

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

Кафедра математической кибернетики и компьютерных наук

**ИССЛЕДОВАНИЕ ПАРАМЕТРОВ ГЕНЕТИЧЕСКОГО АЛГОРИТМА  
ДЛЯ ПОИСКА ЦЕНТРАЛЬНЫХ ВЕРШИН В ГРАФАХ**

**БАКАЛАВРСКАЯ РАБОТА**

студента 4 курса 411 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., доцент \_\_\_\_\_ А. С. Иванов

## СОДЕРЖАНИЕ

<b>ВВЕДЕНИЕ .....</b>	<b>3</b>
1 Описание задачи .....	4
2 Генетические алгоритмы .....	5
2.1 Представление решения и начальная популяция .....	6
2.2 Оператор скрещивания .....	6
2.3 Оператор мутации .....	7
2.4 Оператор естественного отбора .....	7
3 Алгоритмы для поиска центральных вершин .....	8
3.1 Тривиальный алгоритм .....	8
3.2 Алгоритмы с улучшенной асимптотикой .....	8
3.3 Алгоритмы, использующие матричное умножение .....	9
3.4 Существующие генетические алгоритмы .....	9
4 Описание разработанного генетического алгоритма .....	11
4.1 Старт алгоритма .....	12
4.2 Естественный отбор .....	12
4.3 Этап скрещивания .....	12
4.4 Этап мутации .....	13
5 Исследование алгоритма .....	17
6 Модели случайных графов .....	21
6.1 Модель случайного графа Эрдеша-Реньи .....	21
6.2 Модель случайного графа Барабаши-Альберт .....	21
6.3 Геометрический случайный граф .....	22
7 Результаты вычислительных экспериментов .....	23
8 Исследование параметров генетического алгоритма .....	26
9 Описание приложения .....	34
<b>ЗАКЛЮЧЕНИЕ .....</b>	<b>38</b>
<b>СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ .....</b>	<b>39</b>
Приложение А Код генетического алгоритма .....	41
Приложение Б Код классов для работы с графами .....	45
Приложение В Код классов для экспериментов с алгоритмом .....	54

## 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.

Основной целью работы было следующее — разработать и исследовать генетический алгоритм, способный решать задачу поиска центральных вершин. Для достижения этой цели были поставлены следующие задачи:

- реализовать идеи генетических алгоритмов с учетом рассматриваемой задачи,
- реализовать существующие алгоритмы для поиска центральных вершин,
- создать программное приложение, позволяющее проводить запуски алгоритма с различными параметрами,
- исследовать параметры алгоритма и выявить его слабые и сильные стороны.

## 1 Описание задачи

Для описания задачи сначала необходимо дать формальное определение понятию граф. Граф — это упорядоченная пара множеств  $(V, E)$ , где  $V$  — множество вершин графа, а  $E$  — множество упорядоченных и неупорядоченных пар вершин — дуг или ребер. В случае, когда вершины в парах упорядочены, говорят, что граф является ориентированным, иначе — неориентированным.

В случае неориентированного графа также используется понятие связности графа — граф является связным, если между любой парой вершин существует по крайней мере один путь. Кроме этого графы можно разделить на взвешенные или невзвешенные — в случае взвешенного графа каждое ребро имеет некоторый вес — положительное или отрицательное число, в случае невзвешенного графа каждое ребро имеет вес равный единице.

В данной работе предлагается и исследуется задача поиска центральных вершин в графах, поэтому далее приводится формальное описание задачи. Для начала можно дать определение эксцентриситета вершины в графе. Эксцентриситетом  $e$  вершины называется максимальное из расстояний от этой вершины до всех остальных вершин в графе. С использованием этого определения можно сказать, что центральными вершинами в графе называются вершины с минимальным значением эксцентриситета, при чем само значение этого минимального эксцентриситета представляет собой радиус графа.

## **2 Генетические алгоритмы**

Под генетическими алгоритмами принято подразумевать вероятностно-эвристические алгоритмы, которые применяются для решения задач оптимизации. Сфера применения генетических алгоритмов достаточно широка, с одной стороны данные алгоритмы могут быть применены при решении задач оптимизации, в которых недостаточно накопленных математических и алгоритмических знаний ввиду уникальности задачи или ее мало изученности. Кроме этого генетические алгоритмы могут быть применены при решении задач, для которых не существует эффективных алгоритмов решения — задачи из NP-класса, а также подобные алгоритмы могут найти применение при попытках уменьшить временные затраты на решение хорошо изученной задачи.

В основе любого генетического алгоритма лежит моделирование эволюционного развития живых организмов за счет таких факторов, как естественный отбор, мутации и скрещивание. Процесс скрещивания или кроссинговера впервые начал изучаться в XIX веке ученым-ботаником Г. Менделем. В результате его исследований было установлено, что в набор генов живых организмов передаются гены его родителей причем в скомбинированном виде. Этот факт во многом объясняет с одной стороны все многообразие живых существ, а с другой явление передачи полезных свойств через поколения.

В дальнейшем с развитием науки были сделаны ряд открытий, связанных с таким явлением, как мутация генов. Под мутацией понимается изменение генетических участков организма. Чаще всего эти изменения происходят под воздействием внешних факторов или внутренних. Такие мутации чаще всего приводят к негативным последствиям, но вместе с тем у живого организма появляется небольшая возможность получить новые внешние свойства, которые будут выгодно выделять его среди других организмов и переведут на новый виток эволюции.

Вместе с тем элементом, который отвечает за селекцию и определение какие особи являются наиболее приспособленными к окружающей действительности, выступает естественный отбор. Отмеченный в работах Ч. Дарвина как один из ключевых процессов, который обеспечивает эволюционное развитие, естественный отбор сохраняет организмы с наиболее высоким уровнем приспособленности к окружающей среде и удаляет особи из низким уровнем приспособленности.

При рассмотрении природы, которая окружает организм, как некоторой сложно организованной системы легко заметить, что в процессе эволюции популяции живые существа под воздействием описанных факторов способны с легкостью решать некоторую оптимизационную задачу, находя в окружающих условиях оптимальные положения и состояния. В связи с этим была предложена идея генетических алгоритмов — смоделировать описанные три процесса и на их основе запустить оптимизационный поиск. Главный толчок для развития генетических алгоритмов был дан в работе Дж. Г. Холланда [1].

Каждый генетический алгоритм представляет собой итерационное применение операторов мутации, скрещивания и естественного отбора. При этом все эти операторы применяются к основной единице эволюции — популяции. В ходе такого итерационного применения этих операторов популяция должна найти некоторое оптимальное решение, при этом отнюдь не гарантируется, что это решение будет верным или же найденный оптимум будет являться глобальным.

## 2.1 Представление решения и начальная популяция

Первым этапом в реализации генетического алгоритма является выбор способа кодирования решения. Закодированные возможные решения будут представлять собой особи в популяции. Способ должен быть выбран таким образом, чтобы у операторов мутации и скрещивания была возможность с легкостью изменять каждую особь. Чаще всего при поиске оптимума некоторой вещественной функции каждая особь — это набор битов, которые кодируют вещественное число. Но данный подход не всегда может быть применен, поэтому способ кодирования решения выбирается чаще всего из постановки решаемой задачи. Одним из параметров генетического алгоритма является размер популяции  $N$ .

## 2.2 Оператор скрещивания

Данный оператор занимается выбором особей для скрещивания и самим процессом скрещивания. Задача этого оператора скомбинировать гены двух особей и создать на их основе новую особь для перехода в следующее поколение. При этом с этим процесс скрещивания происходит не всегда, а с вероятностью заданной в виде параметра  $p_c$ .

## **2.3 Оператор мутации**

Оператор просматривает каждую особь в популяции и некоторым образом ее изменяет, при этом работает так же с некоторой вероятностью заданной через параметр  $p_m$ .

## **2.4 Оператор естественного отбора**

При естественном отборе важна функция для оценки приспособленности каждой особи в популяции. Чаще всего в качестве такой функции выступает функция, оптимальное значение которой ищется. Для начала оператор вычисляет приспособленность каждого организма в популяции после чего формируется для каждой особи вероятность ее попадания в следующее поколение. Эта вероятность выше, чем наиболее оптимальное решение представляет собой рассматриваемый элемент. Выбор элементов популяции осуществляется при помощи так называемого колеса рулетки — каждой особи ставится в соответствие сектор в зависимости от уровня вероятности, после чего происходит генерация псевдослучайного числа, и в зависимости от того в какой сектор попало число, тот элемент и переходит в следующее поколение. Очевидно, что в результате окажется большинство особей с высоким уровнем приспособленности.

Среди недостатков, которыми обладает данный подход, можно выделить неуниверсальность генетических алгоритмов. Каждая задача требует уникальной разработки и адаптации всех описанных этапов под решаемую задачу. Кроме этого успешность работы алгоритма зависит от значений параметров  $p_c$ ,  $p_m$  и  $N$ .

Таким образом основными вопросами, которые стоят перед разработчиком, является реализация процессов скрещивания, мутации, естественного отбора и выбор критерия остановки алгоритма. Кроме этого необходимо исследовать параметры  $p_c$ ,  $p_m$  и  $N$ , которые существенным образом влияют на работу генетического алгоритма.

### 3 Алгоритмы для поиска центральных вершин

Так как генетические алгоритмы являются методом решения оптимизационных задач, в задаче поиска центральных вершин необходимо выделить функцию, оптимальное значение которой требуется оптимизировать. Легко заметить, что если рассмотреть граф  $G = (V, E)$  и функцию на нем  $F : V \mapsto \mathbb{R}$ , которая определяет эксцентриситет каждой вершины, то получается, что для нахождения центральных вершин необходимо найти минимальное значение этой функции с помощью генетического алгоритма.

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 [2, 3]. Этим объясняется довольно большое количество работ, в которых изучается данная задача.

#### 3.1 Тривиальный алгоритм

Легко заметить, что задача поиска центральных вершин может быть с легкостью решена при использовании алгоритма обхода в ширину. Для того, чтобы найти вершину с минимальным эксцентриситетом необходимо запустить обход в ширину из каждой вершины графа, после чего станут известны все длины путей между всеми вершинами в графе. Алгоритм поиска в ширину имеет асимптотическое время работы равное  $O(n + m)$  [4]. При этом, если запустить его из каждой вершины, то время работы всего алгоритма будет равным  $O(n^2 + nm)$ .

#### 3.2 Алгоритмы с улучшенной ассимптотикой

Кроме этого данная задача была хорошо изучена различными исследователями [5–7], которые предлагали несколько подходов, для решения подобных задач. Например, в работе [8] предлагается алгоритм, использующий эвристику разделения вершин на два множества — множество вершин с высокой степенью и множество вершин с низкой степенью. После такого разделения для множества с вершинами с высокой степенью строится доминирующее множество, после чего из этого множества запускаются обходы в ширину. Кроме этого алгоритм поиска в ширину запускается внутри множества вершин с низкой степенью. Алгоритм обхода проходит не по всем вершинам в

графе, а лишь по вершинам внутри множеств, что позволяет быстрее найти центральные вершины, а кроме этого радиус графа.

### 3.3 Алгоритмы, использующие матричное умножение

В работе [9] приводится алгоритм, который использует в своей основе матричное представление графов. Данный алгоритм оперирует в своей работе матрицами и самыми ресурсоемкими задачами в этом алгоритме являются процессы матричного перемножения, поэтому целиком и полностью асимптотика этого подхода равна времени выполнения матричного умножения. Тривиальный алгоритм имеет асимптотику  $O(n^3)$ , но при этом существует алгоритм быстрого матричного умножения [10], способный решить эту задачу за время  $O(n^{2.81})$ .

### 3.4 Существующие генетические алгоритмы

Для решения рассматриваемой задачи также были созданы различные эвристические алгоритмы. Например, в работе [11], 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 эксцентриситетов 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 реализацию. 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 эксцентризитет, 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 эксцентризитета 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.

#### 4 Описание разработанного генетического алгоритма

Генетический алгоритм должен содержать адаптированные под решаемую задачу этапы скрещивания, мутации и естественного отбора. Основная идея алгоритма может быть выражена следующим образом. Пусть существует некоторое абстрактное или реальное изображение графа, причем в центре этого изображения находятся центральные вершины графа. Тогда, если популяция генетического алгоритма представляет собой набор вершин, то этот набор может быть представлен в виде некоторого шара, внутри которого находится центральные вершины графа (см. рисунок 1).

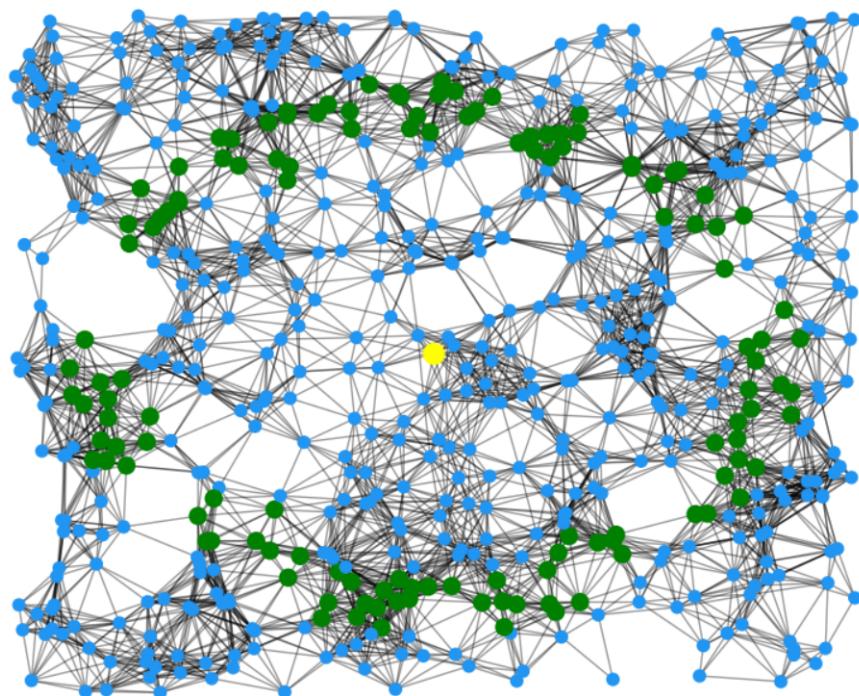


Рисунок 1 – Начальное состояние алгоритма (желтый цвет – центральная вершина, зеленый цвет – начальная популяция)

Из этой идеи следует, что для того чтобы найти центральные вершины необходимо, чтобы эта абстрактная сфера сжималась к центру. Именно этот смысл заложен в работу оператора скрещивания. В добавок к этому алгоритм должен сжимать эту «сферу» к глобальному центру, не сбиваясь в локальные оптимальные значение, за это отвечает оператор мутации. Естественный отбор соответственно занимается выбором вершин с оптимальными эксцентричитетами. Если реализовать все эти три шага и запустить их, то в идеальном варианте после нескольких итераций популяция алгоритма должна находиться в состоянии, которое показано на рисунке 2.

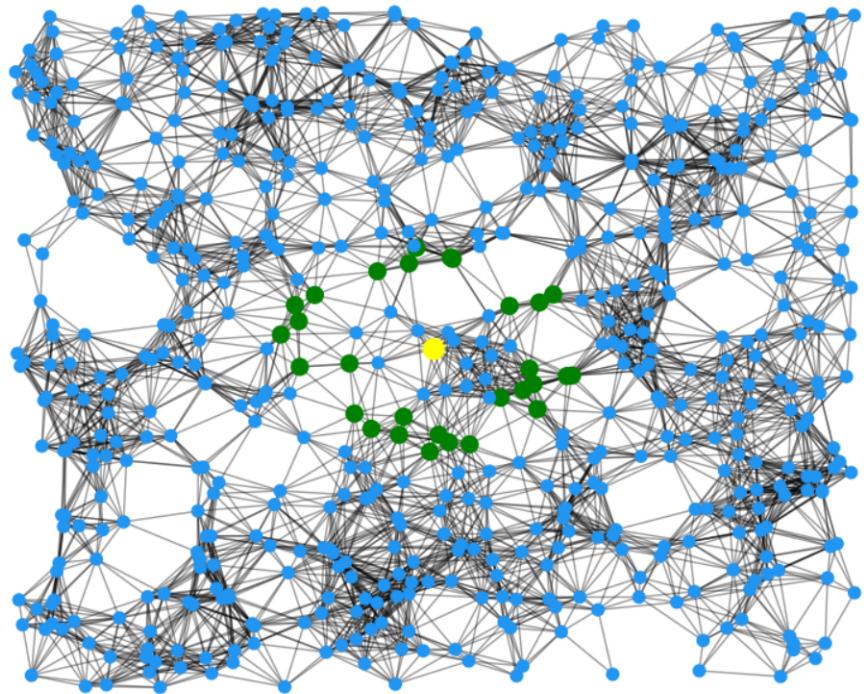


Рисунок 2 – Состояние алгоритма после нескольких итераций (желтый цвет – центральная вершина, зеленый цвет – начальная популяция)

#### 4.1 Старт алгоритма

Для начала алгоритма необходимо сгенерировать начальную популяцию с размером  $N$ , которая и будет эволюционировать. В предлагаемом алгоритме в качестве начальной популяции генерируется случайный набор уникальных вершин, причем вероятность попадания в начальную популяцию одинакова для всех вершин.

#### 4.2 Естественный отбор

Как уже говорилось ранее естественный отбор занимается выбором оптимальных вершин для продолжения работы алгоритма. Для каждой вершины находится ее эксцентриситет при помощи обхода в ширину (см. рисунок 3), после чего методом колеса рулетки, при этом приоритет отдается вершинам с меньшим эксцентриситетом.

#### 4.3 Этап скрещивания

Основной смысл разработанного алгоритма кроется в операторе скрещивания. Так как необходимо, чтобы «сфера» описываемая популяцией с каждой итерацией сжималась, то скрещивание реализовано следующим образом: в качестве родителей выбираются две вершины из популяции, после чего между

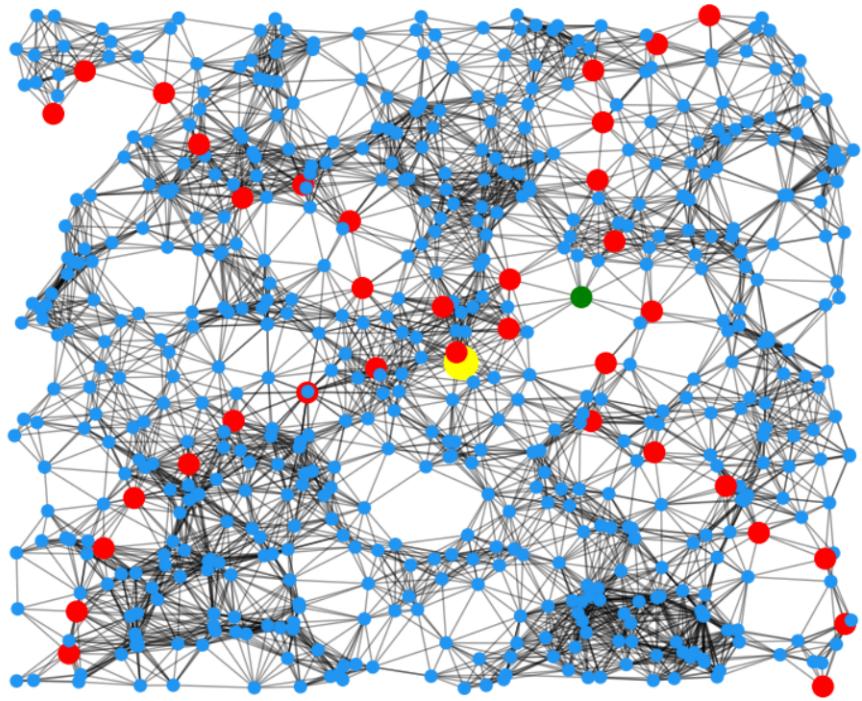


Рисунок 3 – Процесс поиска самой удаленной вершины (желтый цвет – центральная вершина, зеленый цвет – начальная популяция, красный – вершины на пути в самые удаленные точки графа)

ними находится кратчайший путь, после чего из этого пути выбирается одна вершина в качестве потомка (см. рисунок 4). Именно такая реализация данного этапа позволяет алгоритму сходиться к центральным вершинам. При этом процесс скрещивания происходит с вероятностью  $p_c$ .

#### 4.4 Этап мутации

Для того, чтобы у алгоритма была возможность выхода из локальных оптимальных значений существует этап мутации, который имеет следующую реализацию: перебираются все вершины в популяции и для каждой вершины находятся ее соседи — вершины смежные с ней, после чего с вероятностью  $p_m$  вершина заменяется на одного из своих соседей (см. рисунок 5)

Все описанные этапы представлены в виде псевдокода 1.

При работе каждого из этапов в оперативной памяти поддерживается матрица  $n \times n$ , где  $n$  — число вершин в графе, хранящая найденные расстояния между вершинами, а кроме этого та же матрица, внутри которой лежат кратчайшие пути между вершинами. Такое поддержание матриц позволяет многократно не пересчитывать расстояния между вершинами, если этого требует алгоритм. Визуальную работу алгоритма можно увидеть на рисунке 7.

```

begin
    Data: Graph  $G$ 
    Result: Vertex  $c$ , which is center of graph  $G$ 
    for  $i := 1$  to  $populationSize$  do
        |  $population[i] \leftarrow$  random vertex  $\in G$ ;
    end
    for  $i \leftarrow 1$  to  $iterationNumber$  do
        |  $Crossover()$ 
        |  $Mutation()$ 
        |  $Selection()$ 
    end
     $c.eccentricity \leftarrow \infty$ 
    for  $v \in population$  do
        | if  $v.eccentricity < c.eccentricity$  then
        | |  $c \leftarrow v$ 
        | end
    end
end

Function  $Mutation()$ 
    Data: Population on current algorithm step
    Result: Population after applying a mutation operator to each individual
    for  $i \leftarrow 1$  to  $populationSize$  do
        | if  $randomValue < mutationProbability$  then
        | |  $neighbours \leftarrow population[i].getNeighbours$ 
        | |  $population[i] \leftarrow$  random vertex from  $neighbours$ 
        | end
    end
end

Function  $Crossover()$ 
    Data: Population on current algorithm step
    Result: Population after crossover
    for  $i \leftarrow 1$  to  $populationSize$  do
        | if  $randomValue < crossPopulation$  then
        | |  $u \leftarrow$  random vertex from  $popualtion$ 
        | |  $v \leftarrow$  random vertex from  $population$ 
        | |  $path \leftarrow G.pathBetween(u, v)$ 
        | end
    end
end

Function  $Selection()$ 
    Data: Population on current algorithm step
    Result: Population for next algorithm step
    for  $i \leftarrow 1$  to  $populationSize$  do
        |  $eccentricity[i] \leftarrow population[i].eccentricity$ 
    end
    for  $i \leftarrow 1$  to  $populationSize$  do
        |  $probability[i] \leftarrow$  number from  $[0, 1]$ 
    end
    for  $i \leftarrow 1$  to  $populationNumber$  do
        |  $nextPopulation[i] \leftarrow v$  from  $population$  with using probability
    end
end

```

**Algorithm 1:** Псевдокод основных этапов предлагаемого алгоритма

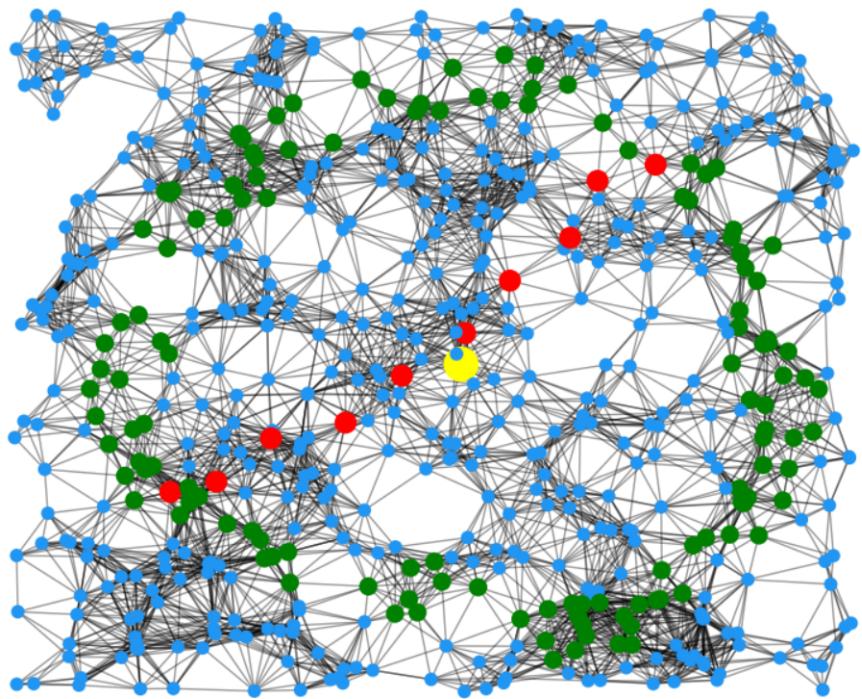


Рисунок 4 – Процесс скрещивания (желтый цвет – центральная вершина, зеленый цвет – начальная популяция, красный – кратчайший путь между вершинами популяции)

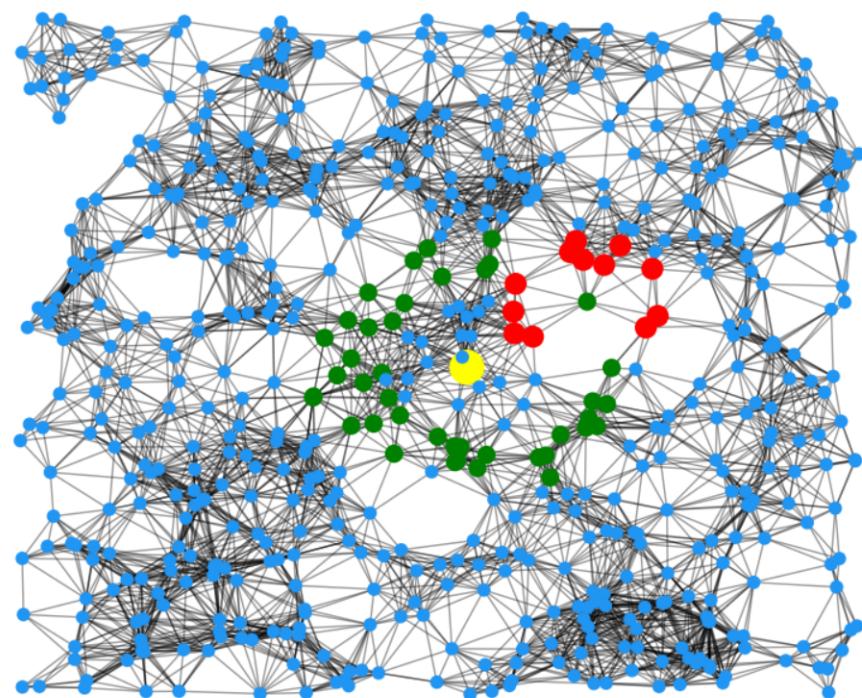
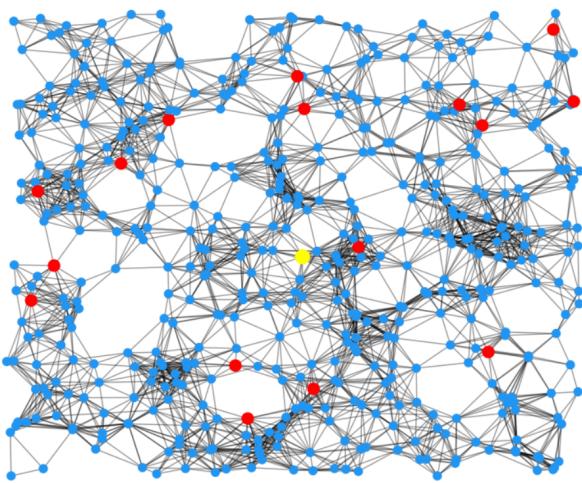
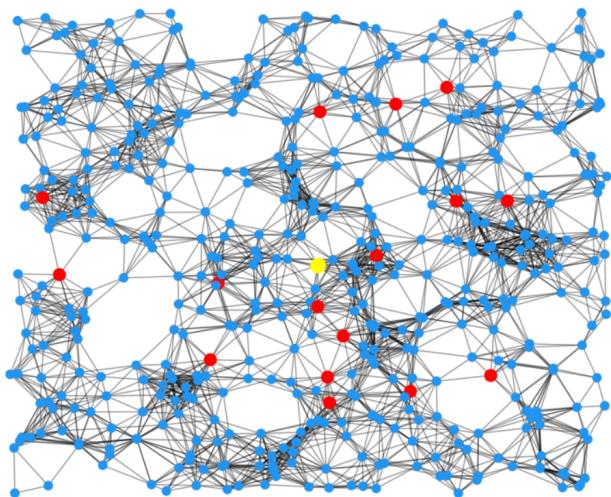


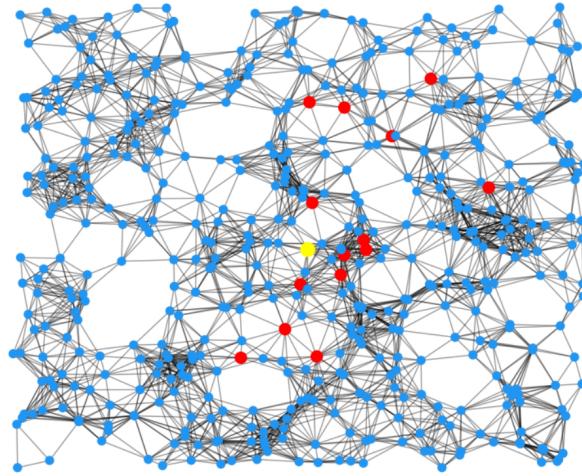
Рисунок 5 – Процесс мутации (желтый цвет – центральная вершина, зеленый цвет – начальная популяция, красный – соседние вершины одной из вершин в популяции)



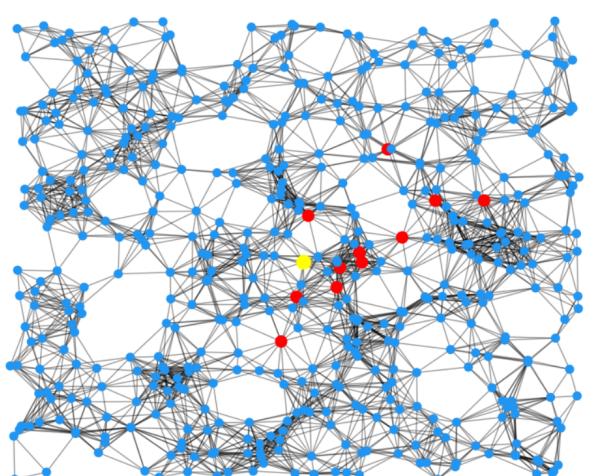
Начальная популяция



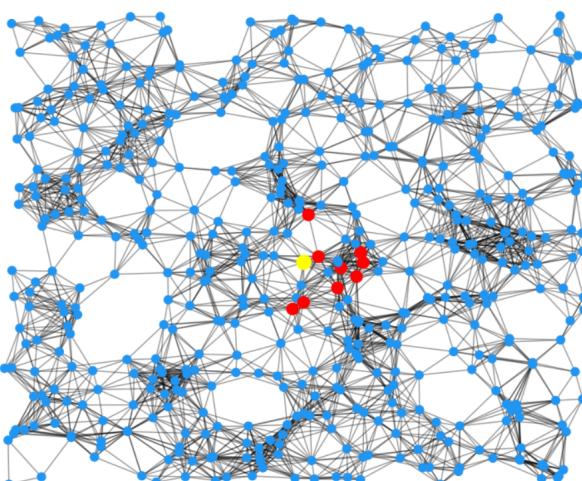
Популяция после двух шагов



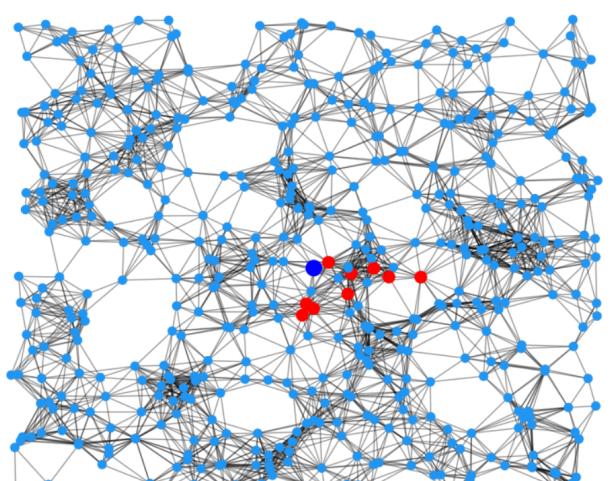
Популяция после четырех шагов



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)

## 5 Исследование алгоритма

В качестве языка программирования для исследования алгоритма был выбран язык программирования C++. Кроме этого в качестве среды разработки, в которой производилась реализация алгоритма, была выбрана Visual Studio 2017, а вычислительная машина, на которой были выполнены все испытания обладает оперативной памятью с размером 6.0 GB и процессором AMD A8-7410 с частотой 2.20 GHz.

Кроме этого для запусков тестов алгоритма необходимы наборы графов с разными свойствами. Для создания графов использовалась Python библиотека NetworkX, которая предоставляет возможности для генерации данных на основе различных моделей случайных графов.

За основную работу алгоритма отвечает класс `GeneticAlgorithm`, в котором реализованы основные методы генетического алгоритма:

- `makeSelection` — метод, отвечающий за реализацию естественного отбора,
- `crossing` — метод, отвечающий за реализацию скрещивания,
- `mutation` — метод, отвечающий за реализацию мутации,
- `getBestResult` — метод, который запускает генетический алгоритм, измеряет время его работы и возвращает найденные оптимальные значения.

Полный код класса можно увидеть в приложении А.

При всем этом для хранения найденных путей и кратчайших расстояний реализован класс `Graph`, внутри которого находятся методы, отвечающие за запуск обхода в ширину и получение найденных расстояний и кратчайших путей. Основные методы этого класса:

- `bfsFromVertex(int x)` — метод, запускающий алгоритм обхода в ширину из вершины, переданной в качестве параметра
- `getPath(int x, int y)` — метод, возвращающий расстояние между двумя вершинами в графе,
- `getEccentricity(int x)` — метод, возвращающий эксцентризитет вершины,
- `getNeighbour(int x)` — метод, получающий соседей заданной вершины.

Также с кодом этого класса можно ознакомиться в приложении Б. Сгे-

нерированные графы, на которых тестируется алгоритм, хранятся в текстовых файлах. В текстовых фалах находятся ребра графа, которые записаны на каждой строке, при этом на первой строке записано число вершин в графе и число ребер. Для того, чтобы прочитать текстовый граф была реализована следующая функция чтения:

```
1 vector<pair<int, int>> readFromFileNM(string file, int& N, int& M) {
2     ifstream in(file);
3     vector<pair<int, int>> e;
4     int n, m;
5     in >> n >> m;
6     for (int i = 0; i < m; i++) {
7         int x, y;
8         in >> x >> y;
9         e.push_back(make_pair(x, y));
10    }
11    N = n;
12    M = m;
13    return e;
14 }
```

методу на вход передаются такие параметры, как имя файла, ссылка на переменную для числа вершин в графе, а так же ссылка на число ребер в графе. Метод возвращает вектор пар, где каждая пара описывает ребро графа.

За связывание файлов с графиками с их найденными ранее радиусами отвечает класс `Repository`, внутри которого закодированы имена файлов и дополнительная информация об этих файлах. Для того, чтобы получить график существует публичный метод `getGraph`, который принимает строку, описывающую какой тип графа запрошен, после чего открывается файл, график считывается и возвращается из метода. Со всеми этими действиями можно ознакомиться в [приложении Б](#).

Для того, чтобы данные, возвращаемые из методов или передаваемые в качестве параметров были соединены в единую структуру были созданы следующие сущности. Класс `GraphDescription`:

```
1 class GraphDescription {
2 public:
3     int n, m;
4     edges e;
```

```

5     int realR;
6     GraphDescription() {}
7
8     GraphDescription(int n, int m, edges e, int realR) {
9         this->n = n;
10        this->m = m;
11        this->e = e;
12        this->realR = realR;
13    }
14 };

```

используется для хранения графа в нем содержаться поля, отвечающие за размеры графа и его радиус.

Кроме этого для того, чтобы хранить результаты вычислительных экспериментов существует класс GaTestResult:

```

1 class GaTestResult {
2 public:
3     GaTestResult() {}
4
5     GaTestResult(double popSize, double pm,
6                  double pc, double functionValue) {
7         this->pm = pm;
8         this->pc = pc;
9         this->popSize = popSize;
10        this->functionValue = functionValue;
11    }
12
13    double pc, pm;
14    int popSize;
15    double functionValue;
16 };

```

В объектах данного класса хранятся значения вычисленной функции, например времени, а кроме этого значения параметров, на которых были получены эти результаты.

Для того, чтобы облегчить работу с запусками алгоритма с разными параметрами и легко проводить различного рода тесты создан класс Experiment. Данный класс включает реализацию всех необходимых методов для исследования параметров алгоритма:

- `calculateTimeErrorValue` — private метод, который запускает генетический алгоритм, и возвращает среднее время работы и процент ошибок,
- `testWithChangebleProb` — метод который принимает размер популяции и фиксированный параметр, а так же флаг показывающий какой из параметров будет изменяться для исследования,
- `oneDimentionFixedGATest` — запуск теста с изменяющимся параметром,
- `simpleTimeErrorTest` — простой запуск алгоритма, с переданными ему вероятностью скрещивания, мутации и размеров популяции,
- `simpleNANTest` — запуск алгоритма N4N,
- `rmpcGaTest` — метод для запуска теста с перебором всех значений  $p_c$  и  $p_m$ ,
- `nGATest` — метод для запуска теста с перебором параметра  $N$ .

Полный код класса представлен в приложении **B**.

## 6 Модели случайных графов

Для исследования алгоритма применялись следующие модели случайных графов: модель Эрдеша-Ренъи, модель Барабаши-Альберт и модель случайного геометрического графа. Все они в той или иной степени частично описывают реальные графы, такие как компьютерные или социальные сети.

### 6.1 Модель случайного графа Эрдеша-Ренъи

Данная модель представленная в работе [12] 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.

### 6.2 Модель случайного графа Барабаши-Альберт

Данная модель случайного графа была предложена в работе [13]. 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 случайнym 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.

### 6.3 Геометрический случайный граф

Данная модель случайного графа [14] 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.

## 7 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  $r = 0.1$ .

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 превосходит 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

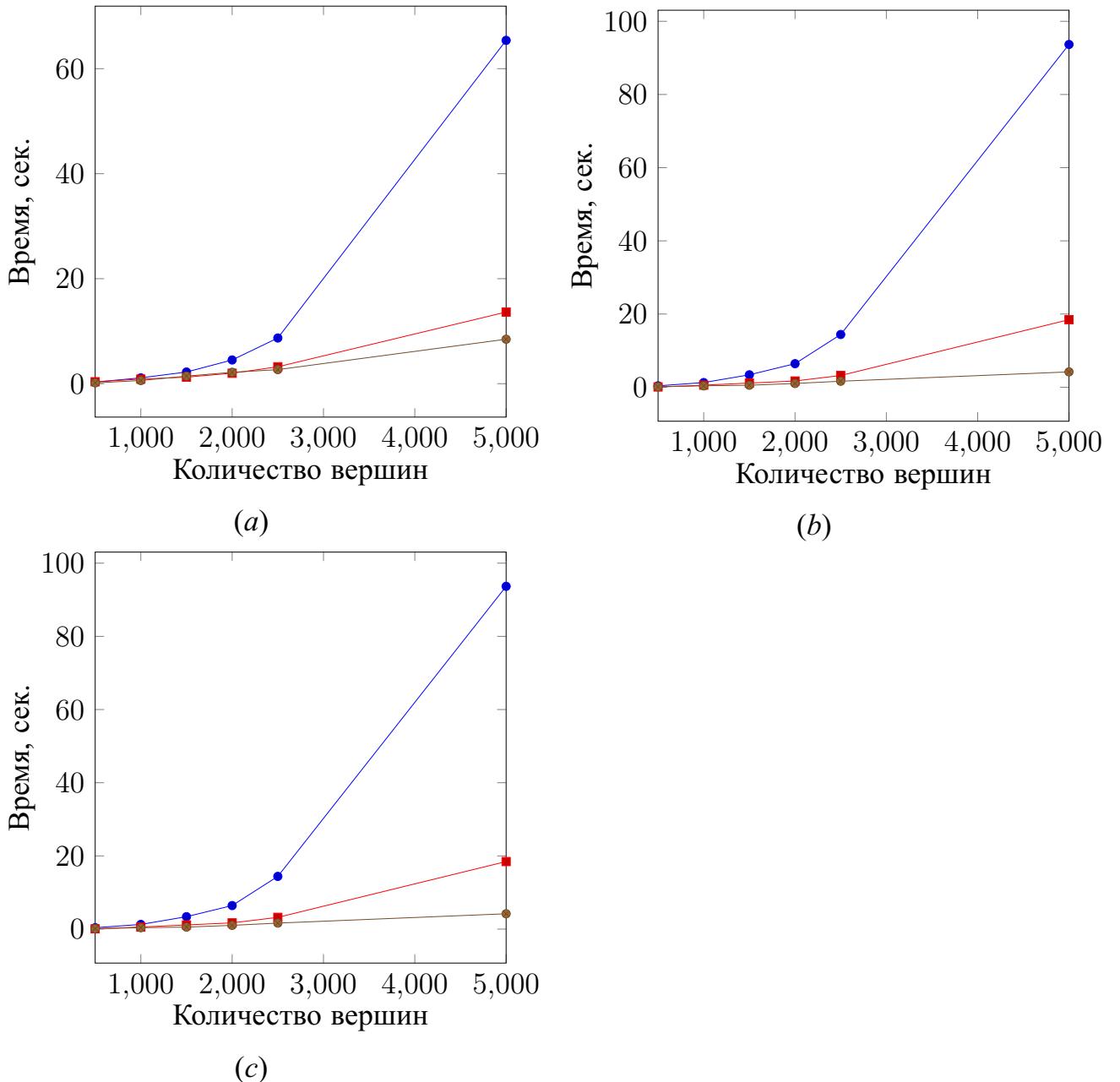


Рисунок 7 – Графики зависимости времени работы от размеров графа (синий цвет – тривиальный  $O(nm + n^2)$  алгоритм, красный – алгоритм с улучшенной асимптотикой  $O(m\sqrt{n})$ , коричневый – генетический алгоритм): (a) – модель Эрдеша-Ренъи  $p = 1\%$ , (b) – модель Барабаши-Альберта  $m = 2$ , (c)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

Таблица 1 – Время работы алгоритма и процент неверных ответов на модели случайного графа Эрдоша-Ренъи. GA – генетический алгоритм, N4NA – N4N алгоритм, AA – алгоритм с улучшенной ассимптотикой

Размеры графа		Время, сек.			Ошибка, %	
$ V $	$ E $	GA	N4NA	AA	GA	N4NA
500	3572	0.11	0.39	0.07	38.0	40.0
1000	14202	0.31	0.77	0.56	21.0	50.0
1500	31861	0.71	1.44	1.13	13.0	62.0
2000	57438	1.20	1.92	1.70	8.0	48.0
2500	90268	1.76	2.68	3.20	4.0	30.0
5000	358553	4.48	8.61	18.45	0.0	0.0
10000	1439255	13.54	26.0	41.47	0.0	0.0

Таблица 2 – Время работы алгоритма и процент неверных ответов на модели случайного графа Барабаши-Альберт. GA – генетический алгоритм, N4NA – N4N алгоритм, AA – алгоритм с улучшенной ассимптотикой

Размеры графа		Время, сек.			Ошибка, %	
$ V $	$ E $	GA	N4NA	AA	GA	N4NA
500	996	0.07	0.29	0.08	16.0	0.0
1000	1996	0.18	0.68	0.36	12.0	0.0
1500	2996	0.37	1.24	0.98	4.0	0.0
2000	3996	0.55	1.67	1.68	1.0	0.0
2500	4996	0.69	2.18	3.22	0.0	0.0
5000	9996	1.84	8.28	14.17	0.0	0.0
10000	19996	3.90	15.8	24.56	0.0	0.0

Таблица 3 – Время работы алгоритма и процент неверных ответов на модели случайного геометрического графа. GA – генетический алгоритм, N4NA – N4N алгоритм, AA – алгоритм с улучшенной ассимптотикой

Размеры графа		Время, сек.			Ошибка, %	
$ V $	$ E $	GA	N4NA	AA	GA	N4NA
500	3572	0.11	0.39	0.07	38.0	40.0
1000	14202	0.31	0.77	0.56	21.0	50.0
1500	31861	0.71	1.44	1.13	13.0	62.0
2000	57438	1.20	1.92	1.70	8.0	48.0
2500	90268	1.76	2.68	3.20	4.0	30.0
5000	358553	4.48	8.61	18.45	0.0	0.0
10000	1439255	13.54	26.0	41.47	0.0	0.0

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.

## 8 Исследование параметров генетического алгоритма

В приведенных ранее результатах в качестве значений параметров генетического алгоритма были выбраны значения  $N = 50$ ,  $p_c = 0.7$  и  $p_m = 0.1$ . Однако эти значения выбирались в качестве тестовых для того, чтобы проверить жизнеспособность идей, заложенных в алгоритм. При этом эти параметры ключевым образом влияют как на точность алгоритма, так и на время его выполнения. В связи с этим был также проведен еще ряд экспериментов, в которых исследовались эти параметры. Для начала были произведены запуски, в которых перебирались значения для  $p_m$  и  $p_c$  от 0 до 1. Для каждого значения этих параметров измерялось время работы алгоритма и его процент ошибок. При этом значение размера популяции было равно 20. Результаты этих экспериментов представлены в таблицах 4 и 5.

Таблица 4 – Время работы (сек.) алгоритма на представителе модели геометрического случайного графа  $|V| = 2500$ ,  $|E| = 90268$ ,  $N = 20$

pm/pc	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
0.0	0.06	0.11	0.13	0.15	0.17	0.19	0.20	0.29	0.28	0.27	0.31
0.1	0.18	0.26	0.27	0.33	0.32	0.36	0.36	0.41	0.40	0.40	0.40
0.2	0.29	0.35	0.52	0.55	0.52	0.54	0.49	0.54	0.56	0.54	0.50
0.3	0.45	0.50	0.53	0.57	0.60	0.54	0.60	0.58	0.65	0.68	0.66
0.4	0.51	0.53	0.64	0.71	0.69	0.62	0.64	0.66	0.70	0.75	0.78
0.5	0.62	0.67	0.74	0.79	0.78	0.84	0.87	0.82	0.80	0.82	0.93
0.6	0.60	0.77	0.77	0.96	0.87	0.86	0.88	0.93	1.01	1.04	1.14
0.7	0.86	0.96	1.04	1.09	1.13	1.09	1.00	1.15	1.03	1.32	1.00
0.8	0.76	0.85	0.89	0.93	0.98	0.96	1.02	1.19	1.05	1.08	1.13
0.9	0.82	0.91	1.01	1.01	1.23	1.29	1.30	1.31	1.18	1.20	1.10
1.0	0.96	1.08	1.03	1.22	1.25	1.30	1.30	1.26	1.18	1.32	1.50

Таблица 5 – Процент ошибок (%) алгоритма на представителе модели геометрического случайного графа  $|V| = 2500$   $|E| = 90268$   $N = 20$

pm/pc	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
0.0	81	54	32	52	42	52	59	53	62	60	61
0.1	48	35	19	28	33	34	28	32	23	39	31
0.2	46	27	29	20	21	25	21	25	26	21	17
0.3	44	23	18	17	20	18	15	14	15	17	21
0.4	32	28	19	22	9	16	21	16	18	30	16
0.5	32	16	14	19	20	16	22	18	14	14	9
0.6	30	18	12	21	11	12	15	9	12	20	18
0.7	32	17	20	14	13	10	19	12	22	8	12
0.8	33	22	20	14	21	15	21	16	15	14	19
0.9	24	18	24	15	23	17	19	14	20	11	14
1.0	32	16	13	14	12	15	16	14	7	13	9

Из этих таблиц видно, что время работы алгоритма растет по мере увеличения как параметра  $p_m$ , так и параметра  $p_c$ . Вместе с тем видно, что и процент неверных ответов падает по мере увеличения тех же параметров. Очевидно, что необходимо найти некоторые оптимальные значения для того, чтобы процент неверных ответов был невелик и одновременно с этим необходимо, чтобы время работы было минимальным. Для того чтобы соединить воедино два этих фактора была введена функция:

$$F(pm, pc) = \alpha \text{time}(pm, pc) + \beta \text{error}(pm, pc), \quad (1)$$

которая учитывает время работы и процент ошибок, причем параметр  $\alpha$  отвечает за уровень значимости временных затрат, а параметр  $\beta$  отвечает за значимость процента ошибок. Если подставить все полученные данные в эту функцию, то получится результат, который представлен в таблице 6.

Таблица 6 – Значения функции  $F$ ,  $\alpha = 0.3$ ,  $\beta = 0.7$ ,  $|V| = 2500$   $|E| = 90268$   $N = 20$

pm/pc	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
0.0	0.71	0.49	0.30	0.48	0.40	0.49	0.55	0.52	0.59	0.57	0.59
0.1	0.45	0.36	0.22	0.31	0.35	0.37	0.32	0.36	0.28	0.42	0.35
0.2	0.46	0.31	0.36	0.28	0.29	0.32	0.28	0.32	0.34	0.29	0.25
0.3	0.47	0.30	0.26	0.26	0.29	0.27	0.25	0.24	0.26	0.28	0.32
0.4	0.38	0.35	0.29	0.33	0.22	0.26	0.31	0.27	0.30	0.41	0.30
0.5	0.40	0.27	0.27	0.32	0.33	0.31	0.37	0.32	0.28	0.29	0.26
0.6	0.38	0.31	0.26	0.37	0.27	0.28	0.31	0.26	0.31	0.38	0.38
0.7	0.45	0.34	0.38	0.34	0.34	0.30	0.36	0.33	0.40	0.33	0.30
0.8	0.44	0.36	0.35	0.31	0.38	0.32	0.39	0.38	0.34	0.34	0.39
0.9	0.37	0.34	0.41	0.33	0.44	0.40	0.42	0.38	0.41	0.34	0.34
1.0	0.47	0.35	0.32	0.36	0.35	0.39	0.40	0.37	0.30	0.38	0.38

Из этой таблицы видно, что оптимальных значений функция  $F$  достигает в центре таблицы. Например, в эксперименте, который описывает таблица 6 оптимальное значение достигается при  $p_m = 0.4$  и  $p_c = 0.4$ . При фиксированных значениях параметра  $p_m = 0.4$  и  $p_m = 0.6$  были построены графики 8 и 9, на которых отражено нормированное время работы и нормированный процент ошибки. Из этих графиков видно, что при увеличении параметра  $p_m$  значительно увеличивается и время работы алгоритма и уменьшается процент неверных ответов.

Кроме этого важным параметром для генетического алгоритма является размер популяции  $N$  и в связи с этим был проведен эксперимент, в котором измерялся процент ошибок и время работы при фиксированных значениях

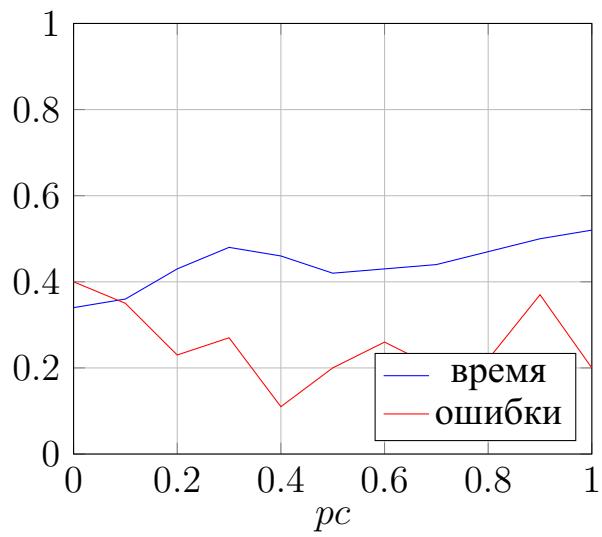


Рисунок 8 – Нормализованные значения времени выполнения алгоритма и процента ошибок с параметрами  $pm = 0.4$ ,  $|V| = 2500$ ,  $|E| = 90268$ ,  $N = 20$

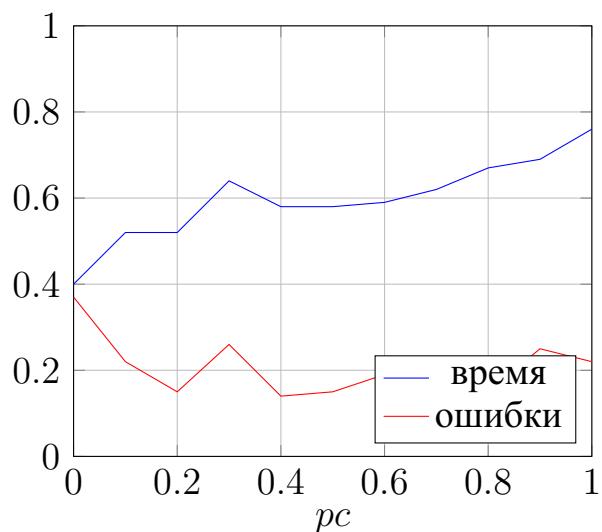


Рисунок 9 – Нормализованные значения времени выполнения алгоритма и процента ошибок с параметрами  $pm = 0.6$ ,  $|V| = 2500$ ,  $|E| = 90268$ ,  $N = 20$

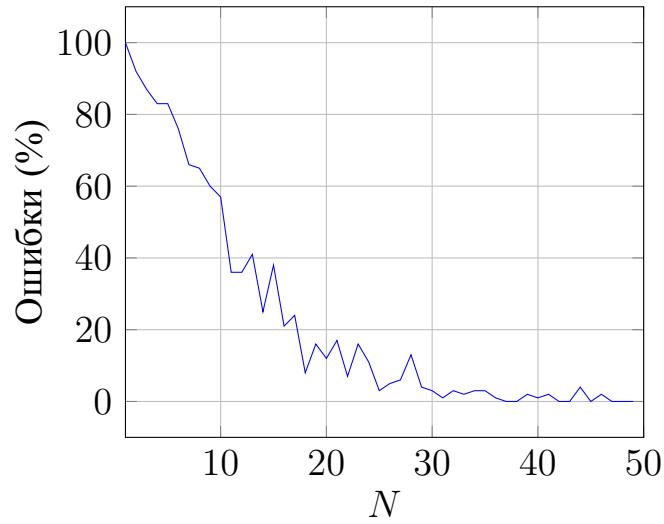


Рисунок 10 – Зависимость процента ошибок (%) от размеров популяции  $N$ , с параметрами  $p_m = 0.4, p_c = 0.4$

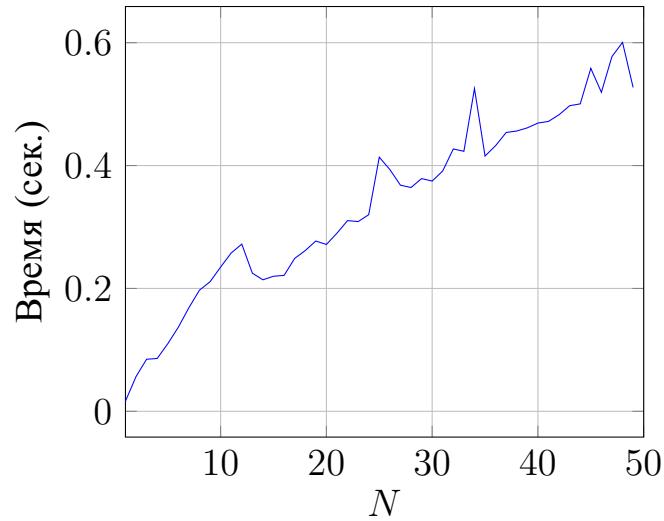


Рисунок 11 – Зависимость времени работы (сек.) от размеров популяции  $N$ , с параметрами  $p_m = 0.4, p_c = 0.4$

$p_m = 0.4$  и  $p_c = 0.4$  [10](#) и [11](#).

На этих графиках видно, что время работы алгоритма имеет практически линейную зависимость, а также процент ошибок экспоненциально уменьшается в зависимости от числа  $N$  и после значения 30, становится близким к нулю.

Очевидно, что в зависимости от того каким образом будут изменяться размеры графов, будет зависеть и набор оптимальных значений параметров. В связи с этим были проведены вычислительные эксперименты, на других графах другой размерности. Результаты этих экспериментов представлены в таблицах [7](#), [8](#), [9](#).

Таблица 7 – Время работы алгоритма (сек.) на представителе модели геометрического случайного графа  $|V| = 5000$ ,  $|E| = 358553$ ,  $N = 20$

pm/pc	0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
0.00	0.07	0.13	0.17	0.21	0.33	0.30	0.39	0.38	0.39	0.32	0.36
0.10	0.22	0.31	0.36	0.39	0.42	0.44	0.47	0.53	0.59	0.55	0.57
0.20	0.44	0.54	0.60	0.64	0.76	0.73	0.64	0.75	0.67	0.69	0.68
0.30	0.52	0.77	0.80	0.83	0.84	0.88	0.80	0.83	0.85	0.85	0.86
0.40	0.61	0.74	0.83	1.17	1.02	1.12	1.07	1.01	1.13	1.20	1.34
0.50	0.88	1.07	0.96	0.98	1.03	1.06	1.08	1.10	1.11	1.13	1.29
0.60	0.82	0.96	1.03	1.16	1.28	1.25	1.30	1.31	1.29	1.39	1.46
0.70	1.17	1.14	1.25	1.46	1.36	1.44	1.58	1.40	1.46	1.44	1.56
0.80	1.05	1.18	1.27	1.36	1.41	1.43	1.49	1.51	1.56	1.54	1.57
0.90	1.17	1.31	1.42	1.66	1.55	1.58	1.61	1.66	1.68	1.74	1.63
1.00	1.19	1.41	1.52	1.73	1.69	1.69	1.75	1.72	1.79	1.86	1.89

Таблица 8 – Процент ошибок (%) на представителе модели геометрического случайного графа  $|V| = 5000$ ,  $|E| = 358553$ ,  $N = 20$

pm/pc	0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
0.00	81	53	41	45	47	40	53	49	53	54	51
0.10	70	39	31	28	33	34	21	25	26	25	24
0.20	45	22	30	14	27	21	20	18	27	22	21
0.30	52	27	23	19	17	15	19	21	21	23	22
0.40	47	25	18	25	18	14	23	19	19	15	16
0.50	36	22	19	23	18	20	16	15	18	17	16
0.60	41	16	27	16	18	12	18	15	20	9	10
0.70	45	23	21	18	17	20	16	23	12	18	16
0.80	39	22	20	22	17	16	16	15	13	19	17
0.90	35	25	17	20	19	20	19	22	11	20	17
1.00	33	19	20	18	17	16	18	21	10	11	17

Таблица 9 – Значения функции  $F$ ,  $\alpha = 0.3$ ,  $\beta = 0.7$ ,  $|V| = 5000$ ,  $|E| = 358553$ ,  $N = 20$

pm/pc	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
0.00	0.71	0.48	0.38	0.42	0.46	0.39	0.52	0.48	0.52	0.52	0.50
0.10	0.64	0.39	0.33	0.30	0.35	0.36	0.26	0.30	0.32	0.30	0.30
0.20	0.46	0.28	0.35	0.22	0.35	0.30	0.27	0.27	0.34	0.30	0.29
0.30	0.53	0.36	0.33	0.30	0.28	0.27	0.29	0.31	0.32	0.33	0.33
0.40	0.50	0.33	0.29	0.40	0.32	0.30	0.37	0.32	0.34	0.32	0.35
0.50	0.45	0.36	0.32	0.35	0.32	0.34	0.31	0.30	0.33	0.33	0.34
0.60	0.48	0.29	0.40	0.32	0.36	0.30	0.36	0.34	0.38	0.30	0.32
0.70	0.57	0.38	0.38	0.39	0.36	0.40	0.39	0.42	0.34	0.38	0.39
0.80	0.50	0.38	0.37	0.41	0.37	0.37	0.37	0.37	0.36	0.41	0.40
0.90	0.49	0.42	0.37	0.44	0.41	0.42	0.42	0.45	0.36	0.45	0.41
1.00	0.47	0.39	0.41	0.43	0.42	0.41	0.43	0.45	0.37	0.39	0.45

Из этих таблиц видно, что тенденция роста времени работы сверху вниз и слева направо сохранена, что можно сказать и про такую же тенденцию к снижению ошибок. Вместе с тем из таблицы 9 видно, что оптимальные

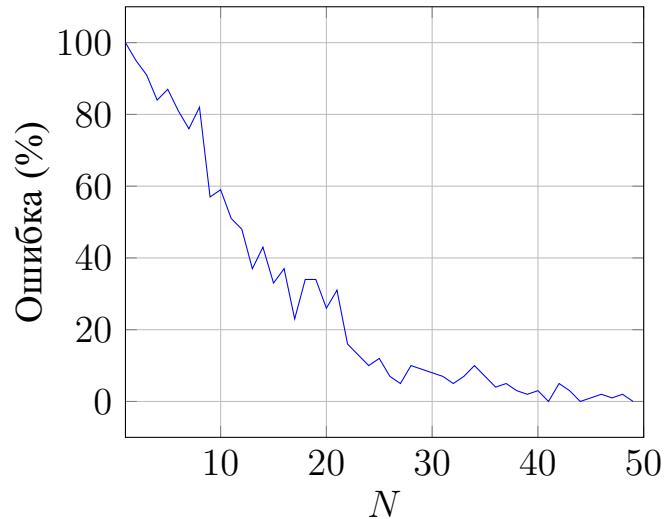


Рисунок 12 – Зависимость процента ошибок (%) от размеров популяции  $N$ , с параметрами  $p_m = 0.2, p_c = 0.3, |V| = 5000, |E| = 358553$

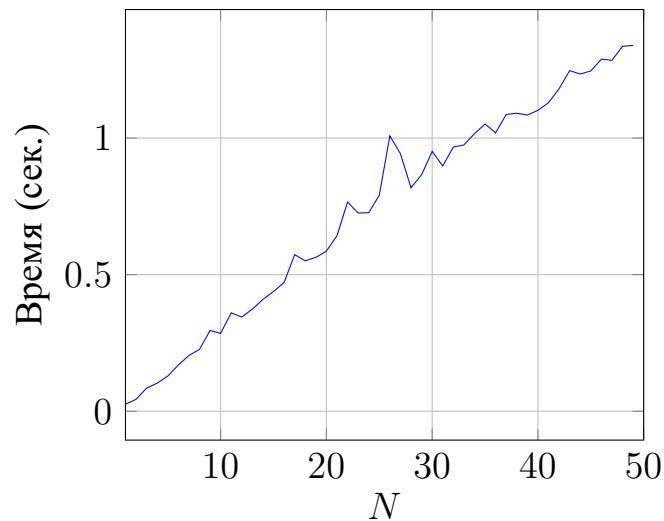


Рисунок 13 – Зависимость времени работы (сек.) от размеров популяции  $N$ , с параметрами  $p_m = 0.2, p_c = 0.3, |V| = 5000, |E| = 358553$

значения параметров равны  $p_m = 0.2$  и  $p_c = 0.3$ . На этом же графике был произведен замер времени работы и процента ошибок в зависимости от размеров популяции. Результаты этих измерений представлены на графиках

Из этих графиков точно так же видно, что время работы растет линейно, а процент неверных ответов падает экспоненциально. Кроме этого для некоторых значений  $N$  было найдено минимальное значение функции  $F$  при различных значениях  $\alpha$  и  $\beta$  (см. рисунки 14 и 15).

Как видно из графиков, при больших значениях параметра  $\alpha$ , отражающего значимость временных затрат значения функции  $F$  возрастают с увеличением размеров популяции, а при равных значениях  $\alpha$  и  $\beta$  такого большого

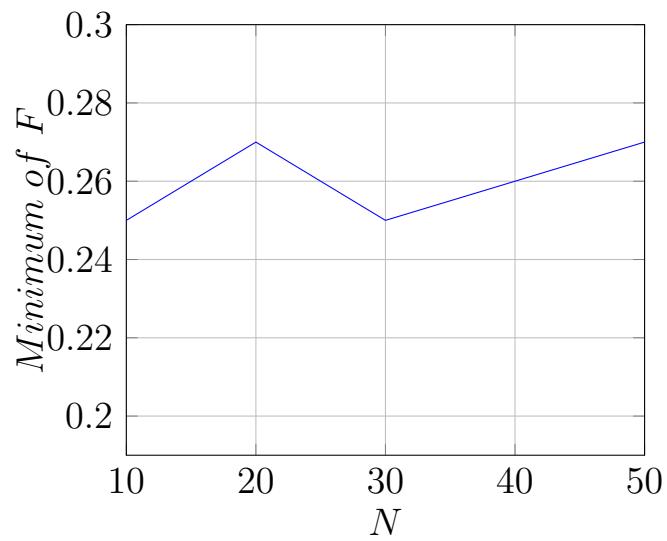


Рисунок 14 – Зависимость минимума функции  $F$  с параметрами ( $\alpha = 0.5 \beta = 0.5$ ) с  $N$

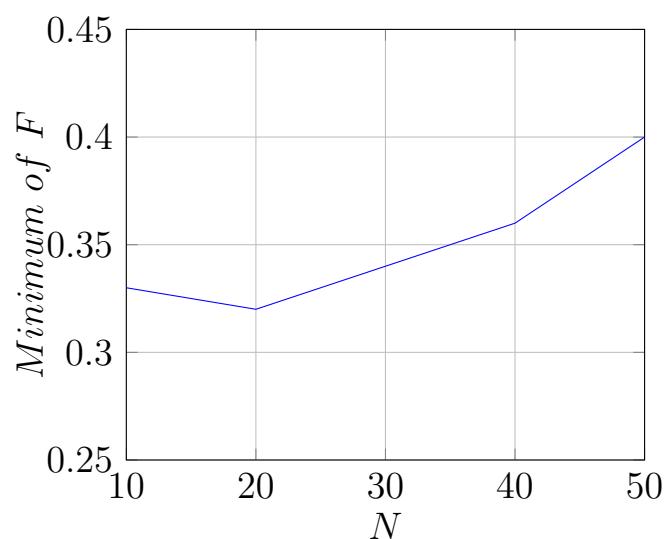


Рисунок 15 – Зависимость минимума функции  $F$  с параметрами  $\alpha = 0.7 \beta = 0.3$  от размера популяции  $N$

изменения функции не происходит.

## 9 Описание приложения

Приложение представляет собой веб-сайт, через который предоставляется возможность для работы с генетическим алгоритмом. С одной стороны у пользователя есть доступ к исследованию алгоритма и его запуску с различными параметрами, а с другой пользователь может заняться исследованием собственного графа и запустить алгоритм на нем.

При входе в приложение пользователь попадает на главную страницу см. рисунок 16.



Рисунок 16 – Главная страница

На главной странице пользователь видит небольшое описание алгоритма и, главным образом, навигационную панель, которая находится наверху страницы. На этой навигационной панели находятся ссылки на страницу с формой для поиска центральных вершин, страницу с формой для запуска алгоритма с различными параметрами и на различных графах, а так же ссылка для входа зарегистрированных пользователей.

После перехода на страницу входа (см. рисунок 17) пользователь может ввести свои логин и пароль и, после успешной аутентификации он будет перенаправлен на главную страницу, при этом в навигационной панели появятся ссылки на страницу добавления новых пользователей, новых графов, а так же будет отображаться его логин (см. рисунок 18)

При переходе на страницу, с которой возможно загрузить граф для поиска радиуса, пользователю показывается форма, через которую загружается граф (см. рисунок 19).

Похожая страница показывается пользователю при загрузке нового графа на сервер (см. рисунок 20).



Рисунок 17 – Страница входа

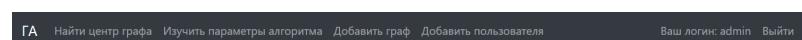


Рисунок 18 – Навигационная панель после входа в систему

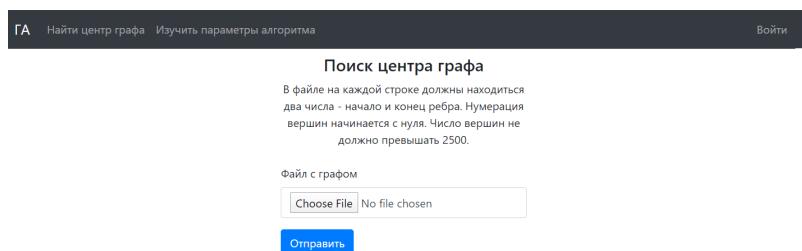


Рисунок 19 – Форма для загрузки графа с целью поиска радиуса

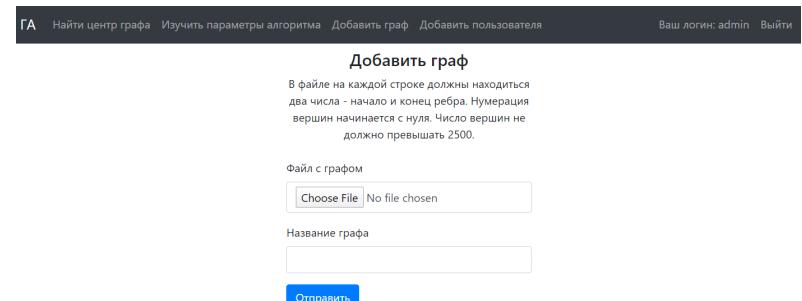


Рисунок 20 – Страница для добавления графа

Если пользователь не загрузит граф или загрузит файл в неверном формате, то получит страницу с сообщением об ошибке (см. рисунок 21, 22).



Рисунок 21 – Страница с сообщением об ошибке

При верно загруженном файле пользователю открывается страница, на которой отображаются результаты запуска алгоритма: время его работы, полученный радиус графа и возможные центральные вершины (см. рисунок



Рисунок 22 – Страница с сообщением об ошибке

23). Кроме этого у пользователя есть возможность запускать генетический



Рисунок 23 – Страница с результатами поиска центральных вершин

алгоритм с различными параметрами (см. рисунок 24). На форме находятся

Тестирование параметров генетического алгоритма

Выберите граф  
Граф Барбаша-Альберт N = 500 M = 996

Вероятность мутации  
0.5

Вероятность скрещивания  
0.5

Размер популяции  
30

Протестировать

Рисунок 24 – Форма для исследования параметров алгоритма

несколько ползунков, через которые можно выставить основные параметры генетического алгоритма, а также там находится выпадающий список, в котором выбирается один из сохраненных графов. После того как пользователь выбрал соответствующие параметры и отправил форму на сервер с выбранными параметрами, запускается алгоритм, причем несколько раз, для того, чтобы получить эмпирическую оценку времени работы и процента неверных ответов. Именно эти результаты показываются на форме, которая открывается пользователю после работы алгоритма (см. рисунок 25). Каждому зарегистри-

Рисунок 25 – Форма с результатами исследования параметров алгоритма

рованному пользователю предоставляется возможность для добавления новых

пользователей через соответствующую форму (см. рисунок 26). К данным, которые вводит пользователь, выдвигаются следующие требования: длина пароля и логина должна быть от 5 до 50 символов, и пользователь с таким же логином не должен уже существовать в базе данных. Если эти условия не будут выполнены, пользователь получит соответствующее сообщение об ошибке (см. рисунок 27).

ГА Найти центр графа Изучить параметры алгоритма Добавить граф Добавить пользователя  
Ваш логин: admin Выйти

Создать аккаунт

Логин

Пароль

Повторите пароль

**Войти**

Рисунок 26 – Форма для регистрации нового пользователя

ГА Найти центр графа Изучить параметры алгоритма Добавить граф Добавить пользователя  
Ваш логин: admin Выйти

Создать аккаунт

Логин

**Пользователь с таким логином уже существует**

Пароль

Повторите пароль

**Войти**

Рисунок 27 – Форма с ошибкой при неверных данных для регистрации пользователя

## **ЗАКЛЮЧЕНИЕ**

В рамках работы поставленная цель была достигнута. Для поиска центральных вершин был предложен генетический алгоритм, а кроме этого созданы программные приложения позволившие его исследовать. Исходя из полученных результатов, можно сказать, что во многом время и качество работы алгоритма зависит от параметров  $p_m$ ,  $p_c$  и  $N$ , и при правильно подобранных значениях алгоритм не уступает по своим характеристикам разработанным ранее алгоритмам.

## СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

- 1 *Holland, J.* Adaption in Natural and Artificial Systems Adaption in Natural and Artificial Systems / J. Holland. — University of Michigan Press, 1975.
- 2 Network Analysis / Ed. by U. Brandes, T. Erlebach. — Springer Berlin Heidelberg, 2005.
- 3 *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.
- 4 *Кормен, Т.* Алгоритмы: построение и анализ / Т. Кормен. — Вильямс, 2019.
- 5 *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.
- 6 *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.
- 7 *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.
- 8 *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.
- 9 *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.
- 10 *Strassen, V.* Gaussian elimination is not optimal / V. Strassen // *Numerische Mathematik*. — Aug 1969. — Vol. 13, no. 4. — Pp. 354–356.
- 11 *Alkhalifah, 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.
- 12 *Erdös, P.* On random graphs I / P. Erdös, A. Rényi // *Publicationes Mathematicae Debrecen*. — 1959. — Vol. 6. — P. 290.

- 13 *Albert, R.* Statistical mechanics of complex networks / R. Albert, A.-L. Barabasi // *Reviews of Modern Physics*. — 2002. — Vol. 74, no. 1. — Pp. 47–97.
- 14 *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

### Код генетического алгоритма

Далее приводится код класса GeneticAlgorithm:

```
1 #pragma once
2
3 #include <vector>
4 #include <algorithm>
5 #include <set>
6 #include <ctime>
7
8 #include "GraphWork.h"
9 #include "Random.h"
10 #include "Strassen.h"
11 #include "VectorOutput.h"
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 // Класс для работы с генетическим алгоритмом
22 class GeneticAlgorithm {
23 private :
24     // Ссылка на граф
25     Graph* G;
26     // Размер популяции
27     int populationSize ;
28     // Параметры, отвечающие за мутацию и скрещивание
29     double mutationP, crossP;
30     // Вектор с набором вершин в популяции
31     vector<int> population ;
32     // Модуль декомпьютер для работы с генерацией псевдо случайных величин
33     Random rd;
34     // Метод отвечающий за старт алгоритма,
35     // принимает ссылку на переменную для записи времени работы алгоритма.
36     void startAlgorithm (double& time) {
37         int stepN = 20;
38         double start = clock();
```

```

39         for (int i = 0; i < stepN; i++) {
40             this ->evolutionStep();
41         }
42         double finish = clock();
43         string message = "Time of genetic algorithm: ";
44         time = ( finish - start ) / CLOCKS_PER_SEC;
45         #ifdef PRINT_TIME
46             cout << "" << ( finish - start ) / CLOCKS_PER_SEC << endl;
47         #endif
48     }
49
50     public :
51         // Конструктор класса, принимает ссылку на граф, размер популяции,
52         // вероятность скрещивания и вероятность мутации
53     GeneticAlgorithm(Graph* g, int popSize, double pC, double pM) {
54         this ->G = g;
55         this ->populationSize = popSize;
56         this ->crossP = pC;
57         this ->mutationP = pM;
58         // Генерация начальной популяции
59         for (int i = 0; i < popSize; i++) {
60             population .push_back(rand() % g->Size());
61         }
62     }
63
64     // Этап селекции
65     void makeSelection() {
66         // Поиск эксцентриситетов всех вершин в популяции
67         vector<int> e;
68         for (int i = 0; i < population .size () ; i++) {
69             e.push_back(G->getEccentricity(population [i]));
70         }
71         // вычисление вероятности для попадания в следующее поколение
72         // вычисление ее таким образом, чтобы вероятность была выше
73         // у вершин с меньшим эксцентриситетом
74         vector<double> probability ;
75         double maxV = *max_element(e.begin(), e.end()), leftSum = 0;
76         for (int i = 0; i < e.size () ; i++) {
77             leftSum += maxV / double(e[i]);
78         }
79         double x = 1.0 / leftSum;

```

```

80         for (int i = 0; i < e.size(); i++) {
81             probability.push_back(maxV / double(e[i]) * x);
82         }
83         // этап селекции на основе вычисленных вероятностей
84         vector<int> nextPopulation;
85         for (int i = 0; i < this->populationSize; i++) {
86             nextPopulation.push_back(rd.choice(population, probability));
87         }
88         this->population = nextPopulation;
89     }
90     // этап скрещивания
91     void crossing () {
92         // вектор для результата скрещивания
93         vector<int> crossed;
94         for (int i = 0; i < populationSize; i++) {
95             // скрещивание происходит с заданной вероятностью
96             if (rd.getRandomLowerOne() < crossP) {
97                 // выбор индексов вершин из популяции
98                 int ind1 = rd.randint(0, populationSize - 1);
99                 int ind2 = rd.randint(0, populationSize - 1);
100                // получение самих вершин
101                int u = population[ind1], v = population[ind2];
102                // поиск кратчайших путей между вершинами
103                vector<int> path = G->getPath(u, v);
104                // выбор вершины из середины пути
105                crossed.push_back(path[path.size() / 2]);
106            }
107        }
108        // добавление потомков в популяцию
109        this->population.insert (population.end(), crossed.begin(), crossed.end())
110        ;
111    }
112    // процесс мутации
113    void mutation () {
114        for (unsigned int i = 0; i < this->population.size(); i++) {
115            // с учетом вероятности мутации
116            if (rd.getRandomLowerOne() < mutationP) {
117                // поиск соседей
118                vector<int> n = G->getNeighbour(population[i]);
119                // выбор одного из соседей
                    population[i] = rd.choice(n);

```

```

120         }
121     }
122 }
123 // вывод популяции
124 void printPopulation () {
125     for (int i = 0; i < this->population.size (); i++) {
126         cout << population [i] << " ";
127     }
128     cout << endl;
129 }
130 // метод для эволюционной итерации
131 void evolutionStep () {
132     this->crossing();
133     this->makeSelection();
134     this->mutation();
135 #ifdef PRINT_GA
136     this->printPopulation ();
137 #endif
138 }
139 // поиск вершины с минимальным эксцентриситетом после работы алгоритма
140 int getBestResult (double& time) {
141     this->startAlgorithm (time);
142     vector<int> e;
143     for (int i = 0; i < population . size () ; i++) {
144         int x = G->getEccentricity (population [i]);
145         e.push_back (x);
146     }
147     vector<int> ind = getCentralVertex (e);
148 #ifdef PRINT_GA
149     this->printPopulation ();
150 #endif
151     return this->G->getEccentricity (population [ind [0]]);
152 }
153 // получение популяции
154 vector<int> getPopulation () {
155     return this->population;
156 }
157 };

```

## ПРИЛОЖЕНИЕ Б

### Код классов для работы с графами

Далее приводится код класса GraphWork:

```
1 #pragma once
2
3 #include <vector>
4 #include <queue>
5 #include <iostream>
6 #include <algorithm>
7 #include <stdio.h>
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     // матрица смежности
25     vector<vector<int>> adjacency;
26     // матрица для расстояний между вершинами
27     vector<vector<int>> distance ;
28     // найденные расстояния между вершинами
29     vector<vector<vector<int>>> path;
30     // флаги на найденные эксцентриситеты
31     vector<bool> calculatedEccentricity ;
32     vector<int> eccentricity ;
33
34     void initializeMatrix () {
35         // начальная инициализация всех матриц
36         // матрица смежности
37         adjacency . resize ( this ->N, vector<int>());
38         // матрица n x n для расстояний
```

```

39         distance . resize ( this ->N, vector<int>( this ->N, INT_MAX));
40         // матрица n x n для путей
41         path . resize ( this ->N, vector<vector<int>>(this ->N));
42         // флаги для отображения был ли найден эксцентриситет этой вершины
43         calculatedEccentricity . resize ( this ->N, false);
44         // вектор для хранения найденных эксцентриситетов
45         eccentricity . resize ( this ->N, INT_MAX);
46         // инициализация матриц
47         for (int i = 0; i < this ->N; i++) {
48             this ->distance[i][i] = 0;
49             this ->path[i][i] = vector<int>(1, i);
50         }
51     }
52
53     public :
54         Graph(int n, int m, vector<pair<int, int>> adj) {
55             this ->N = n, this ->M = m;
56             initializeMatrix ();
57             // создание списка смежности для хранения для графа
58             for (int i = 0; i < m; i++) {
59                 int x = adj[i]. first , y = adj[i]. second;
60                 this ->adjacency[x].push_back(y);
61                 this ->adjacency[y].push_back(x);
62             }
63         }
64
65         // запуск обходя в ширину из переданной вершины
66         void bfsFromVertex(int x) {
67             // реализация классического алгоритма поиска в ширину
68             // очередь обхода
69             queue<int> q;
70             // добавление стартовой вершины
71             q.push(x);
72             // расстояния от стартовой вершины до всех остальных
73             vector<int> d( this ->N, INT_MAX);
74             // начальная инициализация расстояния
75             d[x] = 0;
76             // обход до тех пор, пока очередь не пуста
77             while (!q.empty()) {
78                 // извлечение из очереди
79                 int u = q. front ();

```

```

80     q.pop();
81     // получение пути от старта до рассматриваемой вершины
82     vector<int> currentPath = this->path[x][u];
83     // просмотр всех соседних вершин
84     for (unsigned int i = 0; i < this->adjacency[u].size(); i++) {
85         // извлечение соседней вершины
86         int v = this->adjacency[u][i];
87         // если вершина не была посещена, то
88         // она добавляется в очередь
89         if (d[v] > INT_MAX / 2) {
90             // добавление вершины в текущий путь
91             currentPath.push_back(v);
92             // сохранение найденного пути
93             this->path[x][v] = currentPath;
94             this->path[v][x] = currentPath;
95             currentPath.pop_back();
96             // увеличение найденного пути
97             d[v] = d[u] + 1;
98             // добавление вершины в очередь
99             q.push(v);
100        }
101    }
102  }
103  // сохранение найденных расстояний
104  for (unsigned int i = 0; i < d.size(); i++) {
105      this->distance[x][i] = d[i];
106      this->distance[i][x] = d[i];
107  }
108 }
109 // получение пути между вершинами,
110 // если путь еще неизвестен, то
111 // запускается обход в ширину
112 vector<int> getPath(int x, int y) {
113     if (this->path[x][y].empty()) {
114         this->bfsFromVertex(x);
115     }
116     return this->path[x][y];
117 }
118 // получение эксцентриситета вершины
119 int getEccentricity (int x) {
120     if (x >= this->N) {

```

```

121                     throw exception("Vertex doesn't exist in graph");
122                 }
123             // проверка, что эксцентриситет был найден ранее
124             if (!this->calculatedEccentricity [x]) {
125                 // запуск обхода
126                 this->bfsFromVertex(x);
127                 // пометка вершины, что ее эксцентриситет найден
128                 this->calculatedEccentricity [x] = true ;
129                 int result = -INT_MAX;
130                 // поиск самой удаленной вершины
131                 for (int i = 0; i < this->N; i++) {
132                     result = max(this->distance[x][i], result );
133                 }
134                 this->eccentricity [x] = result ;
135             }
136             return this->eccentricity [x];
137         }
138         // получение соседей заданной вершины
139         vector<int> getNeighbour(int x) {
140             return this->adjacency[x];
141         }
142         // получение размеров графа
143         int Size() {
144             return this->N;
145         }
146     };

```

### Методы для чтения графов:

```

1 // метод для чтения графа из файла
2 vector<pair<int, int>> readFromFileWhereEdges(string file , int& N, int& M) {
3     // создание файлового потока
4     ifstream in( file );
5     // ребра графа
6     vector<pair<int, int>> e;
7     string inputLine;
8     int n = 0, m = 0;
9     // построчное чтение файла
10    while ( getline (in, inputLine)) {
11        if ( inputLine[0] == '%') {
12            continue;
13        }
14        // увеличение счетчика ребер

```

```

15         m++;
16         stringstream s(inputLine);
17         // чтение вершин
18         int x, y;
19         s >> x >> y;
20         // увеличение размера графа
21         n = max(n, max(x, y));
22         // перевод в ноль индексацию
23         x--, y--;
24         e.push_back(make_pair(x, y));
25     }
26     N = n;
27     M = m;
28     return e;
29 }
30 // метод для аналогичного чтения
31 // файла с графом, в котором на первой строке указаны размеры графа
32 vector<pair<int, int>> readFromFileName(string file, int& N, int& M) {
33     ifstream in(file);
34     vector<pair<int, int>> e;
35     int n, m;
36     in >> n >> m;
37     for (int i = 0; i < m; i++) {
38         int x, y;
39         in >> x >> y;
40         e.push_back(make_pair(x, y));
41     }
42     N = n;
43     M = m;
44     return e;
45 }
```

Класс Repository для хранения имен файлов и радиусов графов:

```

1 #pragma once
2
3 #include <iostream>
4 #include <map>
5 #include <iostream>
6 #include <iomanip>
7 #include <algorithm>
8
9 #include "GraphWork.h"
```

```

10 #include "GeneticAlgorithm.h"
11 #include "OtherGA.h"
12 #include "FileWork.h"
13 #include " Entities .h"
14
15 using std :: make_pair;
16 using std :: ifstream ;
17 using std :: stringstream ;
18
19 using std :: pair ;
20
21
22 class Repository {
23 private :
24     struct GraphParams
25     {
26         string fileName;
27         int realR;
28         GraphParams(string fileN , int r) {
29             this ->fileName = fileN;
30             this ->realR = r;
31         }
32     };
33     // реальные предподсчитанные радиусы графов
34     vector<int> RADIUS_BA = { 4, 4, 5, 4, 5, 5, 5 };
35     vector<int> RADIUS_GEOM = { 9, 9, 8, 8, 8, 8, 8 };
36     vector<int> RADIUS_ER = { 5, 4, 4, 4, 3, 3, 3 };
37     int TEST = 4;
38
39     // Названия файлов с графиками
40     vector<string> BA_FILE = {
41         "BA_Graph\\BarabasiAlbertGraph1_M2.txt",
42         "BA_Graph\\BarabasiAlbertGraph2_M2.txt",
43         "BA_Graph\\BarabasiAlbertGraph3_M2.txt",
44         "BA_Graph\\BarabasiAlbertGraph4_M2.txt",
45         "BA_Graph\\BarabasiAlbertGraph5_M2.txt",
46         "BA_Graph\\BarabasiAlbertGraph6_M2.txt",
47         "BA_Graph\\BarabasiAlbertGraph7_M2.txt" };
48
49     vector<string> GEOM_FILE = {
50         "GEOM_Graph\\GeometricGraph1_R01.txt",

```

```

51     "GEOM_Graph\\GeometricGraph2_R01.txt",
52     "GEOM_Graph\\GeometricGraph3_R01.txt",
53     "GEOM_Graph\\GeometricGraph4_R01.txt",
54     "GEOM_Graph\\GeometricGraph5_R01.txt",
55     "GEOM_Graph\\GeometricGraph6_R01.txt",
56     "GEOM_Graph\\GeometricGraph7_R01.txt" };

57
58     vector<string> ER_FILE = {
59         "ERDOSRENYI_Graph\\ErdosRenyi1_P001.txt",
60         "ERDOSRENYI_Graph\\ErdosRenyi2_P001.txt",
61         "ERDOSRENYI_Graph\\ErdosRenyi3_P001.txt",
62         "ERDOSRENYI_Graph\\ErdosRenyi4_P001.txt",
63         "ERDOSRENYI_Graph\\ErdosRenyi5_P001.txt",
64         "ERDOSRENYI_Graph\\ErdosRenyi6_P001.txt",
65         "ERDOSRENYI_Graph\\ErdosRenyi7_P001.txt" };

66     // чтение графа в зависимости от переданного параметра,
67     // отвечающего за тип этого графа
68     GraphParams getGraphData(string graphType) {
69
70         string fileName;
71         int real_r = 0;
72         if (graphType == "BA_GRAPH") {
73             fileName = BA_FILE[TEST];
74             real_r = RADIUS_BA[TEST];
75             cout << "BA_GRAPH" << endl;
76         }
77         else if (graphType == "GEOM_GRAPH") {
78             fileName = GEOM_FILE[TEST];
79             real_r = RADIUS_GEOM[TEST];
80             cout << "GEOM_GRAPH" << endl;
81         }
82         else if (graphType == "ERDOSH_GRAPH") {
83             fileName = ER_FILE[TEST];
84             real_r = RADIUS_ER[TEST];
85             cout << "ERDOSH_GRAPH" << endl;
86         }
87         return GraphParams(fileName, real_r);
88
89
90     public :
91         // метод для получения графа из файла

```

```

92     GraphDescription getGraph( string graphType) {
93         // получение параметров графа
94         GraphParams params = this->getGraphData(graphType);
95         int n, m;
96         // чтение графа из файла
97         edges e = readFromFileName(params.fileName, n, m);
98         cout << "N = " << n << " M = " << m << endl;
99         return GraphDescription(n, m, e, params.realR);
100    }
101 };

```

Классы GraphDescription и GaTestResult:

```

1 #pragma once
2 #include <vector>
3
4 using std :: vector ;
5 using std :: pair ;
6
7 using edges = vector<pair<int, int>>;
8 // Сущности, отвечающие
9 // за передачу информации о графах между методами
10 enum GraphType
11 {
12     None = 0,
13     ER = 1,
14     BA = 2,
15     GEOM = 3,
16 };
17 // описание графа в виде набора вершин,
18 // размеров и радиуса графа
19 class GraphDescription {
20 public :
21     int n, m;
22     edges e;
23     int realR;
24     GraphDescription() {}
25
26     GraphDescription(int n, int m, edges e, int realR) {
27         this->n = n;
28         this->m = m;
29         this->e = e;
30         this->realR = realR;

```

```

31         }
32     };
33
34 // описание результата вычислительного теста
35 // параметры генетического алгоритма и значение функции
36 class GaTestResult {
37 public:
38     GaTestResult() {}
39
40     GaTestResult(double popSize, double pm, double pc, double functionValue) {
41         this ->pm = pm;
42         this ->pc = pc;
43         this ->popSize = popSize;
44         this ->functionValue = functionValue ;
45     }
46
47     double pc, pm;
48     int popSize;
49     double functionValue;
50 };

```

## ПРИЛОЖЕНИЕ В

### Код классов для экспериментов с алгоритмом

Далее приводится код класса Experiment:

```
1 // Класс отвечающий за проведение эксперимента
2 class Experiment {
3     private :
4         // число тестовых итераций
5         const int ITER = 100;
6         // Параметры графа
7         GraphDescription graph;
8         // функция для вычисления времени работы
9         // алгоритма с заданными параметрами
10        pair<double, double> calculateTimeErrorValue( int popSize, double pm, double pc ) {
11            // число ошибок и среднее время работы
12            int error = 0;
13            double sum_time = 0.0;
14            // итеративное выполнение тестов
15            for (int i = 0; i < ITER; i++) {
16                // инициализация графа
17                Graph* g1 = new Graph(graph.n, graph.m, graph.e);
18                // инициализация генетического алгоритма
19                GeneticAlgorithm genAlg(g1, popSize, pc, pm);
20                // замер времени работы алгоритма
21                double time = 0.0;
22                // получение найденного радиуса
23                int R = genAlg.getBestResult(time);
24                sum_time += time;
25                // проверка на верно найденный радиус
26                if (R != graph.realR) {
27                    error++;
28                }
29                delete g1;
30            }
31            // вычисление среднего времени работы и процента ошибок
32            double avg_time = sum_time / double(ITER);
33            double avg_error = error / double(ITER) * 100.0;
34            return make_pair(avg_time, avg_error);
35        }
36        // запуск алгоритма с одним из изменяющихся параметров
37        // popSize – размер популяции, probParam – значение одного из параметров
```

```

38 // pmProb – флаг, отвечающий за то какой из параметров передан в качестве
39 // фиксированного
40 // true – параметр pm, false – параметр pc
41 pair<vector<GaTestResult>, vector<GaTestResult>> testWithChangebleProb(int
42 popSize, double probParam, bool pmProb) {
43     // шаг, с которым будет перебираться параметр
44     double step = 0.1;
45     // число итераций для перебора параметров
46     int itarationCount = ceil(1.0 / step);
47     // вывод информации о параметрах
48     cout << "popSize = " << popSize;
49     if (pmProb) {
50         cout << " pm = " << probParam << endl;
51     }
52     else {
53         cout << " pc = " << probParam << endl;
54     }
55     // вывод шапки для данных
56     if (pmProb) {
57         cout << "pc \t AVG_TIME \t ERROR" << endl;
58     }
59     else {
60         cout << "pm \t AVG_TIME \t ERROR" << endl;
61     }
62     // переменные для хранения результатов измерений
63     vector<GaTestResult> time;
64     vector<GaTestResult> error ;
65     // перебор одного из параметров
66     for (int i = 0; i <= itarationCount ; i++) {
67         double pm, pc;
68         // в зависимости от флага pmProb перебирается или параметр
69         // pm или pc
70         if (pmProb) {
71             pm = probParam;
72             pc = i * step ;
73         }
74         else {
75             pc = probParam;
76             pm = i * step ;
77         }
78         // вычисление среднего времени работы и процента ошибок

```

```

76         pair<double, double> metering = this ->calculateTimeErrorValue(
77             popSize, pm, pc);
78         double avg_time = metering. first ;
79         double error_percent = metering. second;
80         // вывод информации о полученных результатах
81         cout << (pmProb ? pc : pm) << "\t" << avg_time << "\t" <<
82             error_percent << endl;
83         time.push_back(GaTestResult(popSize, pm, pc, avg_time));
84         error.push_back(GaTestResult(popSize, pm, pc, error_percent));
85     }
86 }
87
88 public :
89     Experiment(GraphDescription graph) {
90         srand(time(NULL) % INT_MAX);
91         this ->graph = graph;
92     }
93     // Тест с изменяющимся параметром pc
94     pair<vector<GaTestResult>, vector<GaTestResult>> oneDimentionFixedGATest() {
95         // параметры алгоритма для теста
96         int populationN = 20;
97         double pm = 0.4;
98         return this ->testWithChangebleProb(populationN, pm, true);
99     }
100    // запуск простого теста с переданными параметрами
101    pair<double, double> simpleTimeErrorTest(double pm, double pc, int popSize) {
102        return this ->calculateTimeErrorValue(popSize, pm, pc);
103    }
104    // запуск алгоритма N4N
105    void simpleNANTest() {
106        // действия аналогичны – измеряется среднее
107        // время работы и процент ошибки
108        int error = 0;
109        double avg_time = 0.0;
110        for (int i = 0; i < ITER; i++) {
111            Graph* g1 = new Graph(graph.n, graph.m, graph.e);
112            cout << "Start OTHER genetic algorithm" << endl;
113            SimpleGeneticAlgorithm sgen(g1, 50, 10, 0.7, 0.1);
114            double time = 0.0;

```

```

115         int R = sgen. getBestResult (time);
116         avg_time += time;
117         if (R != graph.realR)
118             error++;
119     }
120     cout << "AVG Time = " << avg_time / double(ITER) << endl;
121     cout << "Error = " << double(error) / double(ITER) << endl;
122 }
123 // тестирование алгоритма с перебором параметров рт и pc
124 pair<vector<GaTestResult>, vector<GaTestResult>> pmpcGaTest() {
125     int popSize = 20;
126     double step = 0.1;
127     double pm = 0.0;
128     int sectionNumber = ceil (1.0 / step);
129     vector<GaTestResult> time;
130     vector<GaTestResult> error;
131     for (int i = 0; i <= sectionNumber; i++) {
132         pm = i * step;
133         pair<vector<GaTestResult>, vector<GaTestResult>> metrings = this
134             ->testWithChangebleProb(popSize, pm, true);
135         time. insert (time.end(), metrings. first .begin(), metrings. first .
136             end());
137         error . insert (error .end(), metrings .second.begin(), metrings .
138             second.end());
139     }
140     return make_pair(time, error);
141 }
142 // тестирование алгоритма с перебором
143 // размера популяции
144 void nGATest() {
145     double pm = 0.2;
146     double pc = 0.3;
147     int maxN = 50;
148     cout << "pm = " << pm << " pc = " << pc << endl;
149     vector<double> time;
150     vector<double> error;
151     for (int popSize = 1; popSize < maxN; popSize++) {
152         pair<double, double> metering = this->calculateTimeErrorValue(
153             popSize, pm, pc);
154         time.push_back(metering. first );
155         error .push_back(metering.second);

```

```
152             cout << "(" << popSize << ", " << metering.second << ")" << endl;
153         }
154     for (int i = 0; i < time.size(); i++) {
155         cout << "(" << i + 1 << ", " << time[i] << ")" << endl;
156     }
157 }
158 };
```