]. Этим объясняется довольно большое количество работ, в которых изучается данная задача. Основные популярные методы решения можно разделить на алгоритмы дающих точные решения (или с фиксированным уровнем ошибки) и алгоритмы, внутри которых реализована эвристика, позволяющая резко увеличить скорость выполнения за счет увеличения вероятности найти субоптимальное решение. Как раз к последним можно отнести генетические алгоритмы.

### **2.1 Точные алгоритмы**

Одним из очевидных наблюдений в данной задачи является тот факт, что рассматриваемый алгоритм поиска центральных вершин неразрывно связана с проблемой поиска кратчайших путей между всеми парами вершин. Существующие алгоритмы, можно разделить на два подхода, в первом из которых граф представляется в виде матрицы смежности и дальнейших вычислениях используется матричное умножение. Другим направлением можно назвать алгоритмы, которые используют в своей работе некоторые оптимизационные идеи для асимптотического улучшения времени работы.

#### **2.1.1 Тривиальный алгоритм**

Тривиальный алгоритм может быть реализован запуском из каждой вершины графа поиска в ширину, который получит расстояние от текущей вершины до всех остальных. Зная расстояния между всеми парами вершин, не составляет труда найти эксцентриситет вершины и значение радиуса. При этом асимптотическое время работы равно  $O(n^2 + nm)$ , где  $n$  — количество вершин, а  $m$  — количество ребер в графе. Данная оценка при высоком количестве ребер равносильна оценке  $O(n^3)$ .

#### **2.1.2 Алгоритмы, использующие матричное умножение**

Одним из естественных способов задания неориентированного невзвешенного графа может служить бинарная матрица смежности, в которой единичные элементы описывают ребра, соединяющие вершины в соответствую-

щей строке и столбце. Ввиду этого для поиска кратчайших расстояний между вершинами может быть применен алгоритм, описанный в статье [3], в основе которого лежит матричное умножение. Алгоритм матричного умножения также имеет время выполнения  $O(n^3)$ , при этом существуют способы оптимизации, способные улучшить эту оценку до  $O(n^w)$ , где  $w \leq 2.81$  [4].

### 2.1.3 Алгоритмы с улучшенной асимптотикой

В работе [5] приводится алгоритм, позволяющий найти радиус графа и расстояние между всеми парами вершин. Время работы алгоритма имеет асимптотическое улучшение и равно  $O(m\sqrt{n} + n^2)$ . Основной идеей данного алгоритма является эвристика разделения вершин на два множества, в одно из которых попадают вершины с высокой степенью, а во второе — с низкой. Для первого набора вершин строится доминирующее множество — множество вершин графа, такое, что любая вершина вне доминирующего множества смежна хотя бы с одной вершиной из него; из которого запускаются обходы в ширину, подсчитывающие расстояния. Также обходы в ширину запускаются внутри множества вершин с низкой степенью, при этом для этого случая обход проводится не по всем вершинам графа, а лишь по элементам, находящимся внутри данного множества. Такой подход позволяет получить искомый радиус графа или же найти центральные вершины за меньшие временные затраты по сравнению с другими алгоритмами. Данная работа получила дальнейшее развитие в [6–8], где используются различные подходы модернизирующие существующий, позволяющие ускорить алгоритм для некоторых видов графов.

## 2.2 Эвристические алгоритмы

### 2.2.1 Генетический алгоритм

Впервые идея генетического алгоритма, как метода решения оптимизационных задач, была предложена в работе [9]. В основе любого генетического алгоритма лежит процесс моделирования эволюционного развития, который во многом определяется генетическими механизмами в живых организмах и естественным отбором.

Для каждой рассматриваемой задачи выбирается способ кодирования решения, после чего формируется начальная популяция, состоящая из набора возможных решений, которая будет эволюционировать. В качестве генетических процессов используется два оператора, которые изменяют текущую

популяцию в сторону правильного ответа.

Первый из этих них — оператор скрещивания (кроссинговера). Ему на вход подается два представителя из текущей популяции, после чего с помощью некоторых преобразований формируется некоторая новая особь, которой частично передались свойства двух ее родителей. Данный оператор позволяет переносить некоторые найденные лучшие решения, при этом комбинируя их для возможного последующего улучшения.

При использовании только такого подхода в качестве решения может найтись некоторое локальное оптимальное значение, которое не будет совпадать с глобальным, поэтому также в работу алгоритма вводится оператор мутации, который некоторым образом с заданной вероятностью изменяет текущую особь, что позволяет расширить круг поиска и не сбиться в локальный экстремум.

Также неотъемлемой частью любого генетического алгоритма является моделирование процесса естественного отбора. С помощью него удается перенести в следующее поколение только особи, которые представляют наилучшее решение, при этом отсеив худшие. Таким образом в генетическом алгоритме итерационно происходит процесс естественного отбора, скрещивания и мутации. Данный подход основывается на эвристике позаимствованной из наблюдений за живой природой и позволяет эффективно решать оптимизационную задачу в тех случаях, где это не представляется возможным сделать с помощью точных методов.

Для решения рассматриваемой задачи также были созданы различные эвристические алгоритмы. Например, в работе [10], приводится генетический алгоритм, позволяющий решать различного рода задачи из теории графов.

В основе построения данного алгоритма лежат классические этапы генетического подхода, при этом решение кодируется набором вершин, заданной мощности. В этой же работе рассматривается применение генетического алгоритма для решения целого ряда задач, одна из которых проблема нахождения  $k$ -центральных вершин, которая заключается в том, что необходимо найти ровно  $k$  центральных вершин, сумма эксцентрикитетов которых была бы минимальна. Несложно заметить, что задача будет совпадать с проблемой поиска центральной вершины, если в качестве  $k$  выбрать 1.

В описываемом алгоритме оператор скрещивания имеет следующую реа-

лизацию. Рассматриваются две особи (два множества), которые должны перейти в следующее поколение. Далее находится разность второго множества с первым, которая называется первым вектором обмена, также находится разность первого со вторым, которая называется вторым вектором обмена, после чего генерируется случайная позиция, относительно которой будет производится скрещивание. После проделанных операций происходит обмен элементами относительно найденной позиции каждого множества с соответствующим ему вектором обмена. За счет такого подхода на каждой итерации алгоритма в множестве, описывающем конкретную особь не существует повторяющихся вершин, при этом размер этого множества остается неизменным.

В качестве оператора мутации используется следующий подход: для каждой вершины находится набор ее соседей, после чего из этого множества случайным образом выбирается 4 вершины, которых нет в множестве особи. Для каждой из четырех вершин находится ее эксцентриситет, после чего с вероятностью заданной для оператора мутации вершина-сосед заменяет вершину в популяции. Данный подход был назван авторами N4N эвристикой, поэтому далее данный алгоритм будет называться «N4N алгоритм». В естественном отборе в качестве целевой функции ставиться значение эксцентриситета вершины.

Данный подход в реализации генетического алгоритма позволяет решать не только задачу поиска центральных вершин, но и еще ряд проблем связанных с теорией графов, однако ввиду общности способа представления решения и реализации оператора скрещивания может иметь свои недостатки, которые могут проявиться в конкретной задаче.

### 3 Описание созданного генетического алгоритма

Как уже отмечалось ранее основными этапами любого генетического алгоритма являются процессы кроссинговера, мутации, естественного отбора и проверки критерия останова алгоритма. Предлагаемый алгоритм включает в себя все описанные процессы, которые реализованы с учетом рассматриваемой проблемы.

Основную идею алгоритма можно изложить следующим образом:

1. на начальном этапе есть некоторое множество вершин, заданных текущей популяцией, при этом искомая центральная вершина (или вершина близкая к центральной) находится на одном из путей между парой вершин в популяции. Текущую популяцию можно представить как некую «сферу» (см. рисунок 1), нарисованную множеством вершин, внутри которой находится искомый ответ;
2. далее, основываясь на идеи о том, что искомый ответ лежит на пути между вершинами популяции, на каждой итерации своеобразная «сфера» начинает сжиматься, сужая круг поиска искомой вершины и приводя к сходимости алгоритма (см. рисунок 2);
3. алгоритм может закончить свою работу на итерации, когда расстояние между вершинами не станет равным 0 или 1.

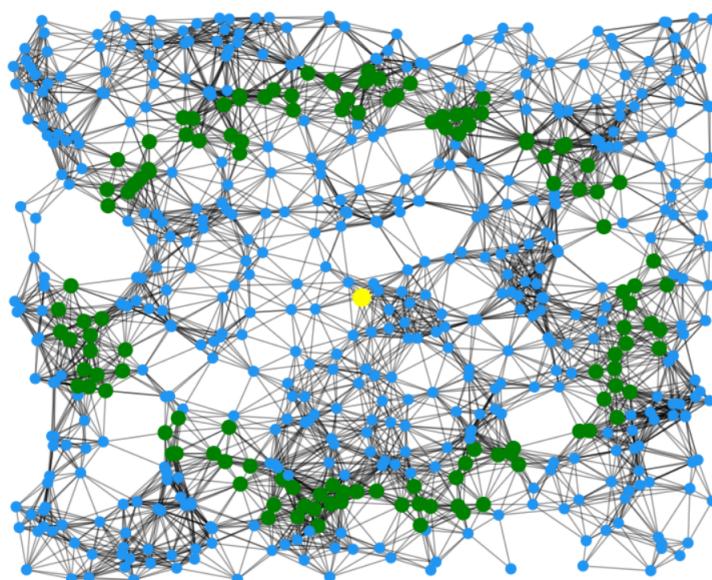


Рисунок 1 – Начальное состояние алгоритма (желтый цвет – центральная вершина, зеленый цвет – начальная популяция)

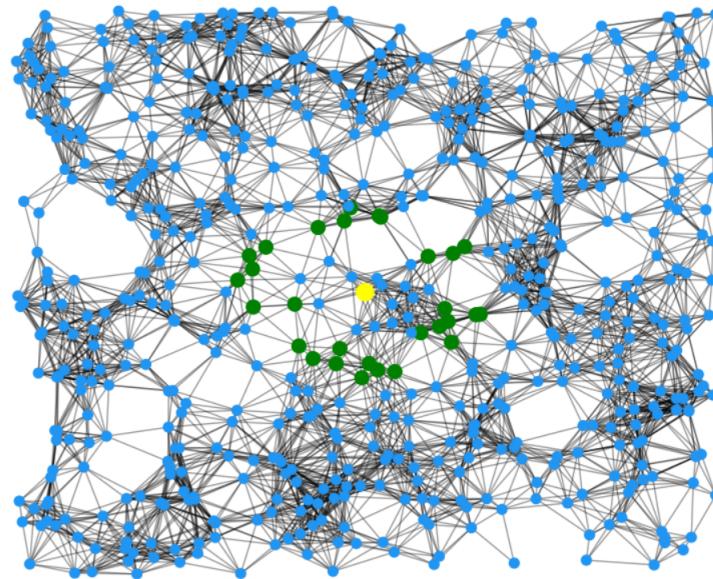


Рисунок 2 – Сходимость алгоритма (желтый цвет – центральная вершина, зеленый цвет – начальная популяция)

### **3.1 Инициализация начальной популяции**

Для старта алгоритма необходимо выбрать начальное множество вершин, которое будет описывать начальную популяцию. Учитывая тот факт, что в предложенном алгоритме не учитываются никакие данные о свойствах графа, то начальная популяция генерируется случайным образом с равновероятным выбором вершины.

### **3.2 Естественный отбор**

Так как основной целью алгоритма является нахождение вершины с минимальным эксцентриситетом, то в качестве целевой функции как раз и выступает эта характеристика вершины. Для каждой вершины в популяции находится ее эксцентриситет, так как граф невзвешенный, то данную процедуру можно выполнить с помощью классического алгоритма обхода в ширину и найти расстояния от вершины до всех остальных (см. рисунок 3). При этом приоритет для попадания в следующее поколение отдается вершинам с меньшим эксцентриситетом.

### **3.3 Процесс кроссинговера**

Каждая вершина представляет потенциальное решение, исходя из предположения, что центральные вершины находятся внутри «сферы», определенной существующей популяцией, то для улучшения решения случайным обра-

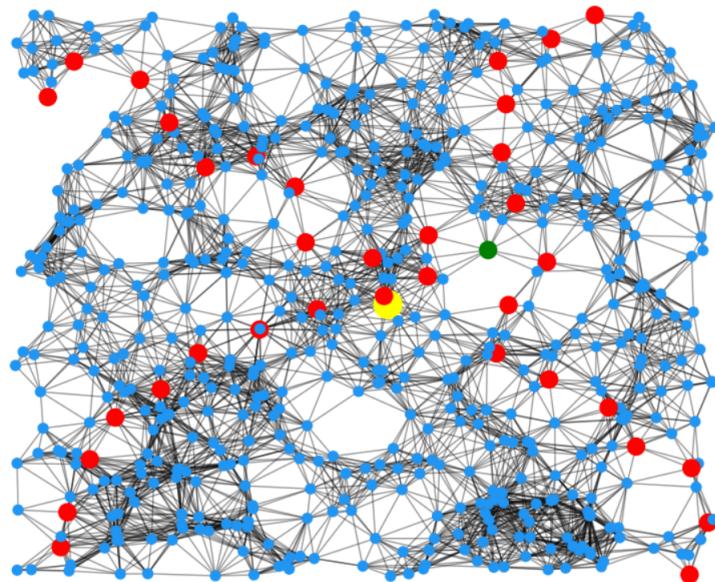


Рисунок 3 – Процесс поиска самой удаленной вершины (желтый цвет – центральная вершина, зеленый цвет – начальная популяция, красный – вершины на пути в самые удаленные точки графа)

зом выбираются две вершины, после чего между ними находится кратчайший путь, из которого в следующее поколение добавляется вершина, находящаяся на половине расстояния от выбранных вершин (см. рисунок 4). Данная идея позволяет так называемой «сфере» сжиматься на каждой итерации и уменьшать расстояние между вершинами внутри популяции.

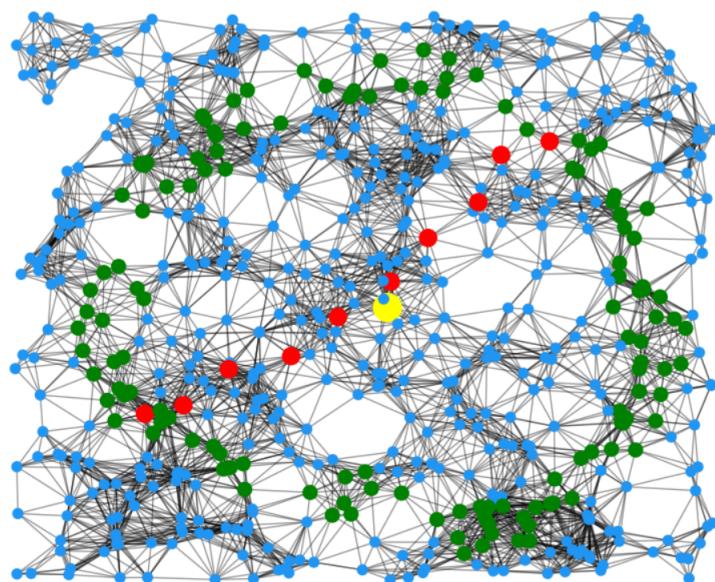


Рисунок 4 – Процесс скрещивания (желтый цвет – центральная вершина, зеленый цвет – начальная популяция, красный – кратчайший путь между вершинами популяции)

### 3.4 Процесс мутации

Для того, чтобы алгоритм в процессе своей работы не сходился к некоторым локальным решениям, а имел возможность рассмотреть новые варианты развития популяции, на каждой итерации в рамках процесса мутации у всех вершин в популяции выбирается случайным вершина смежная с данной и с вероятностью, заданной, параметром мутации переходит в следующее поколение (см. рисунок 5). Данный процесс помогает алгоритму добавлять в новую популяцию «свежие» вершины, которые могут привести к потенциально лучшему решению.

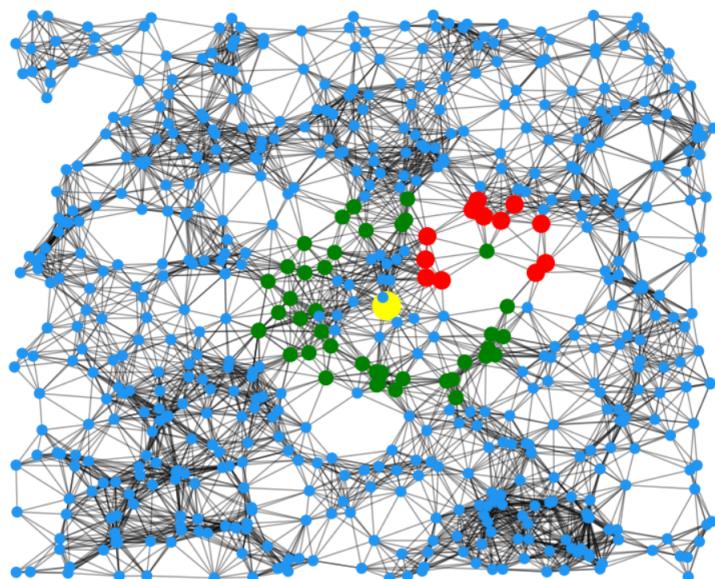


Рисунок 5 – Процесс мутации (желтый цвет – центральная вершина, зеленый цвет – начальная популяция, красный – соседние вершины одной из вершин в популяции)

## **4 Реализация алгоритма**

Для построения предложенного алгоритма был выбран язык программирования C++. В качестве среды разработки, из которой производился запуск проекта была использована Visual Studio 2017, которая была установлена на персональном компьютере с процессором AMD A8-7410 с тактовой частотой 2.20 GHz и оперативной памятью объемом 6.0 GB.

Из описанных ранее действий, которые происходят внутри генетического алгоритма видно, что основным процессом, который необходимо реализовать, является процесс нахождения эксцентриситета. Для более комфортного взаимодействия алгоритма с моделью графа был создан класс Graph, внутри которого содержится информация о его структуре в виде списка смежности, реализованного с помощью вложенных контейнеров vector из стандартной библиотеки шаблонов языка C++. С помощью поддержания такой структуры графа появляется возможность прямого доступа ко всем соседям любой вершины по ее номеру, при этом нумерация вершин ведется с 0 до  $n - 1$ .

Также очевидно, что, так как процесс нахождения эксцентриситета вершины требует больших временных затрат, то внутри класса поддерживается матрица  $n \times n$ , в которую добавляется каждый раз найденный эксцентриситет. Также поддерживается матрица, в которую добавляется каждый кратчайший путь, который находится в процессе алгоритма. С использованием данного класса реализован, как алгоритм, описанный ранее, так и ряд точных алгоритмов.

Для генерации тестовых графов использовалась библиотека NetworkX языка Python. Тестовые графы хранятся в текстовых файлах, в которых перечислен весь список ребер графа. Исходя из этого был написан класс File, который позволяет считывать из файла существующий граф.

Полный код проекта приведен в приложении А.

## **5 Модели случайного графа**

Для исследования созданного алгоритма и его сравнения с существующими были выбраны несколько моделей случайных графов, которые в большей степени описывают свойства реально существующих графов, встречающихся в прикладных задачах.

### **5.1 Модель случайного графа Эрдеша-Ренъи**

Данная модель представленная в работе [11] является одной из самых простых и базовых моделей случайного графа. Изначально для создания графа задаются два параметра  $n$  — число вершин в графе и  $p$  — вероятность проведения ребра. Далее для построения графа рассматриваются все пары вершин, и между ними проводится неориентированное ребро с вероятностью  $p$ . В данной модели ввиду ее простоты формулировки довольно легко выводятся формулы, зависящие от параметра  $p$  и  $n$ , которые описывают основные характеристики графов, такие как связность, размер максимальной клики, распределение степени вершин и т.д. Поэтому если удается установить, что график в рассматриваемой задаче близок по свойствам с графиком Эрдеша-Ренъи, то можно без труда получить много информации об изучаемом графике.

### **5.2 Модель случайного графа Барабаши-Альберт**

Данная модель случайного графа была предложена в работе [12]. На данный момент описанная модель позволяет создавать так называемые безмасштабные сети — графы, в которых распределение степеней подчиняется степенному закону. Исследования данного подхода показали, что графовые модели, которые описывают взаимосвязи внутри различных самоорганизующихся систем, совпадают с моделью Барабаши-Альбера. К таким сетям относятся ряд графов социальных сетей, сеть Интернет, ряд графовых моделей в природных сетях.

В ходе построения случайного графа поддерживаются два основных принципа, которые главным образом характеризуют безмасштабные сети. Первый из них принцип расширения сети, второй — предпочтительное прикрепление. Первый принцип описывается тем фактом, что в существующую сеть могут постоянно добавляться новые узлы, при этом не нарушая его свойств из-за второго принципа. Принцип предпочтительного прикрепления заключается в том, что добавляемый узел случайнным образом прикрепляется ребрами к

вершинам, которые уже существуют в графе, при этом предпочтение отдается вершинам с большей степенью, формально вероятность проведения ребра к  $i$ -му узлу описывается следующий формулой:

$$p_i = \frac{k_i}{\sum_j k_j},$$

где  $k$  — степень узла,  $j$  пробегает все вершины в графе.

Для создания графа задаются два параметра  $n$  — число вершин в создаваемом графе, и  $m$  — число ребер, которые проводится из каждой новой добавляемой вершины в граф. Алгоритм создания достаточно прост, в качестве начального графа берется связный граф с числом вершин большим или равным  $m$ , после чего добавляются новые вершины до необходимого количества, при этом каждая новая вершина соединяется с  $m$  вершинами случайным образом с вероятностным распределением, описанным формулой выше.

### 5.3 Геометрический случайный граф

Данная модель случайного графа [13] описывает основные свойства, которые возникают в графовых моделях компьютерных сетей и сетей, имеющих явную географическую интерпретацию. Для построения графа выбирается размерность пространства, в котором будет создаваться граф, наиболее часто выбирается двумерное пространство, на котором случайным образом генерируется набор геометрических точек равных по количеству числу узлов в создаваемом графе, после чего для каждой пары точек рассчитывается евклидово расстояние между ними и проводится ребро в том случае, если расстояние меньше параметра  $r$ , который задается заранее.

Данная модель случайного графа имеет свои отличительные характеристики, которые не совпадают с моделями Эрдеша-Ренъи и Барабаши-Альберта.

## 6 Результаты вычислительных экспериментов

Для выявления сильных и слабых сторон предложенного алгоритма его сравнение проводилось с рядом точных алгоритмов, а также с алгоритмом «N4N». В качестве тестовых графов были выбраны графовые модели описанные ранее. Для модели Эрдеша-Ренъи в качестве параметра  $p$  было выбрано значение 5%, для модели Барабаши-Альберта  $m = 2$ , а для геометрического случайного графа  $r = 0.1$ .

Чтобы лучше понять механизм алгоритма, природу его работы и определить оптимальные параметры для его настройки, такие как число итераций, размер популяции вероятность мутации и скрещивания были сделаны пошаговые визуализации каждой итерации алгоритма на случайному геометрическому графе (см. рисунок 6).

Для сравнения по временным результатам были реализованы тривиальный алгоритм и алгоритм с улучшенной асимптотикой. Эти алгоритмы и описанный в данной работе запускались на трех моделях случайных графов, с количеством вершин 500, 1000, 1500, 2000, 2500, 5000. Исходя из практических экспериментов в качестве размера популяции выбрано значение 50, для оператора скрещивания 0.7, для оператора мутации 0.1, кроме этого число итераций ограничено числом 20. Результаты временных измерений приведены на графиках 7. Из полученных результатов видно, что созданный генетический алгоритм дает выигрыш по времени в несколько раз по сравнению с точными алгоритмами.

Также так как эвристический алгоритм не гарантирует получение точного ответа, а допускает некий процент ошибки, то для изучения точности алгоритма были проведены тесты позволяющие выявить процент неправильных ответов. Для этого алгоритм запускался на все тех же графах, при этом так как размерности графа, позволяют за приемлемые временные затраты с помощью точного алгоритма найти центральные вершины, то зная эту информацию можно говорить о проценте неправильных ответов. Для сравнения брался алгоритм «N4N», после чего оба алгоритма запускались 100 раз, что позволило подсчитать процент ошибки. Результаты вычислительных экспериментов приведены в таблицах 1, 2 и 3.

Из полученных результатов видно, что созданный алгоритм не уступает существующему эвристическому алгоритму. Предложенный алгоритм превос-

Таблица 1 – Время работы и процент ошибки алгоритмов на случайных геометрических графах

№	Размер графа		Время, сек.		Ошибка, %	
	N	M	Созданный алг.	N4N алг.	Созданный алг.	N4N алг.
1	500	3572	0.11	0.39	38.0	40.0
2	1000	14202	0.31	0.77	21.0	50.0
3	1500	31861	0.71	1.44	13.0	62.0
4	2000	57438	1.20	1.92	8.0	48.0
5	2500	90268	1.76	2.68	4.0	30.0
6	5000	358553	4.48	8.61	0.0	0.0
7	10000	1439255	13.54	26.0	0.0	0.0

Таблица 2 – Время работы и процент ошибки алгоритмов на модели случайного графа Барабаши-Альберта

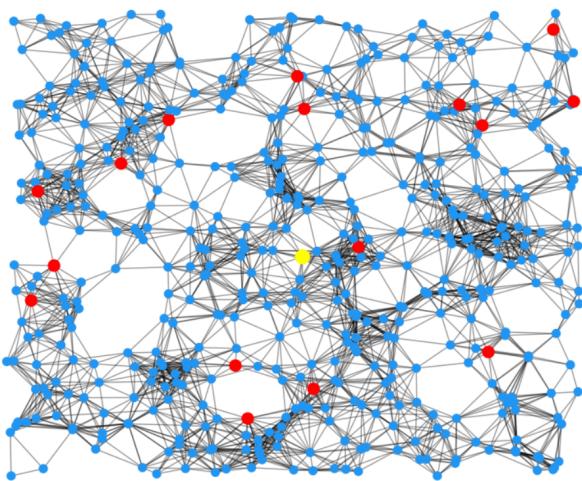
	Размеры графа		Время, сек.		Ошибка, %	
	N	M	Созданный алг.	N4N алг.	Созданный алг.	N4N алг.
1	500	996	0.07	0.29	16.0	0.0
2	1000	1996	0.18	0.68	12.0	0.0
3	1500	2996	0.37	1.24	4.0	0.0
4	2000	3996	0.55	1.67	1.0	0.0
5	2500	4996	0.69	2.18	0.0	0.0
6	5000	9996	1.84	8.28	0.0	0.0
7	10000	19996	3.90	15.8	0.0	0.0

Таблица 3 – Время работы и процент ошибки алгоритмов на модели случайного графа Эрдеша-Ренни

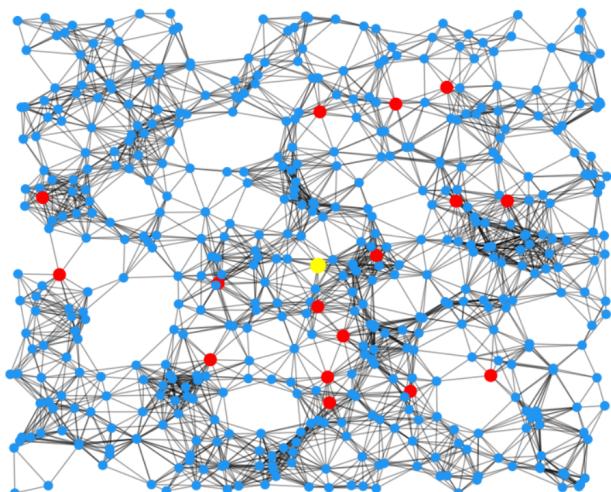
	Размеры графа		Время, сек.		Ошибка, %	
	N	M	Созданный алг.	N4N алг.	Созданный алг.	N4N алг.
1	500	1288	0.16	0.44	29.0	43.0
2	1000	4905	0.60	1.30	0.0	0.0
3	1500	11153	1.44	1.90	0.0	0.0
4	2000	20201	2.17	2.79	0.0	0.0
5	2500	31187	2.68	2.94	0.0	0.0
6	5000	124658	8.47	11.0	0.0	0.0
7	10000	500471	25.4	39.3	0.0	0.0

ходит второй генетический алгоритм в несколько раз. Это во многом объясняется тем, что в алгоритме «N4N» используется множество для описания одной особи в популяции, а также при процессе мутации для вершин, которые претендуют на изменение особи, находится эксцентриситет, всех этих процессов нет в созданном алгоритме, поэтому его время работы меньше. При этом видно, что алгоритм «N4N» на некоторых моделях случайных графов имеет

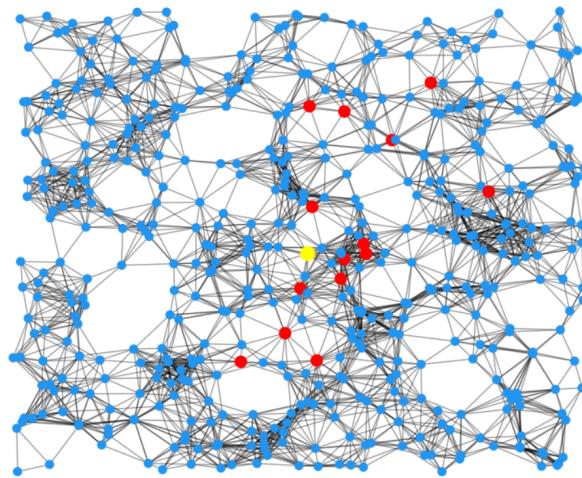
меньший процент ошибки по сравнению с предложенным алгоритмом, однако этот процент нивелируется с увеличением размерности графа.



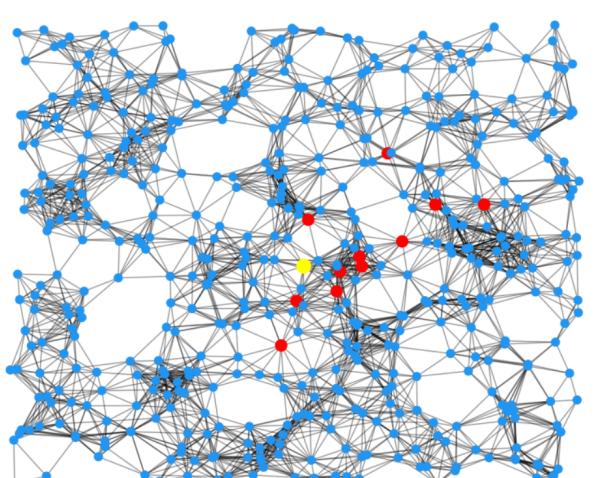
Начальная популяция



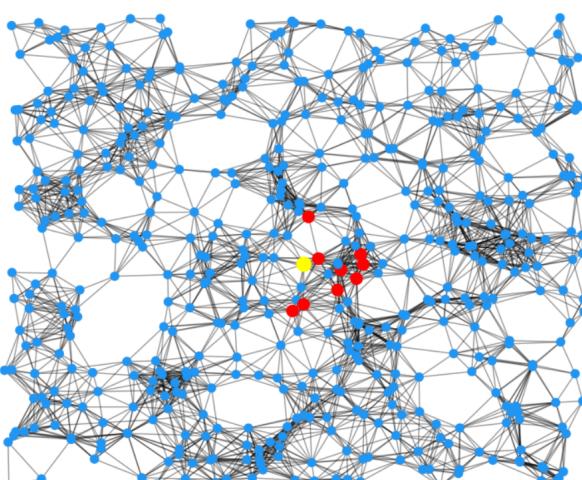
Популяция после двух шагов



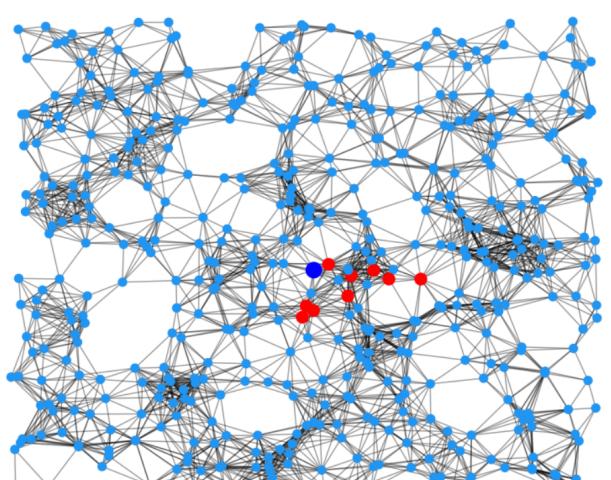
Популяция после четырех шагов



Популяция после шести шагов



Популяция после восьми шагов



Популяция после десяти шагов

Рисунок 6 – Шаги работы алгоритма (желтый цвет – центральная вершина, красный – популяция, темно синий – особь, представляющая правильный ответ)

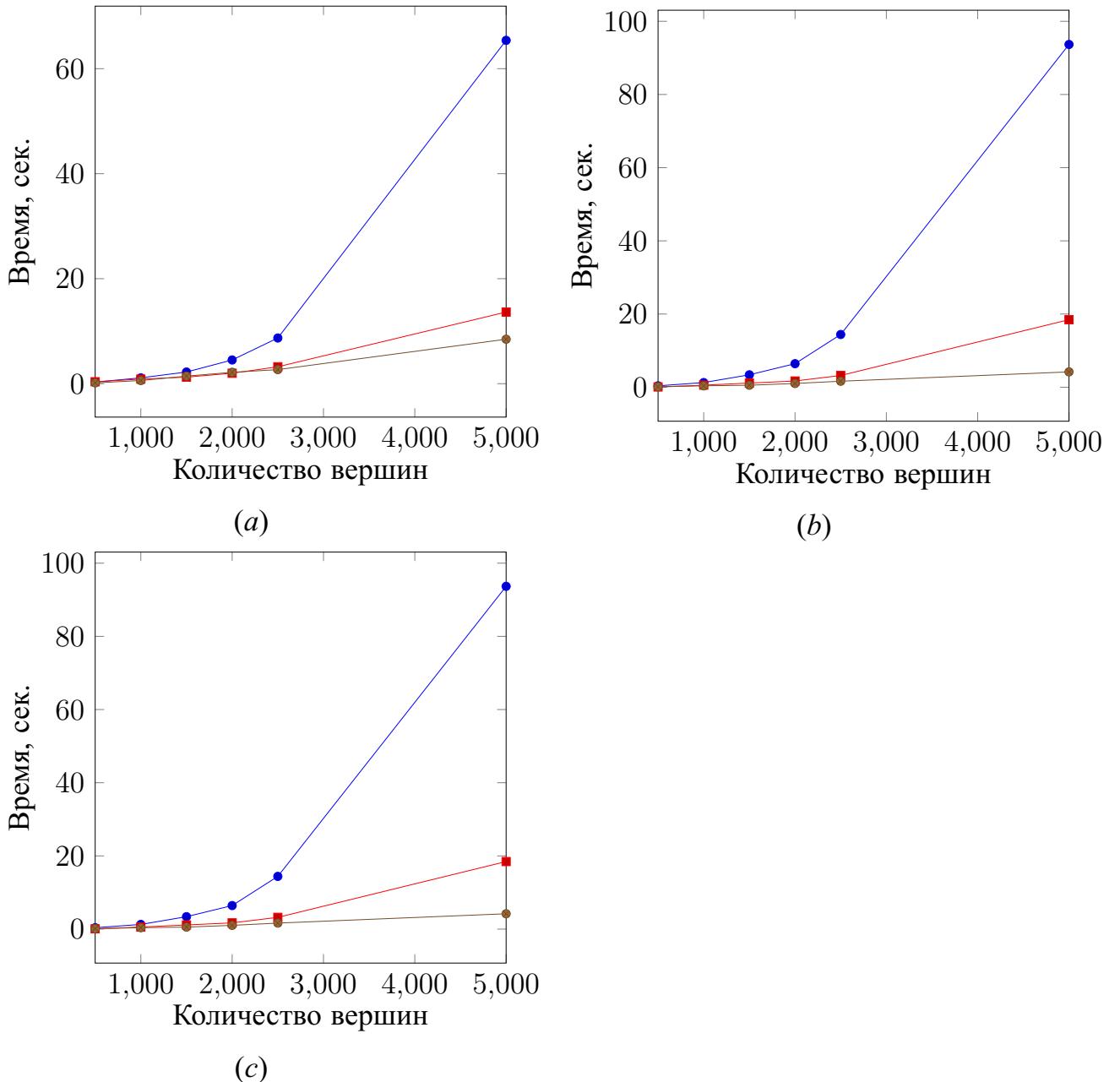


Рисунок 7 – Графики зависимости времени работы от размеров графа (синий цвет – тривиальный  $O(nm + n^2)$  алгоритм, красный – алгоритм с улучшенной асимптотикой  $O(m\sqrt{n})$ , коричневый – генетический алгоритм): (a) – модель Эрдеша-Ренъи  $p = 1\%$ , (b) – модель Барабаши-Альберта  $m = 2$ , (c) – геометрический случайный граф  $r = 0.1$

## **ЗАКЛЮЧЕНИЕ**

В рамках проделанной работы был предложен генетический алгоритм для поиска центральных вершин графов. Был проведен его анализ с различными классическими алгоритмами, а так же с другим генетическим алгоритмом. В ходе анализа было выявлено, что данный алгоритм имеет незначительные недостатки в точности получаемых ответов на графах малой размерности, при этом его целесообразно использовать на больших графовых моделях, где его временные затраты и процент точности оптимальны.

## СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

- 1 Network Analysis / Ed. by U. Brandes, T. Erlebach. — Springer Berlin Heidelberg, 2005. <https://doi.org/10.1007%2Fb106453>.
- 2 Watts, D. J. Collective dynamics of ‘small-world’ networks / D. J. Watts, S. H. Strogatz // *Nature*. — jun 1998. — Vol. 393, no. 6684. — Pp. 440–442. <https://doi.org/10.1038%2F30918>.
- 3 Seidel, R. On the all-pairs-shortest-path problem in unweighted undirected graphs / R. Seidel // *J. Comput. Syst. Sci.* — 1995. — Vol. 51, no. 3. — Pp. 400–403.
- 4 Strassen, V. Gaussian elimination is not optimal / V. Strassen // *Numerische Mathematik*. — Aug 1969. — Vol. 13, no. 4. — Pp. 354–356. <https://doi.org/10.1007/BF02165411>.
- 5 Aingworth, D. Fast estimation of diameter and shortest paths (without matrix multiplication) / D. Aingworth, C. Chekuri, P. Indyk, R. Motwani // *SIAM J. Comput.* — 1999. — Vol. 28, no. 1. — Pp. 1167–1181.
- 6 Chan, T. M. All-pairs shortest paths for unweighted undirected graphs in  $o(mn)$  time / T. M. Chan // *ACM Trans. Algorithms*. — oct 2012. — Vol. 8, no. 4. — Pp. 34:1–34:17. <http://doi.acm.org/10.1145/2344422.2344424>.
- 7 Berman, P. Faster approximation of distances in graphs // Algorithms and Data Structures / Ed. by F. Dehne, J.-R. Sack, N. Zeh. — Berlin, Heidelberg: Springer Berlin Heidelberg, 2007.
- 8 Roditty, L. Fast approximation algorithms for the diameter and radius of sparse graphs // Proceedings of the Forty-fifth Annual ACM Symposium on Theory of Computing. — STOC ’13. — New York, NY, USA: ACM, 2013. — Pp. 515–524. <http://doi.acm.org/10.1145/2488608.2488673>.
- 9 Holland, J. Adaption in Natural and Artificial Systems Adaption in Natural and Artificial Systems / J. Holland. — University of Michigan Press, 1975.
- 10 Alkhalfah, Y. A genetic algorithm applied to graph problems involving subsets of vertices // Proceedings of the 2004 Congress on Evolutionary Computation (IEEE Cat. No.04TH8753). — Vol. 1. — June 2004. — Pp. 303–308.

- 11 *Erdös, P.* On random graphs I / P. Erdös, A. Rényi // *Publicationes Mathematicae Debrecen*. — 1959. — Vol. 6. — P. 290.
- 12 *Albert, R.* Statistical mechanics of complex networks / R. Albert, A.-L. Barabasi // *Reviews of Modern Physics*. — 2002. — Vol. 74, no. 1. — Pp. 47–97.
- 13 *Gilbert, E.* Random plane networks / E. Gilbert // *Journal of the Society for Industrial and Applied Mathematics*. — 1961. — Vol. 9, no. 4. — Pp. 533–543.

## ПРИЛОЖЕНИЕ А

### Листинг проекта

Ниже представлен код созданного класса Graph:

```
1 #pragma once
2 #include <vector>
3 #include <queue>
4 #include <iostream>
5 #include <algorithm>
6 #include <stdio.h>
7 #include <string>
8
9 using std :: vector ;
10 using std :: queue;
11 using std :: pair ;
12 using std :: exception ;
13 using std :: string ;
14 using std :: to_string ;
15 using std :: cin ;
16 using std :: cout ;
17 using std :: endl ;
18
19 using std :: max;
20
21 class Graph {
22 private :
23     int N, M;
24     vector<vector<int>> adjacency;
25     vector<vector<int>> distance ;
26     vector<vector<vector<int>>> path;
27     vector<bool> calculatedEccentricity ;
28     vector<int> eccentricity ;
29     vector<bool> subGraph;
30     bool subGraphMode = false;
31     void initializeMatrix () {
32         adjacency . resize ( this ->N, vector<int>());
33         distance . resize ( this ->N, vector<int>(this ->N, INT_MAX));
34         path . resize ( this ->N, vector<vector<int>>(this ->N));
35         calculatedEccentricity . resize ( this ->N, false);
36         eccentricity . resize ( this ->N, INT_MAX);
37
38         for ( int i = 0; i < this ->N; i++) {
```

```

39         this ->distance[i][i] = 0;
40         this ->path[i][i] = vector<int>(1, i);
41     }
42
43
44 }
45
46 public:
47 Graph(int n, int m, vector<pair<int, int>> adj) {
48     this ->N = n, this ->M = m;
49     initializeMatrix();
50     for (int i = 0; i < m; i++) {
51         int x = adj[i].first, y = adj[i].second;
52         this ->adjacency[x].push_back(y);
53         this ->adjacency[y].push_back(x);
54     }
55 }
56
57 void bfsFromVertex(int x) {
58     if (x < 0 || x >= this ->N) {
59         throw exception("Vertex doesn't exist");
60     }
61     if (this ->subGraphMode && !this ->subGraph[x]) {
62         throw exception("Graph is subgraph mode and vertex isn't in
63             subgraph");
64     }
65     queue<int> q;
66     q.push(x);
67     vector<int> d(this ->N, INT_MAX);
68     d[x] = 0;
69     while (!q.empty()) {
70         int u = q.front();
71         q.pop();
72         vector<int> currentPath = this ->path[x][u];
73         for (unsigned int i = 0; i < this ->adjacency[u].size(); i++) {
74             int v = this ->adjacency[u][i];
75             if (this ->subGraphMode && !this ->subGraph[v]) {
76                 continue;
77             }
78             if (d[v] > INT_MAX / 2) {
79                 currentPath.push_back(v);
80             }
81         }
82     }
83 }
```

```

79             this->path[x][v] = currentPath ;
80             this->path[v][x] = currentPath ;
81             currentPath.pop_back();
82             d[v] = d[u] + 1;
83             q.push(v);
84         }
85     }
86 }
87 for (unsigned int i = 0; i < d.size(); i++) {
88     this->distance[x][i] = d[i];
89     this->distance[i][x] = d[i];
90 }
91 }
92
93 void printDistance () {
94     cout << '\t';
95     for (int i = 0; i < this->N; i++) {
96         cout << i << '\t';
97     }
98     cout << std::endl;
99     for (int i = 0; i < this->N; i++) {
100        cout << i << '\t';
101        for (int j = 0; j < this->N; j++) {
102            if (this->distance[i][j] > INT_MAX / 2) {
103                cout << "INF";
104            }
105            else {
106                cout << this->distance[i][j];
107            }
108            cout << '\t';
109        }
110        cout << std::endl;
111    }
112 }
113
114 void printPath (int x, int y) {
115     if (this->path[x][y].empty())
116         return;
117     cout << x << " ~~~> " << y << ": ";
118     for (unsigned int i = 0; i < this->path[x][y].size(); i++) {
119

```

```

120             cout << path[x][y][ i ] << " ";
121         }
122         cout << std :: endl;
123     }
124
125     vector<int> getPath( int x, int y ) {
126         if ( this ->path[x][y].empty() ) {
127             this ->bfsFromVertex(x);
128         }
129         return this ->path[x][y];
130     }
131
132     int getEccentricity ( int x ) {
133         if ( x >= this ->N) {
134             throw exception("Vertex doesn't exist in graph");
135         }
136
137         if ( ! this ->calculatedEccentricity [x] ) {
138             this ->bfsFromVertex(x);
139             this ->calculatedEccentricity [x] = true ;
140             int result = -INT_MAX;
141             for ( int i = 0; i < this ->N; i++ ) {
142                 result = max(this->distance[x][ i ], result );
143             }
144             this ->eccentricity [x] = result ;
145         }
146         return this ->eccentricity [x];
147     }
148
149     vector<int> getNeighbour(int x) {
150         return this ->adjacency[x];
151     }
152
153     int Size() {
154         return this ->N;
155     }
156
157     void setSubGraph(vector<int> v) {
158         this ->subGraph.resize( this ->N);
159         for ( unsigned int i = 0; i < v.size () ; i++ ) {
160             this ->subGraph[v[i]] = true ;

```

```

161         }
162         this ->subGraphMode = true;
163     }
164     void resetSubGraph() {
165         this ->subGraph.clear();
166         this ->subGraphMode = false;
167     }
168     int getDistance(int u, int v) {
169         return this ->distance[u][v];
170     }
171
172     // This can make some error
173     void setDistance(int u, int v, int d) {
174         this ->distance[u][v] = d;
175         this ->distance[v][u] = d;
176     }
177
178     void printDistanceMatrix () {
179         for (int i = 0; i < this ->distance.size (); i++) {
180             for (int j = 0; j < this ->distance[i].size () ; j++) {
181                 cout << this ->distance[i][j] << " ";
182             }
183             cout << endl;
184         }
185     }
186     vector<vector<int>> getDistanceMatrix () {
187         return this ->distance;
188     }
189
190 };

```

Далее представлен код, отвечающий за работу генетических и нескольких точных алгоритмов:

```

1 #pragma once
2
3 #include <vector>
4 #include <algorithm>
5 #include <set>
6 #include <time.h>
7
8 #include "GraphWork.h"
9 #include "Random.h"

```

```

10 #include "Strassen.h"
11
12
13 using std :: begin;
14 using std :: end;
15 using std :: max_element;
16 using std :: min_element;
17 using std :: pair ;
18 using std :: vector ;
19 using std :: set ;
20
21 // #define PRINT_GA
22 #define PRINT_TIME
23
24 vector<int> getCentralVertex ( vector<int> e) {
25     int minV = *min_element(e.begin(), e.end());
26     #ifdef PRINT_GA
27         cout << "R = " << minV << " " << "Vertex = ";
28     #endif
29     vector<int> res ;
30     for (int i = 0; i < e.size () ; i++) {
31         if (e[i] == minV) {
32             res .push_back(i);
33         }
34     }
35     return res ;
36 }
37
38 void printMatrix (const vector<vector<int>>& X) {
39     for (int i = 0; i < X.size () ; i++) {
40         for (int j = 0; j < X[i].size () ; j++) {
41             cout << X[i][j] << " ";
42         }
43         cout << endl;
44     }
45     cout << "-----\n";
46 }
47
48 void printVector (const vector<int>& v) {
49     for (int i = 0; i < v.size () ; i++) {
50         cout << v[i] << " ";

```

```

51         }
52         cout << endl;
53     }
54
55     class GeneticAlgorithm {
56     private :
57         Graph* G;
58         int populationSize ;
59         double mutationP, crossP ;
60         vector<int> population ;
61         Random rd;
62
63         void startAlgorithm (double& time) {
64             int stepN = 20;
65             double start = clock () ;
66             for (int i = 0; i < stepN; i++) {
67                 this ->evolutionStep();
68             }
69             double finish = clock () ;
70             string message = "Time of genetic algorithm: ";
71             time = ( finish - start ) / CLOCKS_PER_SEC;
72             #ifdef PRINT_TIME
73                 cout << "" << ( finish - start ) / CLOCKS_PER_SEC << endl;
74             #endif
75         }
76
77
78     public :
79         GeneticAlgorithm(Graph* g, int popSize, double pC, double pM) {
80             this ->G = g;
81             this ->populationSize = popSize;
82             this ->crossP = pC;
83             this ->mutationP = pM;
84             for (int i = 0; i < popSize; i++) {
85                 population .push_back(rand() % g->Size());
86             }
87         }
88
89         void makeSelection() {
90             vector<int> e;
91             for (int i = 0; i < population .size () ; i++) {

```

```

92             e.push_back(G->getEccentricity(population[i]));
93         }
94
95         vector<double> probability ;
96         double maxV = *max_element(e.begin(), e.end()), leftSum = 0;
97         for (int i = 0; i < e.size(); i++) {
98             leftSum += maxV / double(e[i]);
99         }
100        double x = 1.0 / leftSum;
101        for (int i = 0; i < e.size(); i++) {
102            probability .push_back(maxV / double(e[i]) * x);
103        }
104        vector<int> nextPopulation ;
105        for (int i = 0; i < this->populationSize; i++) {
106            nextPopulation .push_back(rd.choice(population , probability ));
107        }
108        this->population = nextPopulation ;
109    }
110
111    void crossing () {
112        vector<int> crossed ;
113        for (int i = 0; i < populationSize ; i++) {
114            if (rd.getRandomLowerOne() < crossP) {
115                int ind1 = rd . randint (0, populationSize - 1);
116                int ind2 = rd . randint (0, populationSize - 1);
117                int u = population [ind1], v = population [ind2];
118                vector<int> path = G->getPath(u, v);
119                crossed .push_back(path[path . size () / 2]);
120            }
121        }
122        this->population.insert (population .end(), crossed .begin(), crossed .end())
123        ;
124    }
125    void mutation() {
126        for (unsigned int i = 0; i < this->population.size(); i++) {
127            if (rd.getRandomLowerOne() < mutationP) {
128                vector<int> n = G->getNeighbour(population[i]);
129                population [i] = rd.choice(n);
130            }
131        }

```

```

132     }
133
134     void printPopulation () {
135         for (int i = 0; i < this->population.size (); i++) {
136             cout << population[i] << " ";
137         }
138         cout << endl;
139     }
140
141     void evolutionStep () {
142         this->crossing();
143         this->makeSelection();
144         this->mutation();
145 #ifdef PRINT_GA
146         this->printPopulation ();
147 #endif
148     }
149
150     int getBestResult (double& time) {
151         this->startAlgorithm(time);
152         vector<int> e;
153         for (int i = 0; i < population . size () ; i++) {
154             int x = G->getEccentricity( population [ i ] );
155             e.push_back(x);
156         }
157         vector<int> ind = getCentralVertex (e);
158 #ifdef PRINT_GA
159         for (int i = 0; i < ind . size () ; i++) {
160             cout << population[ ind[ i ] ] << " ";
161         }
162         cout << endl;
163 #endif
164         return this->G->getEccentricity( population[ ind [ 0 ] ] );
165     }
166
167     vector<int> getPopulation () {
168         return this->population;
169     }
170
171 };
172

```

```

173 class ExactAlgorithm {
174 private :
175     int n, m;
176     vector<pair<int, int>> adj;
177     vector<vector<int>> adjTable;
178
179     vector<vector<int>> APSP(const vector<vector<int>>& A) {
180         bool stopecurtion = true ;
181         cout << "Step into " << endl;
182         for (unsigned int i = 0; i < A.size () ; i++) {
183             for (unsigned int j = 0; j < A.size () ; j++) {
184                 if (i != j && !A[i][j]) {
185                     stopecurtion = false ;
186                     break;
187                 }
188             }
189             if (! stopecurtion ) {
190                 break;
191             }
192         }
193         if ( stopecurtion ) {
194             return A;
195         }
196         vector<vector<int>> Z = A * A;
197         vector<vector<int>> B(A.size() , vector<int>(A.size() , 0));
198         for (int i = 0; i < B.size () ; i++) {
199             for (int j = 0; j < B.size () ; j++) {
200                 if (i != j && (A[i][j] == 1 || Z[i][j] > 0)) {
201                     B[i][j] = 1;
202                 }
203             }
204         }
205         vector<vector<int>> T = APSP(B);
206         vector<vector<int>> X = T * A;
207         vector<int> degree(A.size ());
208         for (int i = 0; i < A.size () ; i++) {
209             int sum = 0;
210             for (int j = 0; j < A.size () ; j++) {
211                 sum += A[i][j];
212             }
213             degree[i] = sum;

```

```

214 }
215 vector<vector<int>> D(A.size(), vector<int>(A.size()));
216 for (int i = 0; i < A.size(); i++) {
217     for (int j = 0; j < A.size(); j++) {
218         if (X[i][j] >= T[i][j] * degree[j]) {
219             D[i][j] = 2 * T[i][j];
220         }
221         else {
222             D[i][j] = 2 * T[i][j] - 1;
223         }
224     }
225 }
226 return D;
227 cout << "Step out" << endl;
228 }
229
230 vector<vector<int>> goodAPSP(const vector<vector<int>>& A) {
231     cout << "Step" << endl;
232     int n = A.size();
233     vector<vector<int>> Z = A * A;
234     vector<int> inner(n);
235     vector<vector<int>> B(n, inner);
236
237     for (int i = 0; i < n; i++) {
238         for (int j = 0; j < n; j++) {
239             if (i != j && (A[i][j] == 1 || Z[i][j] > 0)) {
240                 B[i][j] = 1;
241             }
242             else {
243                 B[i][j] = 0;
244             }
245         }
246     }
247
248     bool goodMatrix = true;
249     for (int i = 0; i < n; i++) {
250         for (int j = 0; j < n; j++) {
251             if (i != j && !B[i][j]) {
252                 goodMatrix = false;
253             }
254         }

```

```

255 }
256 vector<vector<int>> D(n, inner);
257 if (goodMatrix) {
258     for (int i = 0; i < n; i++) {
259         for (int j = 0; j < n; j++) {
260             D[i][j] = 2 * B[i][j] - A[i][j];
261         }
262     }
263     return D;
264 }
265 vector<vector<int>> T = goodAPSP(B);
266 vector<vector<int>> X = T * A;
267 vector<int> degree(n);
268 for (int i = 0; i < n; i++) {
269     int sum = 0;
270     for (int j = 0; j < n; j++) {
271         sum += A[i][j];
272     }
273     degree[i] = sum;
274 }
275
276 for (int i = 0; i < n; i++) {
277     for (int j = 0; j < n; j++) {
278         if (X[i][j] > T[i][j] * degree[j]) {
279             D[i][j] = 2 * T[i][j];
280         }
281         else {
282             D[i][j] = 2 * T[i][j] - 1;
283         }
284     }
285 }
286 return D;
287 }
288
289 vector<int> makeDominatingSet(const vector<vector<int>>& A) {
290     if (A.size() == 0) {
291         return vector<int>();
292     }
293     vector<bool> U(this->n);
294     vector<int> C;
295     vector<bool> usedS(A.size());

```

```

296     while ( true ) {
297         int maxV = 0, index = -1;
298         for ( int i = 0; i < A.size () ; i++) {
299             int current = 0;
300             if ( !usedS[i] ) {
301                 for ( int j = 0; j < A[i].size () ; j++) {
302                     int x = A[i][j];
303                     if ( !U[x] ) {
304                         current++;
305                     }
306                 }
307             }
308             if ( current > maxV ) {
309                 maxV = current;
310                 index = i;
311             }
312         }
313         if ( index == -1 ) {
314             break;
315         }
316         for ( int i = 0; i < A[index].size () ; i++) {
317             U[A[index][i]] = true ;
318         }
319         C.push_back(index);
320     }
321
322     set<int> Union;
323     for ( int i = 0; i < C.size () ; i++) {
324         Union.insert (A[C[i]].begin() , A[C[i]].end());
325     }
326     vector<int> result ;
327     for ( auto i = Union.begin(); i != Union.end(); i++) {
328         result .push_back(*i);
329     }
330     return result ;
331 }
332
333 public :
334     ExactAlgorithm(int n, int m, vector<pair<int, int>> a) {
335         this ->n = n;
336         this ->m = m;

```

```

337     this ->adj = a;
338     this ->adjTable.resize(n);
339     for (unsigned i = 0; i < a.size(); i++) {
340         int u = a[i].first, v = a[i].second;
341         this ->adjTable[u].push_back(v);
342         this ->adjTable[v].push_back(u);
343     }
344 }
345
346 double TrivialAlgorithm () {
347     cout << "Trivial algorithm" << endl;
348     Graph* g = new Graph(n, m, adj);
349     vector<int> e;
350     double start = clock();
351     for (int i = 0; i < n; i++) {
352         int x = g->getEccentricity(i);
353         e.push_back(x);
354     }
355     double finish = clock();
356     double diff = (finish - start) / CLOCKS_PER_SEC;
357     cout << "Time of trivial algorithm: " << diff << endl;
358     printVector (getCentralVertex (e));
359     delete g;
360     return diff;
361 }
362
363 void SeidelsAlgorithm () {
364     cout << "Seidels Algorithm" << endl;
365     vector<int> inner (this ->n);
366     vector<vector<int>> A(this ->n, inner);
367     for (int i = 0; i < adj.size(); i++) {
368         int u = adj[i].first, v = adj[i].second;
369         A[u][v] = 1;
370         A[v][u] = 1;
371     }
372
373     vector<vector<int>> result = APSP(A);
374     vector<int> e (this ->n);
375     for (int i = 0; i < result.size(); i++) {
376         e[i] = *max_element(result[i].begin(), result[i].end());
377     }

```

```

378         printVector ( getCentralVertex (e));
379     }
380
381     void AingworthAlgorithm() {
382         cout << "Aingworth algorithm" << endl;
383         double start = clock();
384         Graph g(n, m, adj);
385         int s = sqrt(n * log(n));
386         vector<int> L;
387         vector<int> H;
388         for (int i = 0; i < n; i++) {
389             int x = g.getNeighbour(i).size();
390             if (x < s) {
391                 L.push_back(i);
392             }
393             else {
394                 H.push_back(i);
395             }
396         }
397         cout << "S = " << s << endl;
398         cout << "L = " << L.size() << endl;
399         cout << "H = " << H.size() << endl;
400         g.setSubGraph(L);
401         for (int i = 0; i < L.size(); i++) {
402             g.bfsFromVertex(L[i]);
403         }
404         g.resetSubGraph();
405         vector<vector<int>> A;
406         for (int i = 0; i < H.size(); i++) {
407             vector<int> Ne = g.getNeighbour(H[i]);
408             Ne.push_back(H[i]);
409             A.push_back(Ne);
410         }
411         cout << "Making domain set" << endl;
412         vector<int> D = makeDominatingSet(A);
413         cout << "Domain set maked" << endl;
414         cout << D.size() << endl;
415         for (int i = 0; i < D.size(); i++) {
416             g.bfsFromVertex(D[i]);
417         }
418         vector<bool> Diff( this->n, true);

```

```

419     for (int i = 0; i < D.size(); i++) {
420         Diff[D[i]] = false ;
421     }
422
423     cout << "Making Table" << endl;
424     vector<vector<int>> d = g.getDistanceMatrix();
425     for (int u = 0; u < n; u++) {
426         if (!Diff[u]) {
427             continue;
428         }
429         for (int v = u + 1; v < n; v++) {
430             if (!Diff[v]) {
431                 continue;
432             }
433             int dUV = d[u][v];
434             int minV = INT_MAX;
435             for (int w = 0; w < D.size(); w++) {
436                 int d1 = d[D[w]][u];
437                 int d2 = d[D[w]][v];
438                 if (d1 + d2 < minV) {
439                     minV = d1 + d2;
440                 }
441             }
442             d[u][v] = min(minV, dUV);
443             d[v][u] = min(minV, dUV);
444         }
445     }
446     double finish = clock();
447     cout << "Time of AingworthAlgorithm: " << (finish - start) /
448         CLOCKS_PER_SEC << endl;
449     vector<int> e;
450     for (int i = 0; i < n; i++) {
451         int x = *max_element(d[i].begin(), d[i].end());
452         e.push_back(x);
453     }
454     printVector( getCentralVertex(e));
455 }

```

1 **#pragma** once

2

3 **#include** <vector>

```

4 #include <set>
5 #include <algorithm>
6 #include "GraphWork.h"
7 #include "Random.h"
8 #include "SetOperation.h"
9 #include "GeneticAlgorithm.h"

10
11
12 using std :: set ;
13 using std :: max_element;

14
15 class SimpleGeneticAlgorithm {
16 private :
17     Graph* G;
18     int populationSize ;
19     int chromoLength;
20     double mutationP, crossP;
21     Random rd;
22     vector<set<int>> population;

23
24     void startAlgorithm (double& time) {
25         int stepN = 20;
26         double start = clock();
27         for (int i = 0; i < stepN; i++) {
28             this ->evolutionStep();
29         }
30         double finish = clock();
31         time = ( finish - start ) / CLOCKS_PER_SEC;
32         string message = "Time of OTHER genetic algorithm: ";
33 #ifdef PRINT_TIME
34         cout << "" << time << endl;
35 #endif
36     }
37
38
39     int function ( vector<int> ch) {
40         int sum = 0;
41         for (auto it = ch.begin(); it != ch.end(); it++) {
42
43             sum += this->G->getEccentricity(*it);
44         }

```

```

45     return sum;
46 }
47
48 int function( set<int> ch) {
49     int sum = 0;
50     for (auto it = ch.begin(); it != ch.end(); it++) {
51         sum += this->G->getEccentricity(*it);
52     }
53     return sum;
54 }
55
56
57 public:
58     SimpleGeneticAlgorithm(Graph* g, int popSize, int chromoLength, double pC,
59     double pM) {
60         this->G = g;
61         this->populationSize = popSize;
62         this->crossP = pC;
63         this->mutationP = pM;
64         this->chromoLength = chromoLength;
65         for (int i = 0; i < popSize; i++) {
66             set<int> chromo;
67             while (chromo.size() < chromoLength) {
68                 int v = rand() % g->Size();
69                 chromo.insert(v);
70             }
71             population.push_back(chromo);
72         }
73     }
74
75     void evolutionStep() {
76         this->makeSelection();
77         this->crossing();
78         this->N4N();
79 #ifdef PRINT_GA
80         for (int i = 0; i < populationSize; i++) {
81             cout << "{ ";
82             for (auto it = population[i].begin(); it != population[i].end(); it++) {
83                 cout << *it << " ";
84             }
85         }
86     }
87 }
```

```

84             cout << "}" << endl;
85         }
86         cout << "----|----|----|----|----" << endl;
87 #endif
88     }
89
90
91     void N4N() {
92         const int COUNT = 4;
93         int percent10 = 0.1 * populationSize ;
94         vector<bool> used( populationSize , false );
95         vector<int> indexChromo;
96         for ( int i = 0; i < percent10; i++) {
97             while ( true ) {
98                 int c = rand() % populationSize;
99                 if ( !used[c] ) {
100                     used[c] = true ;
101                     indexChromo.push_back(c);
102                     break;
103                 }
104             }
105         }
106         for ( int k = 0; k < indexChromo.size(); k++) {
107             vector<int> best = makeVector(population[indexChromo[k]]);
108             vector<int> X = best;
109             int currentBest = function (X);
110             for ( int i = 0; i < X.size () ; i++) {
111                 vector<int> neight = this ->G->getNeighbour(X[i]);
112                 for ( int j = 0, t = 0; j < neight . size () && t < COUNT;
113                     j++) {
114                     auto it = find(X.begin(), X.end(), neight[j]);
115                     if ( it != X.end()) {
116                         continue;
117                     }
118                     int tmp = X[i];
119                     X[i] = neight[j];
120                     int currentFunction = function (X);
121                     if ( currentFunction < currentBest ) {
122                         best = X;
123                         currentBest = currentFunction ;

```

```

124                         }
125                         X[i] = tmp;
126                     }
127                 }
128             population [indexChromo[k]] = makeSet(best);
129         }
130     }
131
132
133     void mutation() {
134         for (int i = 0; i < populationSize ; i++) {
135             set<int> chromo = population[ i ];
136             while ( true ) {
137                 int v = rand() % this->G->Size();
138                 if (chromo.find(v) == chromo.end()) {
139                     chromo.erase(chromo.begin());
140                     chromo.insert (v);
141                     break;
142                 }
143             }
144         }
145     }
146
147     void makeSelection() {
148         vector<int> fitnessFunction ;
149
150         for (int i = 0; i < this->populationSize; i++) {
151             int sum = 0;
152             set<int> chromo = this->population[i];
153             for (auto it = chromo.begin(); it != chromo.end(); it++) {
154                 sum += G->getEccentricity(*it);
155             }
156             fitnessFunction .push_back(sum);
157         }
158
159         vector<double> probability ;
160         double maxV = *max_element(fitnessFunction.begin(), fitnessFunction .end()
161             ), leftSum = 0;
162         for (int i = 0; i < fitnessFunction .size () ; i++) {
163             leftSum += maxV / double(fitnessFunction [i]);
164         }

```

```

164     double x = 1.0 / leftSum;
165     for (int i = 0; i < fitnessFunction .size (); i++) {
166         probability .push_back(maxV / double(fitnessFunction [i]) * x);
167     }
168
169     vector<set<int>> nextPopulation ;
170     for (int i = 0; i < this->populationSize; i++) {
171         nextPopulation .push_back(rd.choice(population , probability ));
172     }
173     this->population = nextPopulation ;
174 }
175
176
177
178
179
180 void crossing () {
181     for (int i = 0; i < this->populationSize / 2; i++) {
182         if (rd.getRandomLowerOne() >= crossP) {
183             continue;
184         }
185         int i1 = rand() % populationSize;
186         int i2 = rand() % populationSize;
187         set<int> vp1 = population [i1] - population [i2];
188         set<int> vp2 = population [i2] - population [i1];
189         if (vp1.size () == 0 || vp2.size () == 0) {
190             continue;
191         }
192         if (vp1.size () != vp2.size ()) {
193             throw exception("Зря ты так думал");
194         }
195
196         int r = rand() % vp1.size ();
197         vector<int> v1 = makeVector(population [i1]);
198         vector<int> v2 = makeVector(population [i2]);
199         vector<int> xch1 = makeVector(vp2);
200         vector<int> xch2 = makeVector(vp1);
201
202         for (int j = 0; j < r; j++) {
203             v1[rand() % v1.size ()] = xch1[j];
204             v2[rand() % v2.size ()] = xch2[j];

```

```

205
206         }
207         population[i1] = makeSet(v1);
208         population[i2] = makeSet(v2);
209     }
210
211     int getBestResult(double& time) {
212         this->startAlgorithm(time);
213         int index = 0, minSum = INT_MAX;
214         for (int i = 0; i < this->populationSize; i++) {
215             int sum = 0;
216             for (auto it = population[i].begin(); it != population[i].end();
217                 it++) {
218                 sum += this->G->getEccentricity(*it);
219             }
220             if (sum < minSum) {
221                 minSum = sum, index = i;
222             }
223             vector<int> bestChromo = makeVector(population[index]);
224             vector<int> e;
225             for (int i = 0; i < bestChromo.size(); i++) {
226                 e.push_back(G->getEccentricity(bestChromo[i]));
227             }
228             vector<int> bestIndex = getCentralVertex(e);
229 #ifdef PRINT_GA
230             for (int i = 0; i < bestIndex.size(); i++) {
231                 cout << bestChromo[bestIndex[i]] << " ";
232             }
233             cout << endl;
234 #endif
235             return this->G->getEccentricity(bestChromo[bestIndex[0]]);
236         }
237     };

```

В следующем фрагменте кода приведена реализация класса для работы со случайными числами и для файлового ввода графов:

- 1 #pragma once
- 2 #include <vector>
- 3 #include <string>
- 4 #include <fstream>
- 5 #include <iostream>

```

6  #include <algorithm>
7
8  using std :: vector ;
9  using std :: pair ;
10 using std :: make_pair;
11 using std :: string ;
12 using std :: ifstream ;
13 using std :: exception ;
14 using std :: getline ;
15 using std :: stringstream ;
16 using std :: max;
17
18 vector<pair<int, int>> readFromFileWhereEdges(string file , int& N, int& M) {
19     ifstream in( file );
20     if (!in) {
21         throw exception("Проблема с файлом");
22     }
23     vector<pair<int, int>> e;
24     string inputLine;
25     int n = 0, m = 0;
26     while (getline (in, inputLine)) {
27         if (inputLine[0] == '%') {
28             continue;
29         }
30         m++;
31         stringstream s(inputLine);
32         int x, y;
33         s >> x >> y;
34         x--, y--;
35         n = max(n, max(x, y));
36         e.push_back(make_pair(x, y));
37     }
38     N = n;
39     M = m;
40     return e;
41 }
42
43 vector<pair<int, int>> readFromFileNM(string file , int& N, int& M) {
44     ifstream in( file );
45     if (!in) {
46         throw exception("Проблема с файлом");

```

```

47     }
48     vector<pair<int, int>> e;
49     int n, m;
50     in >> n >> m;
51     for (int i = 0; i < m; i++) {
52         int x, y;
53         in >> x >> y;
54         e.push_back(make_pair(x, y));
55     }
56     N = n;
57     M = m;
58     return e;
59 }
```

```

1 #pragma once
2 #include <vector>
3 #include <ctime>
4 #include <algorithm>
5
6 using std::vector;
7 using std::exception;
8 using std::max;
9 using std::min;
10
11 class Random {
12 private :
13     static const int a = 10'000;
14 public :
15     Random() {
16         srand(time(NULL));
17     }
18
19     int choice(vector<int> v) {
20         int i = rand() % int(v.size());
21         return v[i];
22     }
23
24     static double getRandomLowerOne() {
25         double x = rand() % a;
26         return x / double(a);
27     }
28 }
```

```

29         template<typename T>
30             T choice( vector<T> v, vector<double> d) {
31                 if (v.size () != d.size ()) {
32                     throw exception("The size of vectors must be equal");
33                 }
34                 double x = getRandomLowerOne();
35                 for ( int i = 0; i < v.size () ; i++) {
36                     x -= d[i];
37                     if (x <= 0) {
38                         return v[i];
39                     }
40                 }
41                 return v[v.size () - 1];
42             }
43
44             static int randint( int a, int b) {
45                 int maxV = max(a, b), minV = min(a, b);
46                 int x = rand() % (maxV - minV + 1);
47                 return minV + x;
48             }
49
50 };

```

В последнем листинге приведена главная функция из которой запускался весь проект:

```

1 #include <iostream>
2 #include "GraphWork.h"
3 #include "Random.h"
4 #include "GeneticAlgorithm.h"
5 #include "OtherGA.h"
6 #include "FileWork.h"
7
8 using std :: make_pair;
9 using std :: ifstream ;
10
11
12 // #define GEOM_GRAPH
13 // #define BA_GRAPH
14 #define ERDOSH_GRAPH
15
16 #define SIMPLE_GA
17 // #define OUR_GA

```

```

18
19 const string BA_FILE[] = { "BA_Graph\\BarabasiAlbertGraph1_M2.txt",
20                               "BA_Graph\\BarabasiAlbertGraph2_M2.txt",
21                               "BA_Graph\\BarabasiAlbertGraph3_M2.
22                               txt",
23                               "BA_Graph\\BarabasiAlbertGraph4_M2.
24                               txt",
25                               "BA_Graph\\BarabasiAlbertGraph5_M2.
26                               txt",
27                               "BA_Graph\\BarabasiAlbertGraph6_M2.
28                               txt",
29                               "BA_Graph\\BarabasiAlbertGraph7_M2.
30                               txt" };

31
32 const string GEOM_FILE[] = { "GEOM_Graph\\GeometricGraph1_R01.txt",
33                               "GEOM_Graph\\GeometricGraph2_R01
34                               .txt",
35                               "GEOM_Graph\\GeometricGraph3_R01
36                               .txt",
37                               "GEOM_Graph\\GeometricGraph4_R01
38                               .txt",
39                               "GEOM_Graph\\GeometricGraph5_R01
40                               .txt",
41                               "GEOM_Graph\\GeometricGraph6_R01
42                               .txt",
43                               "GEOM_Graph\\
44                               GeometricGraph7_R01.txt" };

45
46 const string ER_FILE[] = { "ERDOSRENYI_Graph\\ErdosRenyi1_P001.txt",
47                               "ERDOSRENYI_Graph\\
48                               ErdosRenyi2_P001.txt",
49                               "ERDOSRENYI_Graph\\
50                               ErdosRenyi3_P001.txt",
51                               "ERDOSRENYI_Graph\\
52                               ErdosRenyi4_P001.txt",
53                               "ERDOSRENYI_Graph\\
54                               ErdosRenyi5_P001.txt",
55                               "ERDOSRENYI_Graph\\
56                               ErdosRenyi6_P001.txt",
57                               "ERDOSRENYI_Graph\\
58                               ErdosRenyi7_P001.txt" };

```

```

42
43
44 const int TEST = 5;
45 const int ITER = 10;
46
47 const int RADIUS_BA[] = { 4, 4, 5, 4, 5, 5, 5 };
48 const int RADIUS_GEOM[] = { 9, 9, 8, 8, 8, 8, 8 };
49 const int RADIUS_ER[] = { 5, 4, 4, 4, 3, 3, 3 };

50
51 int main() {
52     srand(time(NULL) % INT_MAX);
53     int n, m;
54 #ifdef BA_GRAPH
55     vector<pair<int, int>> e = readFromFileNM(BA_FILE[TEST], n, m);
56 #endif
57 #ifdef GEOM_GRAPH
58     vector<pair<int, int>> e = readFromFileNM(GEOM_FILE[TEST], n, m);
59 #endif
60
61 #ifdef ERDOSH_GRAPH
62     vector<pair<int, int>> e = readFromFileNM(ER_FILE[TEST], n, m);
63 #endif
64
65     cout << "N = " << n << " M = " << m << endl;
66     ExactAlgorithm alg(n, m, e);
67     double t1 = alg.TrivialAlgorithm();
68     cout << "Aingworth " << badTimeFunction(t1, n, m) << endl;
69
70     // alg.SeidelsAlgorithm();
71     // alg.AingworthAlgorithm();
72 #ifdef BA_GRAPH
73     const int REAL_R = RADIUS_BA[TEST];
74 #endif
75
76 #ifdef GEOM_GRAPH
77     const int REAL_R = RADIUS_GEOM[TEST];
78 #endif
79 #ifdef ERDOSH_GRAPH
80     const int REAL_R = RADIUS_ER[TEST];
81 #endif
82

```

```

83 #ifdef OUR_GA
84 {
85     int error = 0;
86     double avg_time = 0.0;
87     for (int i = 0; i < ITER; i++) {
88         Graph* g1 = new Graph(n, m, e);
89         cout << "Start genetic algorithm" << endl;
90         GeneticAlgorithm genAlg(g1, 50, 0.7, 0.1);
91         double time = 0.0;
92         int R = genAlg.getBestResult(time);
93         avg_time += time;
94         if (R != REAL_R)
95             error++;
96     }
97     cout << "AVG Time = " << avg_time / double(ITER) << endl;
98     cout << "Error = " << double(error) / double(ITER) << endl;
99 }
100#endif
101
102#ifndef SIMPLE_GA
103{
104    int error = 0;
105    double avg_time = 0.0;
106    for (int i = 0; i < ITER; i++) {
107        Graph* g1 = new Graph(n, m, e);
108        cout << "Start OTHER genetic algorithm" << endl;
109        SimpleGeneticAlgorithm sgen(g1, 50, 10, 0.7, 0.1);
110        double time = 0.0;
111        int R = sgen.getBestResult(time);
112        avg_time += time;
113        if (R != REAL_R)
114            error++;
115    }
116    cout << "AVG Time = " << avg_time / double(ITER) << endl;
117    cout << "Error = " << double(error) / double(ITER) << endl;
118 }
119#endif
120}

```

## **ПРИЛОЖЕНИЕ Б**

### **CD с отчетом о проделанной работе**

На приложенном диске можно ознакомиться со следующими файлами:

**Папка Project** — проект с набором тестовых графов;

**Папка CourseWork** — L<sup>A</sup>T<sub>E</sub>Xвариант работы;

CourseWork3.pdf — текст курсовой работы.