

**МИНОБРНАУКИ РОССИИ**  
**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ**  
**ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**  
**«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)**  
**Кафедра МОЭВМ**

**ОТЧЕТ**  
**по учебной практике**  
**Тема: Задача о назначениях**

Студенты гр. 0304

Руководитель

\_\_\_\_\_

\_\_\_\_\_

Решоткин А.С.  
Докучаев Р.А.  
Крицын Д.Р.  
Козиков А.Е.

Жангиров Т.Р.

Санкт-Петербург

2022

## ЗАДАНИЕ НА УЧЕБНУЮ ПРАКТИКУ

Студенты Решоткин А.С., Докучаев Р.А., Крицын Д.Р., Козиков А.Е.

Группа 0304

Тема практики: Задача о назначениях.

Задание на практику:

Пусть имеется  $N$  работ и  $N$  кандидатов на выполнение этих работ, причем назначение  $j$ -й работы  $i$ -му кандидату требует затрат  $c_{ij} > 0$ .

Необходимо назначить каждому кандидату по работе, чтобы минимизировать суммарные затраты. Причем каждый кандидат может быть назначен на одну работу, а каждая работа может выполняться только одним кандидатом.

Сроки прохождения практики: 29.06.2022 – 12.07.2022

Дата сдачи отчета: 2.07.2022

Дата защиты отчета: 2.07.2022

Студенты

\_\_\_\_\_

Решоткин А.С.  
Докучаев Р.А.  
Крицын Д.Р.,  
Козиков А.Е.

Руководитель

\_\_\_\_\_

Жангиров Т.Р.

## **АННОТАЦИЯ**

Основная задача работы заключается в знакомстве и применении на практике генетических алгоритмов, а также их оптимизаций, для решения поставленной задачи о назначениях. Генетические алгоритмы (далее ГА) — это адаптивные методы поиска, которые в последнее время используются для решения задач оптимизации. В них используются как аналог механизма генетического наследования, так и аналог естественного отбора. При этом сохраняется биологическая терминология в упрощенном виде и основные понятия линейной алгебры.

## СОДЕРЖАНИЕ

	Введение	5
1.	Итерация 2	6
1.1.	Скетч с GUI, который планируется реализовать.	7
1.2.	Описание сценариев взаимодействия пользователя с программой	7
1.3	Определение и обоснование параметров модификации ГА для решения задачи.	7
2.	Итерация 3	9
2.1	Частичная реализация GUI	9
2.2	Реализовано хранение данных и основные элементы ГА.	12
3.	Итерация 4	13
3.1	GUI	13
3.2	Описание алгоритма	14
4.	Итерация 5	15
4.1	Доработки	15
4.2	GUI	15
	Заключение	16
	Список использованных источников	17
	Приложение А. Исходный код программы	18

## **ВВЕДЕНИЕ**

Основной целью работы является решение задачи о назначениях, в которой требуется минимизировать суммарные затраты на работу. Для этого используется генетический алгоритм с некоторыми модификациями для оптимальной работы, такими как гибрид элитарного отбора и отбор усечением, рандомизированный многоточечный кроссинговер. На вход программе должно подаваться количество работ и кандидатов  $N$  и матрица стоимостей  $C$ .

## 1. Итерация 2

### 1.1. Скетч с GUI, который планируется реализовать.

		-	X
<b>Настройка:</b>			
Ввод значений:			
Овыбрать файл...		<input type="radio"/> Ввести в программе	
Параметры ГА:			
Размер начальной популяции:	<input type="text"/>		
Критерий остановки:	<input type="text"/>		
Коэффициент влияния:	<input type="text"/>		
Плотность мутации:	<input type="text"/>		
количество особей при элитарном отборе:	<input type="text"/>		

		-	X
<b>Пошаговая визуализация:</b>			
Начальная популяция:			
Особи	<input type="text"/>	<input type="text"/>	
приспособленность	<input type="text"/>	<input type="text"/>	
Выбор родителей:			
Особи	<input type="text"/>	<input type="text"/>	
Вероятность	<input type="text"/>	<input type="text"/>	
Кроссинговер:			
Родители	<input type="text"/>	<input type="text"/>	<input type="text"/>
Потомки	<input type="text"/>		<input type="text"/>

		-	X
<b>Ввод данных:</b>			
<input type="text" value="Введите значение N"/>			
кандидаты:	1	...	
Работы:			
1			
...			

		-	X
<b>Пошаговая визуализация:</b>			
Мутации:			
До мутации	<input type="text"/>	<input type="text"/>	
После мутации	<input type="text"/>	<input type="text"/>	
Новая популяция:			
Особи	<input type="text"/>	<input type="text"/>	
лучшая особь:	<input type="text"/>		
<input type="button" value="К ответу &gt;&gt;"/>		<input type="button" value="Следующий шаг &gt;"/>	

		-	X
<b>Результат:</b>			
Распределение работ:			
Работник	<input type="text"/>	<input type="text"/>	
Работа	<input type="text"/>	<input type="text"/>	
Суммарные затраты:	<input type="text"/>		
<input type="button" value="Задать новые данные"/>			

## **1.2. Описание сценариев взаимодействия пользователя с программой**

Возможный сценарий взаимодействия пользователя с программой представлен ниже.

- 1) Пользователь запускает программу.
- 2) Пользователю предоставляется выбор: ввести данные из файла или ввести данные в меню.
- 3) Если пользователь выбирает ввод данных из файла, то программа проверяет корректность введенных данных и запускает алгоритм, выводя промежуточные итерации в виде таблицы.
- 4) Если пользователь выбирает ввод данных в меню, то программа соответственно проверяет наличие всех необходимых данных для решения задачи, а затем решает задачу в соответствии с алгоритмом, выводя промежуточные результаты в виде таблицы.
- 5) Пользователь на выходе получает ответ на задачу в виде таблицы.

## **1.3 Определение и обоснование параметров модификации ГА для решения задачи.**

Пробные решения для генетического алгоритма (хромосомы) представлены в виде перестановок, которые будут переведены в числа в факториальной системе счисления при помощи кода Лемера, которые затем будут переведены в двоичную систему счисления.

Оператор выбора родителей - метод рулетки. Выбор обоснован тем, что при настройке макропараметров можно настроить формулу отбора, изменив влияние приспособленности на выбор родителей: это может быть полезно, так как расположение и количество локальных минимумов зависят от поданной на вход матрицы затрат.

Оператор рекомбинации (кроссинговера) - многоточечный кроссинговер. В отличие от кроссинговера с фиксированным количеством точек, такой метод

позволяет лучше адаптироваться к разным размерностям входной матрицы затрат, в то же время не затрачивая много времени на генерацию случайной двоичной строки, как в методе однородного кроссинговера.

Оператор мутации - мутация с использованием понятия плотности. В отличие от двоичной мутации, работает без особых отличий для двоичных строк разной размерности, что хорошо подходит для данной задачи (размерность матрицы затрат заранее неизвестна).

Оператор отбора в новую популяцию - элитарный отбор в сочетании с отбором усечением. Выбор обоснован тем, что содержимое поданной на вход матрицы заранее неизвестно, и при не сильно отличающихся значениях элементов такой матрицы элитарный отбор не будет терять решения, сошедшиеся к локальным экстремумам, которых может быть довольно много. При этом слишком неперспективные решения рассматривать нет особого смысла - лучше отсеять их и попытаться выйти из локального минимума за счёт выбора случайных хорошо приспособленных особей через отбор усечением и их дальнейшей мутации.



## 2. Итерация 3

### 2.1 Частичная реализация GUI.

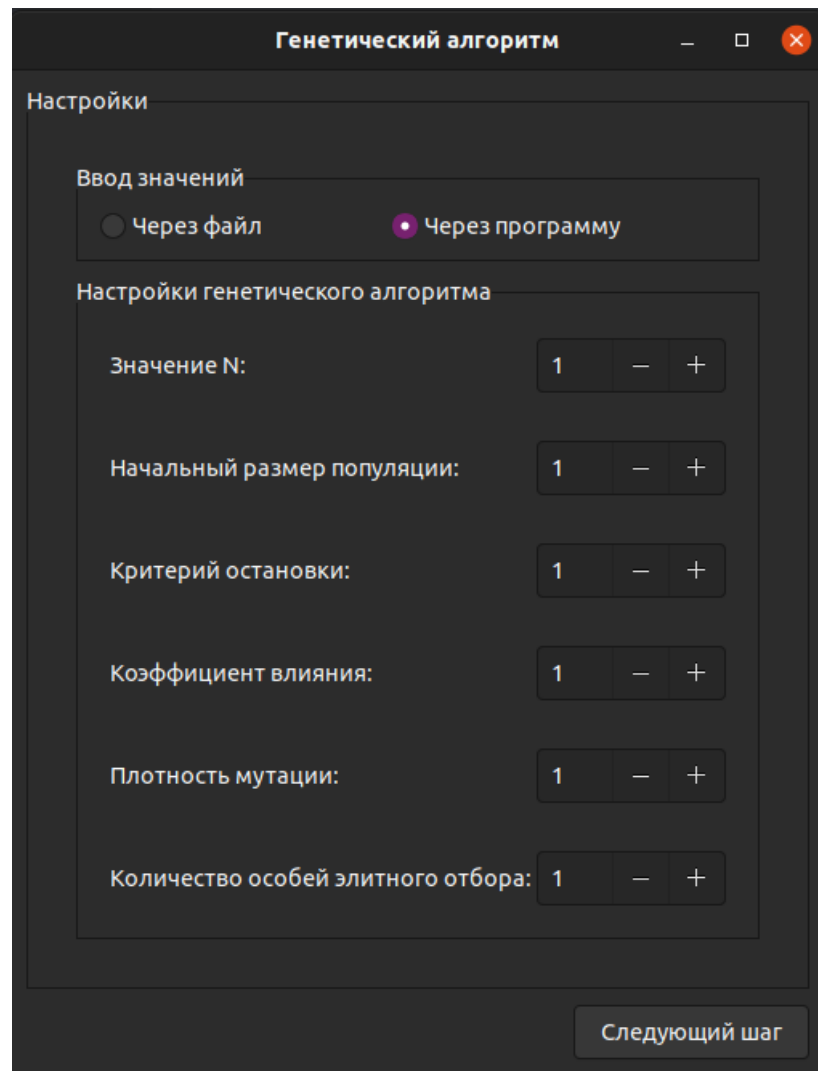


Рисунок 1. Главное окно.

Генетический алгоритм

Введите значения:

	1			2			3			4		
1	1	-	+	1	-	+	1	-	+	1	-	+
2	1	-	+	1	-	+	1	-	+	1	-	+
3	1	-	+	1	-	+	1	-	+	1	-	+
4	1	-	+	1	-	+	1	-	+	1	-	+
5	1	-	+	1	-	+	1	-	+	1	-	+

Следующий шаг

Рисунок 2. Окно ввода данных.

Генетический алгоритм

Пошаговая визуализация:

Особи	Приспособленность
34	32
23	23

Особи	Вероятность
34	32
23	23

Родители	Потомки
34	32
23	23

До мутации	После мутации
34	32
23	23

Новая популяция	
34	32
23	23

Лучшая особь	
34	32
23	23

Следующий шаг

К результату

Рисунок 3. Пошаговая визуализация.

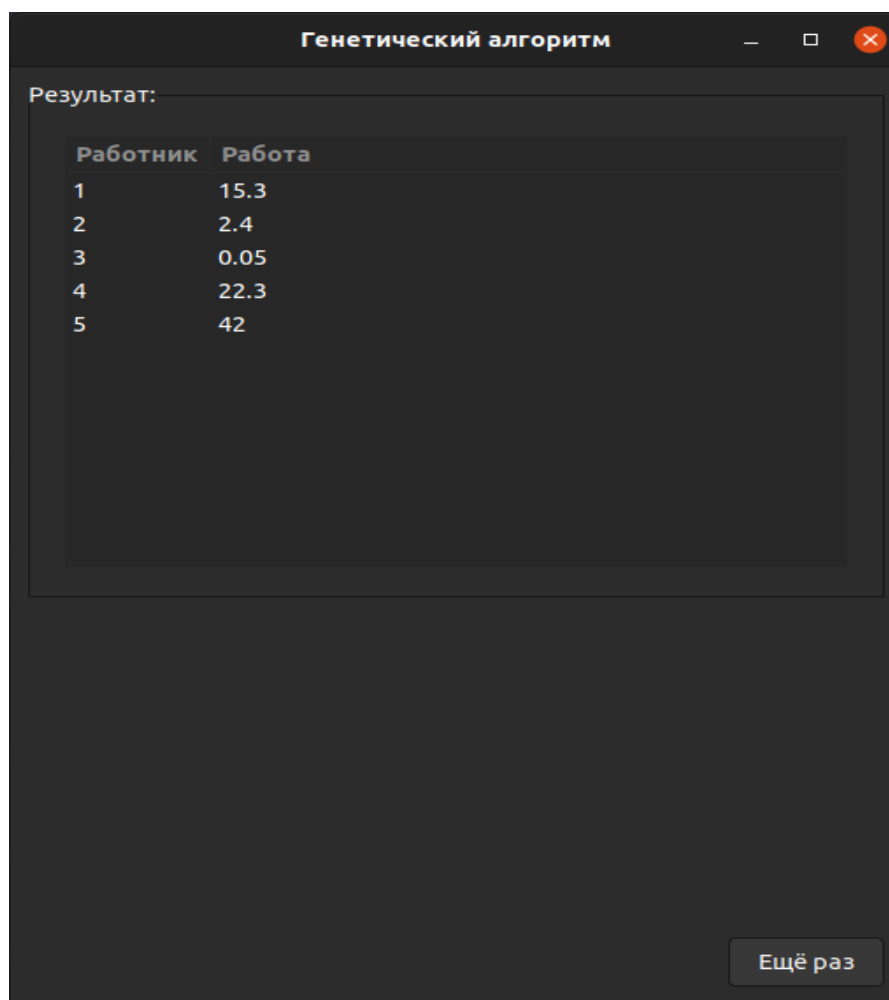


Рисунок 4. Результат работы.

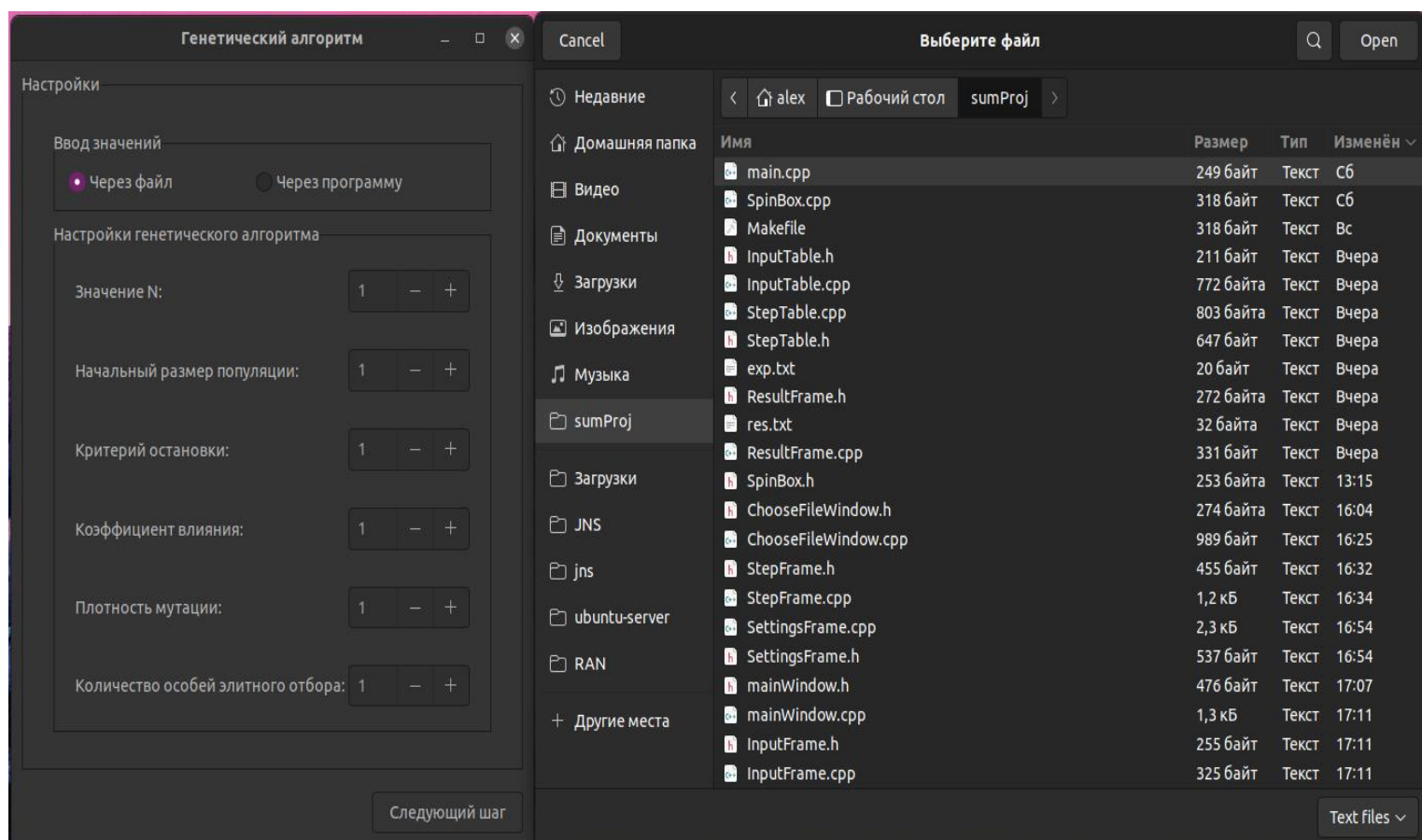


Рисунок 5. Окно для выбора файла.

Что реализовано в GUI:

Поля для настройки параметров и способ задания данных. Границы макропараметров ГА всегда зависят от заданного значения  $N$ , значения для всех пунктов могут быть только больше 0. При выборе через файл открывается меню выбора. При выборе через программу, открывается матрица "работник x работа", в которой можно задать неотрицательные значения для сложности. Далее происходит пошаговый вывод работы ГА, с возможностью перейти к результату. В результате выводится таблица работник и подходящая ему работа. При желании можно запустить программу вновь, не закрывая её.

## 2.2 Реализовано хранения данных и основные элементы ГА

1) Хранение данных реализовано с помощью вектора `current_population` (вектора двоичных строк).

2) Основные элементы ГА реализованы с помощью функций:

a) `fitness_func` - функция приспособленности. Чем больше значение, тем больше приспособленность (знак учитывается).

b) `recombination_op` - оператор рекомбинации. Возвращает одного или несколько потомков двух родителей. Предполагается, что количество генов в обоих родителях совпадает (в большинстве видов такого оператора).

c) `mutation_op` - оператор мутации. Возвращает модифицированную хромосому.

d) `parent_selection_op` - оператор выбора родителей. Выбор производится из поданного на вход списка особей.

e) `survivor_selection_op` - оператор отбора в новую популяцию. Напрямую модифицирует поданный на вход вектор.

Данные функции задаются в конструкторе класса `GARunner`. Функция `do_iteration` производит одну итерацию ГА: выбирает родителей, формирует потомков, создает новую популяцию и возвращает количество потомков. Функция `thread_iterate_count` совершает заданное количество итераций, функция

thread\_iterate\_until\_no\_children совершает итерации пока не будет произведено ни одного потомка. Thread\_terminate останавливает генетический алгоритм.

### 3. Итерация 4

#### 3.1 GUI.

В данной итерации были доделаны элементы GUI: матрица ввода (InputTable, рис. 7) через GUI корректно работает с числами с точностью до сотых и пересоздаётся с каждым запуском алгоритма. Была доделана настройка макропараметров (SettingsFrame, рис. 6). В GUI код были добавлены поясняющие комментарии. Также GUI было связано с работой генетического алгоритма, с помощью методов:

- В MainWindow добавлен метод StepFrame::initGARunner(SettingsFrame& sfr, InputTable& ifr), принимающий на вход ссылку на настройки параметров и таблицу ввода. Инициализирует работу генетического алгоритма.
- В класс StepFrame были добавлены методы, которые заполняют таблицу данными из генетического алгоритма: GACBIterationStart() — инициализация таблицы, GACBNewChildren() — добавление потомков, GACBMutation() — таблица мутаций потомков, GACBNewPopulation() — таблица новой популяции.
- Для класса SettingsFrame были написаны методы, возвращающие одноимённые данные класса. Для того, чтобы не записывать их файл, а напрямую передавать алгоритму.
- Для класса InputFrame была разработана функция addTable(unsigned N), создающая InputTable размера N, эта же функция была перегружена как addTable(std::string filePath), если ввод данных происходил через файл, а не GUI. Через метод getMatrix() можно получить матрицу вещественных чисел, которые были введены через файл или GUI.

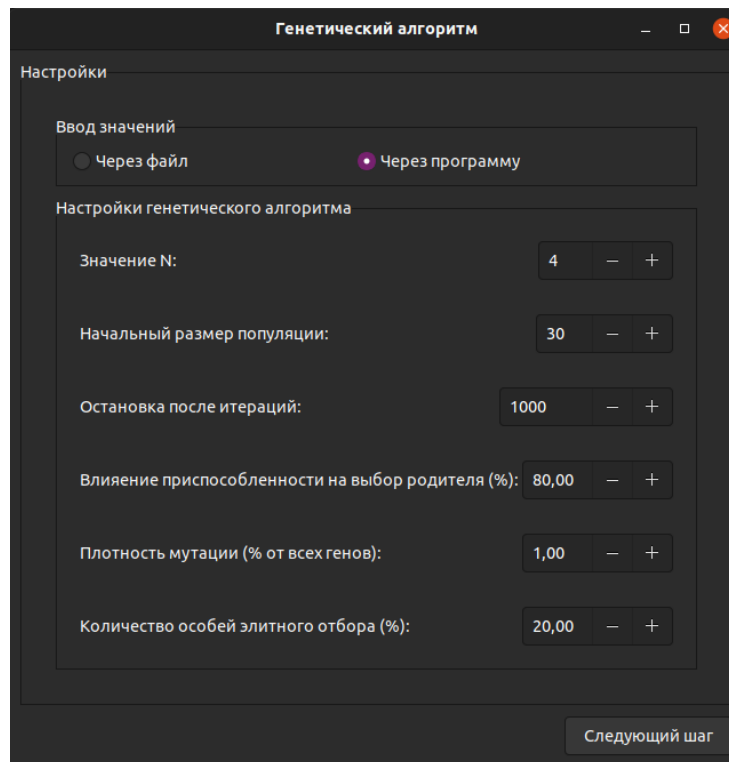


Рисунок 6. Новое окно настроек

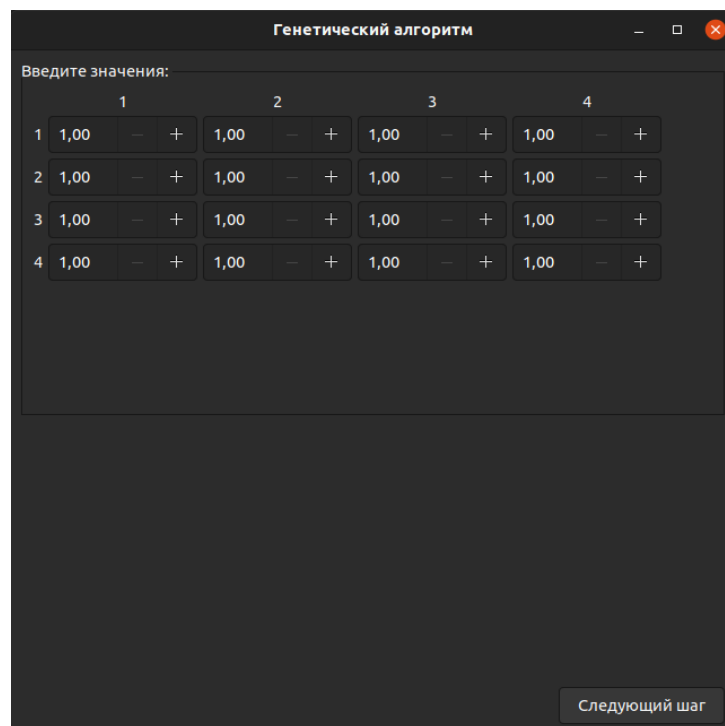


Рисунок 7. Новое окно матрицы ввода данных через GUI

### 3.2 Описание алгоритма.

Реализован алгоритм, выполняющий операции по работе с перестановками. Создана функция, генерирующая случайные перестановки `generate_rand_perms`. Также разработана функция (`perm_fitness_func`), проверяющая, совпадает ли размер перестановки с размером матрицы. Если это не так, то решение невер-

ное и оно имеет минимальную выживаемость. Также в данной функции происходит суммирование стоимостей каждой работы. Параллельно с этим реализован ряд функций, работающий с переводом перестановки в булеву строку и наоборот. Функция `biolog` отвечает за подсчёт логарифма целого числа. Функция `perm_to_bool_string` выполняет конвертацию перестановки в булеву строку. Берётся число и уменьшаются числа, стоящие справа от него и превосходящие его (код Лемера). Обратную операцию также при помощи кода Лемера выполняет функция `bool_string_to_perm`.

## 4. Итерация 5

### 4.1 Доработки.

- Добавлена обработка ошибок, а именно ошибочных файлов (с недостаточным количеством элементов для заданного  $N$  или которые просто невозможно считать из-за прав доступа).
- Добавлен вывод информации в файл. Название файла – `ga_out.txt`. Создание файла происходит в рабочей директории программы.

### 4.2 GUI.

В интерфейсе программы был добавлен класс всплывающего окна об ошибке `ErrorWindow`, конструктор которого принимает две строки (название ошибки и её описание) и запускает окно. Данный класс используется в отлавливании возможных ошибок, которые возникают в процессе открытия и чтения файла.

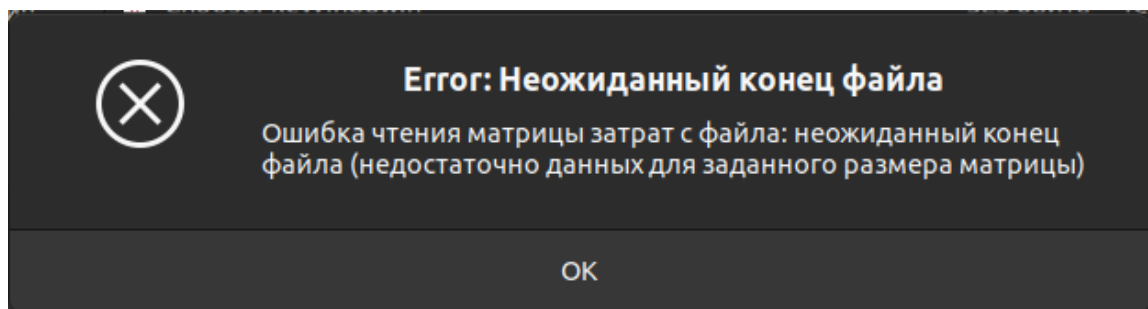


Рисунок 8. Пример всплывающего окна об ошибке

В интерфейс был добавлен индикатор выполнения для ожидания исполнения алгоритма при переходе к результату. Объект `activityBar` класса `Gtk::ProgressBar` был добавлен в `mainWindow` и отвечает за отображение индикатора (рис. 9). Благодаря функции `Glib::signal_timeout()` происходит вызов функции `timeCheck()` каждые 50 микросекунд, которая активирует индикатор и проверяет условия завершения работы алгоритма.

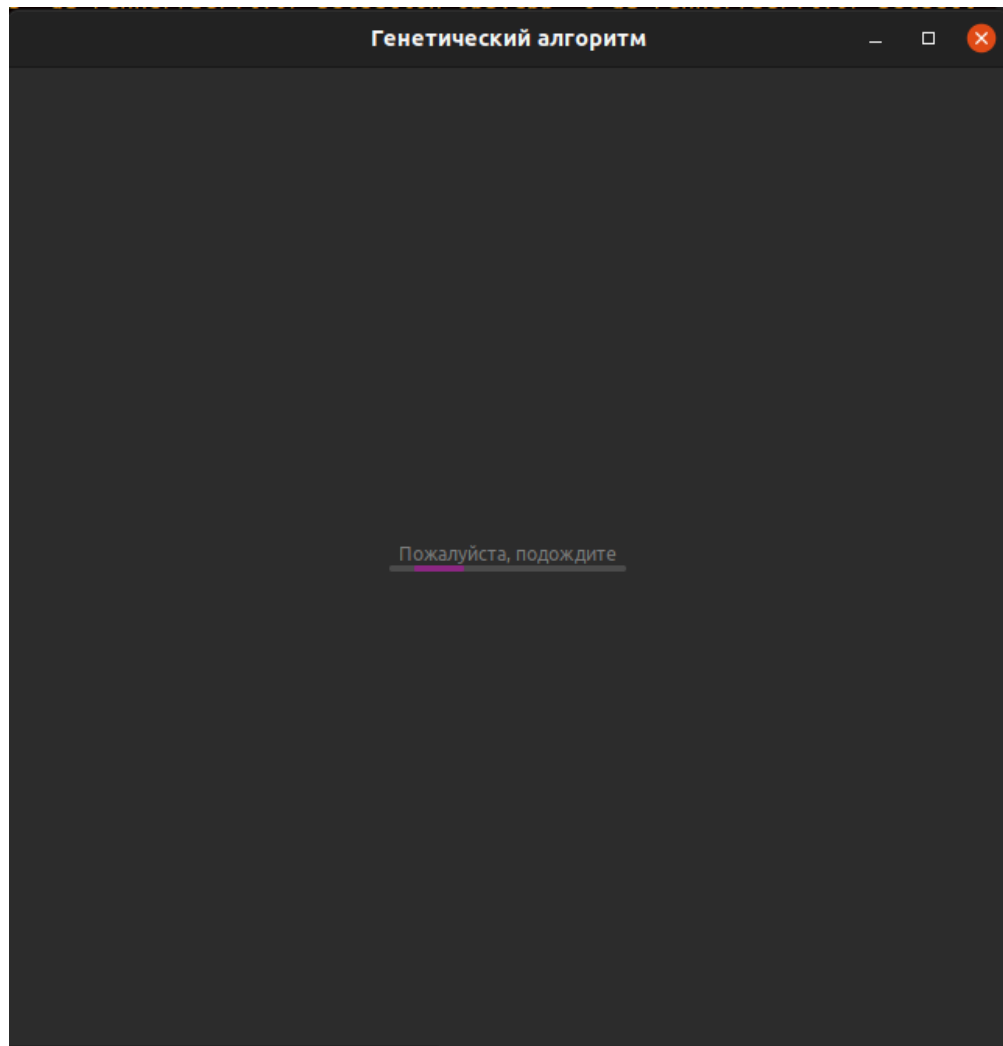


Рисунок 9. Индикатор исполнения алгоритма.



## **ЗАКЛЮЧЕНИЕ**

На второй итерации практики удалось создать скетч с графическим интерфейсом, который будет в программе, определить сценарий взаимодействия пользователя с программой, определить и обосновать модификации ГА, которые были выбраны для решения поставленной задачи. На третьей итерации было реализовано GUI и основные элементы ГА. На последующих итерациях вносились некоторые изменения в программу для оптимизации её работы и фикса багов, а также GUI был связан с алгоритмом. В итоге была решена задача о назначениях при помощи генетического алгоритма. Были реализованы алгоритмы, генерирующие случайные перестановки и затем работающие с ними. Затем сгенерированные случайные перестановки в виде двоичных строк, предварительно обработанные при помощи кода Лемера (популяция), обрабатываемые генетическим алгоритмом: к данной популяции применяем операторы ГА.

## **СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ**

1. Панченко Т.В. Учебно-методическое пособие “Генетический алгоритмы”.

## ПРИЛОЖЕНИЕ А

### ИСХОДНЫЙ КОД ПРОГРАММЫ

#### Файл bool\_string.cpp

```
#include "bool_string.h"

std::ostream& operator<<(std::ostream& os, BoolString bs)
{
    for(size_t i = 0; i < bs.size(); ++i)
        os << bs[i];
    return os;
}
```

#### Файл bool\_string.h

```
#ifndef BOOL_STRING_H
#define BOOL_STRING_H

#include <ostream>
#include <vector>

class BoolString : public std::vector<bool>
{
};

std::ostream& operator<<(std::ostream& os, BoolString bs);

#endif
```

#### Файл ga\_runner.h

```
#ifndef GA_RUNNER_H
#define GA_RUNNER_H

#include <iostream>
#include <vector>
#include <random>
#include <pthread.h>

#include "bool_string.h"

template<typename data>
class GARunner
{
private:
    size_t population_size;
    std::vector<data> current_population;

    double mutation_chance;
    double (*fitness_func)(data);

    std::vector<data> (*recombination_op)(data, data);
    data (*mutation_op)(data);
    std::pair<data, data> (*parent_selection_op)(std::vector<data>, double (*fitness_func_arg)(data));
    std::vector<data> (*survivor_selection_op)(std::vector<data>, size_t amount, double (*fitness_func_arg)(data));
};
```

```

std::random_device dev_urandom;

void print_data(std::vector<data> wh)
{
    for(size_t i = 0; i < wh.size(); ++i){
        std::cout << wh[i] << ' ';
    }
    std::cout << '\n';
}

pthread_t work_thread;

pthread_mutex_t thread_mode_mutex;
enum {
    GATerminated,
    GADoNIterations,
    GADoUntilNoChildren
} thread_mode;
size_t iter_counter;

friend void* thread_iteration(void* _args);
static void* thread_iteration(void* _args)
{
    GARunner* self = (GARunner*)_args;
    size_t prev_children = 1;
    while(1){
        pthread_mutex_lock(&self->thread_mode_mutex);
        switch(self->thread_mode){
            case GATerminated: goto terminate;
            case GADoNIterations: if(self->iter_counter == 0)
goto terminate; else --self->iter_counter; break;
            case GADoUntilNoChildren: if(prev_children == 0)
goto terminate; break;
        }
        pthread_mutex_unlock(&self->thread_mode_mutex);
        prev_children = self->do_iteration();
    }
    terminate:
    self->thread_mode = GATerminated;
    pthread_mutex_unlock(&self->thread_mode_mutex);
    return NULL;
}

public:
    std::vector<data>& get_current_population() { return cur-
rent_population; }

    GARunner(size_t population_size, std::vector<data> ini-
tial_population,
              double mutation_chance,
              double (*fitness_func)(data),
              std::vector<data> (*recombination_op)(data, data),
              data (*mutation_op)(data),
              std::pair<data, data>
(*parent_selection_op)(std::vector<data>, double (*fitness_func_arg)(data)),
              std::vector<data> survivor_selection_op(std::vector<data>,
size_t amount, double (*fitness_func_arg)(data)))
        : population_size(population_size), cur-
rent_population(initial_population), mutation_chance(mutation_chance),
        fitness_func(fitness_func), recombination_op(recombination_op), mu-
tation_op(mutation_op), parent_selection_op(parent_selection_op), survi-
vor_selection_op(survivor_selection_op)

```

```

{
    thread_mode = GATerminated;
    iter_counter = 0;
    pthread_mutex_init(&thread_mode_mutex, NULL);
}

size_t do_iteration()
{
    std::vector<data> children;
    for(size_t i = 0; i <= population_size; i += 2){
        // отбираем двух родителей
        std::pair<data, data> parents = parent_selection_op(current_population, fitness_func);
        std::cout << "parents: "; print_data({parents.first,
parents.second});

        if(parents.first != parents.second){
            std::vector<data> more_children = recombination_op(parents.first, parents.second); // проводим скрещивание
            std::copy(more_children.begin(),
more_children.end(), std::back_inserter(children));
        }
        std::cout << "children: "; print_data(children);

        std::mt19937 rng(dev_urandom());
        std::uniform_real_distribution<double> mutation_dist(0, 1);
        for(size_t i = 0; i < children.size(); ++i)
            if(mutation_dist(rng) < mutation_chance)
                children[i] = mutation_op(children[i]); //
производим мутацию с заданным шансом
        std::cout << "mutated children: "; print_data(children);

        // заносим детей и родителей в один список и отбираем оттуда
особей для следующей популяции
        std::copy(children.begin(), children.end(),
std::back_inserter(current_population));
        current_population = survivor_selection_op(current_population,
population_size, fitness_func);

        std::cout << "current population: ";
print_data(current_population);

        return children.size();
    }

void thread_terminate()
{
    pthread_mutex_lock(&thread_mode_mutex);
    thread_mode = GATerminated;
    pthread_mutex_unlock(&thread_mode_mutex);
    pthread_join(work_thread, NULL);
}

void thread_iterate_count(size_t count)
{
    bool create_thread = (thread_mode == GATerminated);
    pthread_mutex_lock(&thread_mode_mutex);
    iter_counter = count;
    thread_mode = GADoNIterations;
    pthread_mutex_unlock(&thread_mode_mutex);
    if(create_thread)
        pthread_create(&work_thread, NULL, GARunner::thread_iteration, (void*)this);
}

```

```

        void thread_iterate_until_no_children()
        {
            bool create_thread = (thread_mode == GATerminated);
            pthread_mutex_lock(&thread_mode_mutex);
            thread_mode = GADoUntilNoChildren;
            pthread_mutex_unlock(&thread_mode_mutex);
            if(create_thread)
                pthread_create(&work_thread, NULL, GARun-
ner::thread_iteration, (void*)this);
        }

        void print_current_population() { print_data(current_population); }
};

#endif

```

## Файл mutation\_ops.cpp

```

#include "mutation_ops.h"
#include <random>

static std::random_device dev_urandom;
static std::mt19937 rng(dev_urandom());

double density_mutation_chance = 0.01;
BoolString density_mutation_op(BoolString bs)
{
    BoolString _out = bs;
    std::uniform_real_distribution<double> mut_dist(0, 1);
    for(size_t i = 0; i < bs.size(); ++i){
        if(mut_dist(rng) <= density_mutation_chance)
            _out[i] = !_out[i];
    }
    return _out;
}

```

## Файл mutation\_ops.h

```

#ifndef MUTATION_OPS_H
#define MUTATION_OPS_H

#include "ga_runner.h"

/* Мутация бинарных строк */

// Мутация с использованием понятия плотности
extern double density_mutation_chance; // шанс мутации (инверсии) каждого ге-
на: [0; 1], по умолчанию 0.01
BoolString density_mutation_op(BoolString bs);

```

```

#endif

```

Файл parent\_selection\_ops.cpp

```

#include "parent_selection_ops.h"
#include <random>

static std::random_device dev_urandom;
static std::mt19937 rng(dev_urandom());

double roulette_fitness_influence = 0.8;

```

```

std::pair<BoolString, BoolString> roulette_bs_selection_op(std::vector<BoolString> individuals,

double (*fitness_func)(BoolString))
{
    double fsum = 0; // сумма всех значений функций приспособленности
    for(size_t i = 0; i < individuals.size(); ++i)
        fsum += fitness_func(individuals[i]);

    BoolString p1, p2; unsigned pcnt = 0;
    std::uniform_real_distribution<double> roulette_dist(0, 1);
    double roll1 = roulette_dist(rng), roll2 = roulette_dist(rng);

    double fcur = 0;
    for(size_t i = 0; i < individuals.size(); ++i){
        double finc = (fitness_func(individuals[i]) / fsum) * roulette_fitness_influence + (1. / individuals.size()) * (1 - roulette_fitness_influence);
        if(fcur < roll1 && fcur + finc >= roll1) { p1 = individuals[i]; ++pcnt; }
        if(fcur < roll2 && fcur + finc >= roll2) { p2 = individuals[i]; ++pcnt; }
        if(pcnt == 2) break;
        fcur += finc;
    }
    return {p1, p2};
}

```

## Файл parent\_selection\_ops.h

```

#ifndef PARENT_SELECTION_OPS_H
#define PARENT_SELECTION_OPS_H

#include "ga_runner.h"

/* Выбор родителей для бинарных строк */

// Метод рулетки
extern double roulette_fitness_influence; // влияние значения функции приспособленности на шанс хромосомы быть выбранной в методе рулетки: [0, 1]
std::pair<BoolString, BoolString> roulette_bs_selection_op(std::vector<BoolString> individuals,

double (*fitness_func)(BoolString));

#endif

```

## Файл recombination\_ops.cpp

```

#include "recombination_ops.h"
#include <random>
#include <cmath>

static std::random_device dev_urandom;
static std::mt19937 point_rng(dev_urandom());
std::vector<BoolString> multi_point_crossingover(BoolString bs1, BoolString bs2)
{
    std::vector<size_t> points;
    // Генерируем ceil(ln(N)) уникальных точек разреза
    std::uniform_int_distribution<size_t> point_dist(0, bs1.size() - 1);
    for(size_t i = 0; i < ceil(log(bs1.size())); ++i){

```

```

        while(1){
            size_t pt = point_dist(point_rng);
            int repeat = 0;
            for(size_t j = 0; j < i; ++j)
                if(points[j] == pt)
                    { repeat = 1; break; }
            if(!repeat){
                points.push_back(pt);
                break;
            }
        }
        std::sort(points.begin(), points.end());
        // Генерируем первого потомка: берём первую булеву строку и вставляем в
        неё фрагменты из второй согласно точкам разреза
        BoolString ch1 = bs1;
        for(size_t i = 0; i < points.size(); i += 2){
            if(i != points.size() - 1){
                std::copy(bs2.begin() + points[i], bs2.begin() + points[i +
1], ch1.begin() + points[i]);
            }
            else{ // обмен генами с закольцовыванием от конца в начало
                std::copy(bs2.begin() + points[i], bs2.end(), ch1.begin() +
points[i]);
                std::copy(bs2.begin(), bs2.begin() + points[0], ch1.begin());
            }
        }
        // Генерируем второго потомка: берём вторую булеву строку и вставляем в
        неё фрагменты из первой согласно точкам разреза
        BoolString ch2 = bs2;
        for(size_t i = 0; i < points.size(); i += 2){
            if(i != points.size() - 1){
                std::copy(bs1.begin() + points[i], bs1.begin() + points[i +
1], ch2.begin() + points[i]);
            }
            else{ // обмен генами с закольцовыванием от конца в начало
                std::copy(bs1.begin() + points[i], bs1.end(), ch2.begin() +
points[i]);
                std::copy(bs1.begin(), bs1.begin() + points[0], ch2.begin());
            }
        }
        return {ch1, ch2};
    }
}

```

## Файл recombination\_ops.h

```

#ifndef RECOMBINATION_OPS
#define RECOMBINATION_OPS

#include "ga_runner.h"

/* Бинарная рекомбинация (кроссинговер) */

// Количество точек равно ceil(ln(N)), где N - количество бит в двоичных строках
std::vector<BoolString> multi_point_crossingover(BoolString bs1, BoolString
bs2);

#endif

```

## Файл survivor\_selection\_ops.cpp

```

#include "survivor_selection_ops.h"

```



```

#include <random>

static std::random_device dev_urandom;
static std::mt19937 rng(dev_urandom());

struct FitnessBsComp
{
    double (*fitness_func) (BoolString);
    FitnessBsComp(double (*fitness_func) (BoolString)) { this->fitness_func =
fitness_func; }
    bool operator() (BoolString a, BoolString b) { return fitness_func(a) >
fitness_func(b); }
};

double survivor_selection_elite_fraction = 0.2;
double survivor_selection_truncate_threshold = 0.4;
std::vector<BoolString>
elite_truncation_survivor_selection_op(std::vector<BoolString> individuals,
size_t amount,
double (*fitness_func) (BoolString))
{
    std::sort(individuals.begin(), individuals.end(), Fitness-
BsComp(fitness_func));

    size_t elite_amount = ceil(amount * survivor_selection_elite_fraction);
    std::vector<BoolString> new_individuals;
    std::copy(individuals.begin(), individuals.begin() + elite_amount,
std::back_inserter(new_individuals));

    size_t trunc_amount = floor(amount * (1 - survi-
vor_selection_elite_fraction));
    size_t trunc_fraction = ceil(amount * survi-
vor_selection_truncate_threshold);

    std::uniform_int_distribution<size_t> trunc_dist(elite_amount,
elite_amount + trunc_fraction);
    for(size_t i = 0; i < trunc_amount; ++i)
        new_individuals.push_back(individuals[trunc_dist(rng)]);

    return new_individuals;
}

```

## Файл survivor\_selection\_ops.h

```

#ifndef SURVIVOR_SELECTION_OPS_H
#define SURVIVOR_SELECTION_OPS_H

#include "ga_runner.h"

/* Отбор особей в новую популяцию для бинарных строк */

// Метод рулетки
extern double survivor_selection_elite_fraction;
// доля особей, выбираемая элитарным отбором: [0, 1]
extern double survivor_selection_truncate_threshold; //
// доля лучших особей, участвующая в отборе усечением: (0, 1]
std::vector<BoolString>
elite_truncation_survivor_selection_op(std::vector<BoolString> individuals, //
комбинация элитарного отбора и отбора усечением
size_t amount,
double (*fitness_func) (BoolString));

```

```
#endif
```

## Файл perm\_conv.cpp

```
#include "perm_conv.h"
#include <gmpxx.h>
#include <string>
#include <iostream>
#include <cmath>

static double bilog(mpz_t x) {
    signed long int ex;
    const double di = mpz_get_d_2exp(&ex, x);
    return log(di) + log(2) * (double) ex;
}

BoolString perm_to_bool_string(Permutation p) {
    for(size_t i = 0; i < p.size(); i++){
        for(size_t j = i+1; j < p.size(); j++){
            if(p[j] > p[i]) --p[j];
        }
    }

    unsigned fact_mod = 1;
    mpz_class fact = 1, num = 0;
    for(size_t i = p.size()-1; ; i--){
        num += fact*p[i];
        fact *= fact_mod;
        fact_mod++;
        if(i == 0) break;
    }

    std::string s = num.get_str(2);
    BoolString bs;
    size_t i = 0;
    for(; i < s.length(); i++) bs.push_back(s[i] - '0');
    for(; i <= ceil(bilog(fact.get_mpz_t())); i++) bs.insert(bs.begin(), 0);
    return bs;
}

Permutation bool_string_to_perm(BoolString bs, size_t matrix_size) {
    std::string s;
    for(size_t i = 0; i < bs.size(); i++) s.push_back(bs[i] + '0');

    Permutation p;
    mpz_class num;
    num.set_str(s, 2);
    size_t div = 1;
    size_t k = 0;
    for(; num != 0; num /= div, div++, k++){
        mpz_class dig = num % div;
        p.insert(p.begin(), dig.get_ui());
    }

    for(; k < matrix_size; k++) p.insert(p.begin(), 0);

    for(size_t i = p.size()-1; i--){
        for(size_t j = i+1; j < p.size(); j++) if(p[j] >= p[i]) p[j]++;
        if(i == 0) break;
    }
    return p;
}
```

## Файл perm\_conv.h

```
#ifndef PERM_CONV_H
#define PERM_CONV_H

#include "ga_runner/bool_string.h"
#include "permutation.h"

BoolString perm_to_bool_string(Permutation p);
Permutation bool_string_to_perm(BoolString bs, size_t matrix_size);

#endif
```

## Файл perm\_ga.cpp

```
#include "perm_ga.h"
#include <random>

static std::random_device dev_r;
static std::mt19937 rnd(dev_r());

std::vector<BoolString> generate_rand_perms(size_t matrix_size, size_t
pop_size){
    std::vector<BoolString> vbs;
    for(size_t j = 0; j < pop_size; j++){
        Permutation perm;
        std::vector<unsigned> indexes;
        for(size_t i = 0; i < matrix_size; i++){
            indexes.push_back(i);
        }

        for(size_t i = 0; i < matrix_size; i++){
            std::uniform_int_distribution<unsigned> d(0, indexes.size()-1);
            unsigned temp = d(rnd);
            perm.push_back(indexes[temp]);
            indexes.erase(indexes.begin() + temp);
        }
        vbs.push_back(perm_to_bool_string(perm));
    }
    return vbs;
}

double perm_fitness_func(GARunner<BoolString> &gar, BoolString bs){
    std::vector<std::vector<double>> mat =
std::any_cast<std::vector<std::vector<double>>>(gar.get_parameter("cost_matrix",
{}));
    Permutation p = bool_string_to_perm(bs, mat.size());
    if (p.size() != mat.size()){
        return -INFINITY;
    }
    double summ = 0;
    for(size_t i = 0; i < p.size(); i++){
        summ -= mat[i][p[i]];
    }
    return summ;
}
```

## Файл perm\_ga.h

```
#ifndef PERM_GA_H
#define PERM_GA_H

#include "perm_conv.h"
#include "ga_runner/ga_runner.h"

std::vector<BoolString> generate_rand_perms(size_t matrix_size, size_t
pop_size);

double perm_fitness_func(GARunner<BoolString> &gar, BoolString bs);

#endif
```

## Файл permutation.cpp

```
#include "permutation.h"

std::ostream& operator<<(std::ostream& os, Permutation bs)
{
    for(size_t i = 0; i < bs.size(); ++i)
        os << bs[i] << " ";
    return os;
}
```

## Файл permutation.h

```
#ifndef PERMUTATION_H
#define PERMUTATION_H

#include <ostream>
#include <vector>

class Permutation : public std::vector<unsigned>
{
};

std::ostream& operator<<(std::ostream& os, Permutation bs);
```