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

ОТЧЕТ

по учебной практике

Тема: Задача о назначениях

Решоткин А.С.
Докучаев Р.А.
Крицын Д.Р.
Студенты гр. 0304 _____ Козиков А.Е.

Руководитель _____ 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.

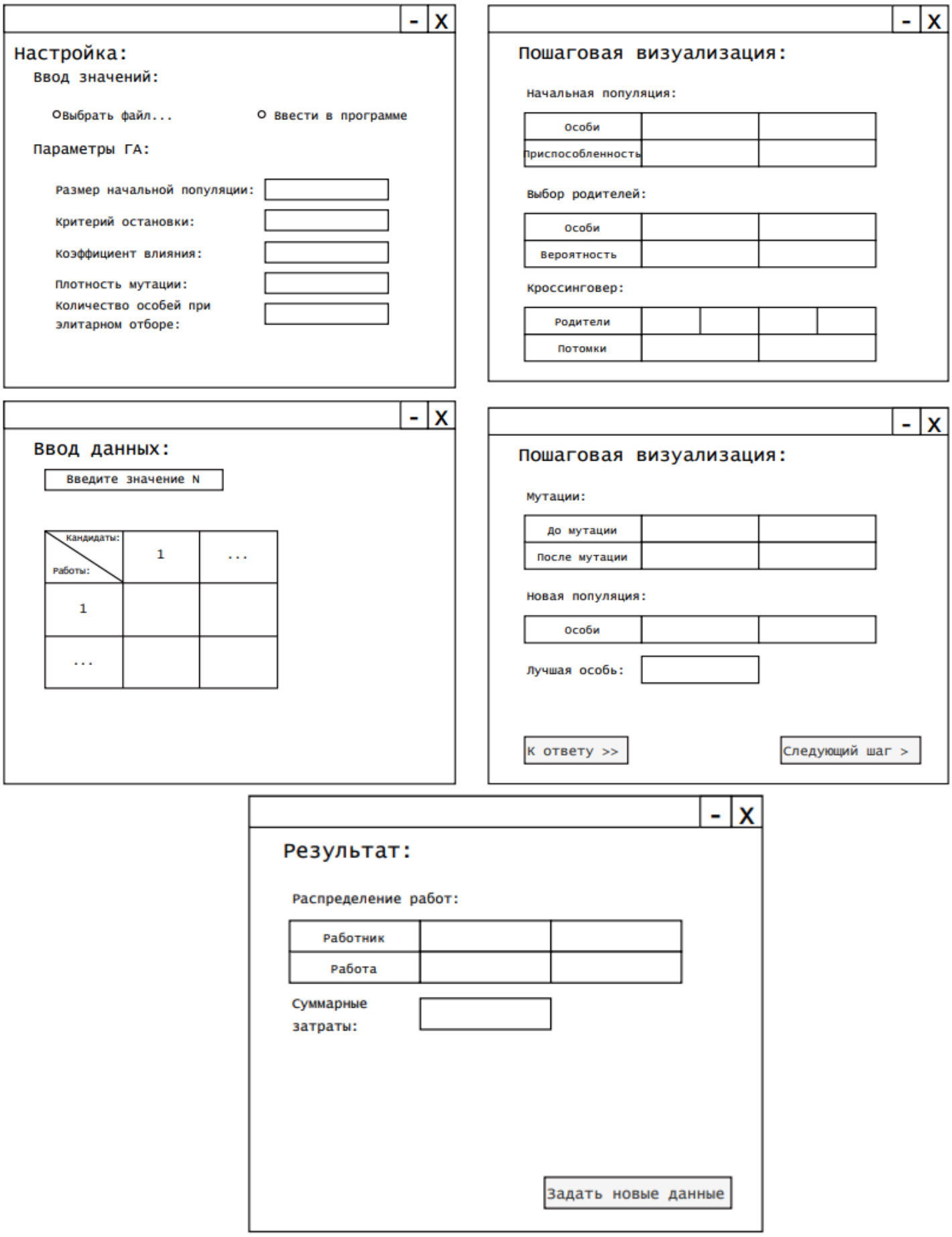
СОДЕРЖАНИЕ

ВВЕДЕНИЕ

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.



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 отсечь 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.

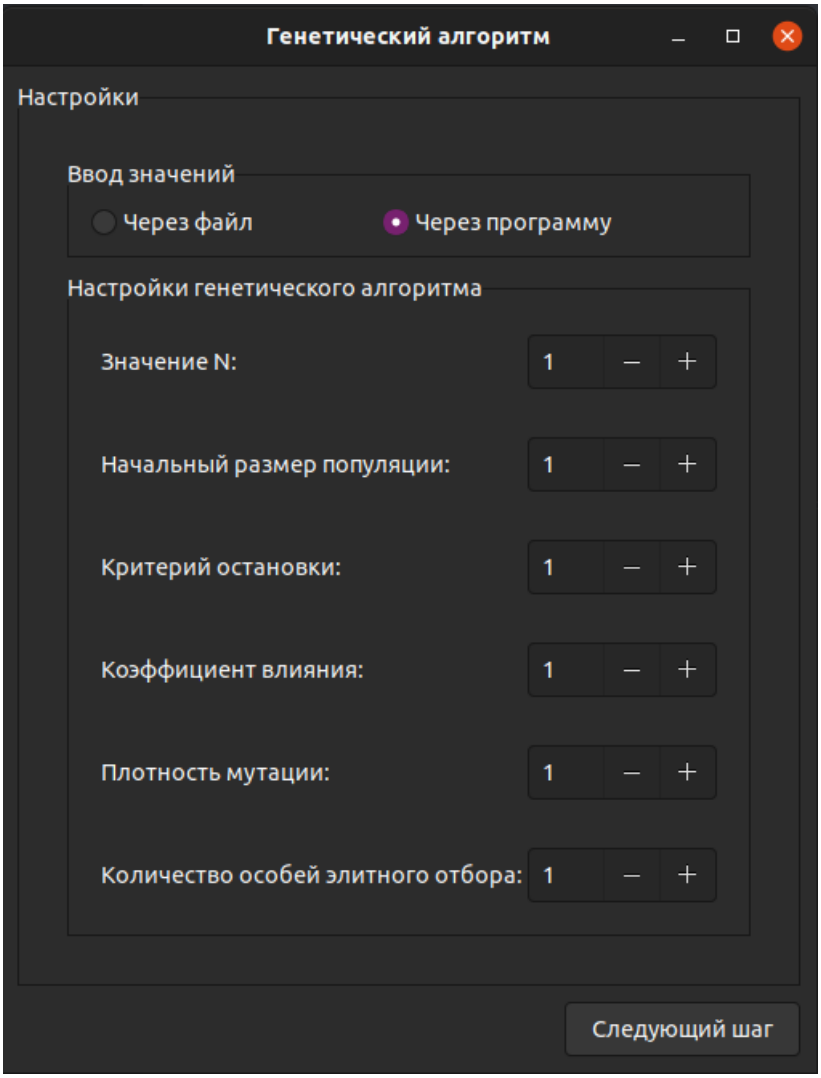


Рисунок 1. Главное окно.

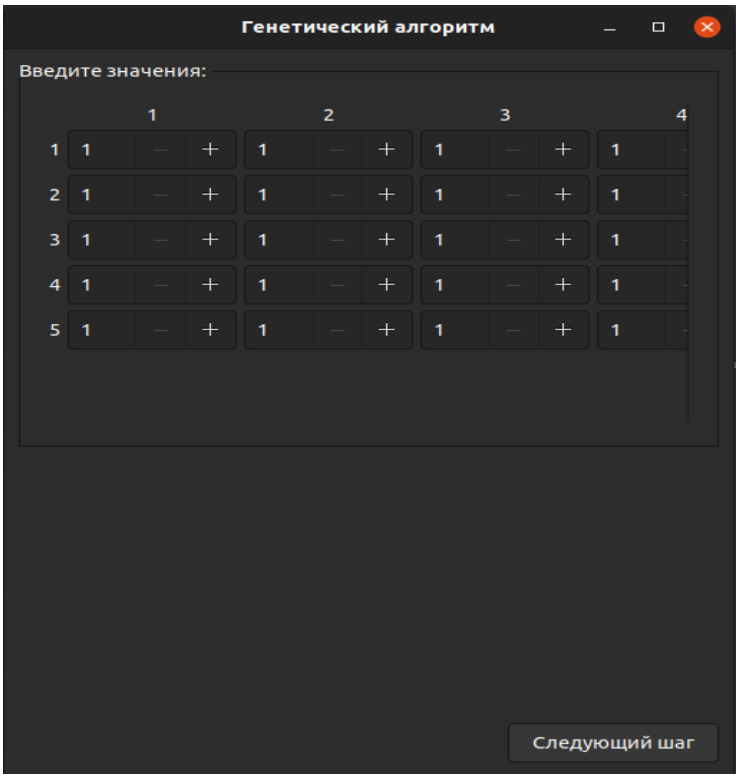


Рисунок 2. Окно ввода данных.

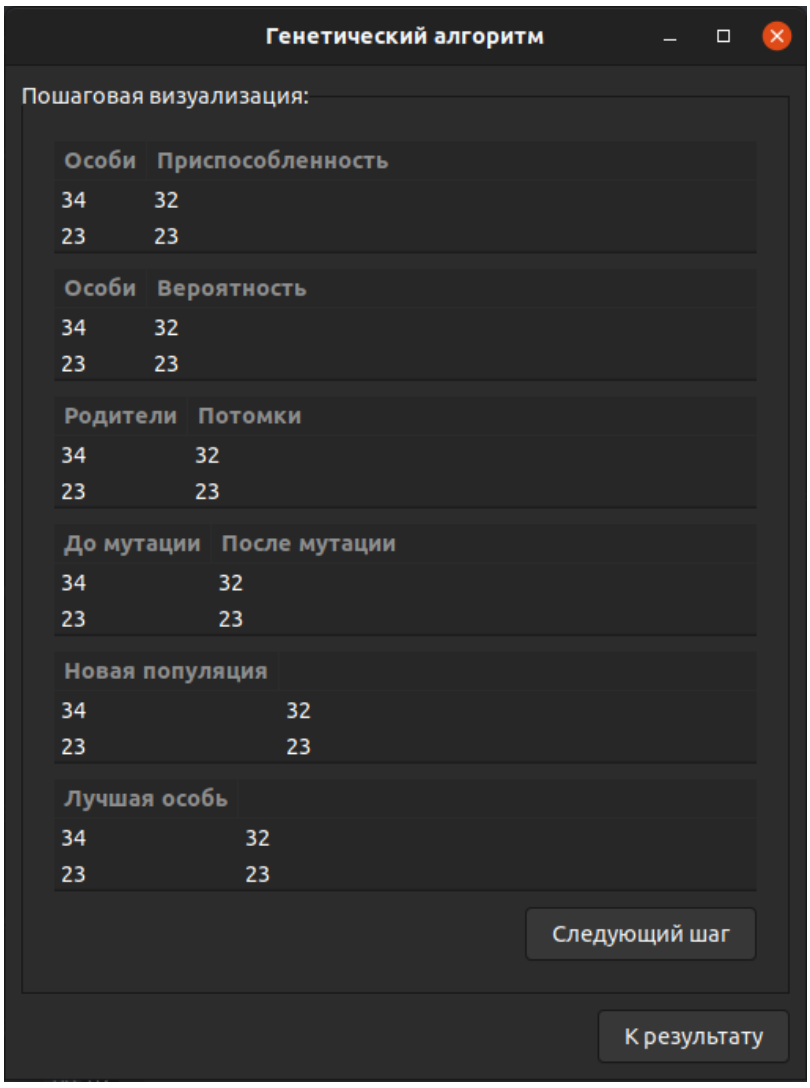


Рисунок 3. Пошаговая визуализация.

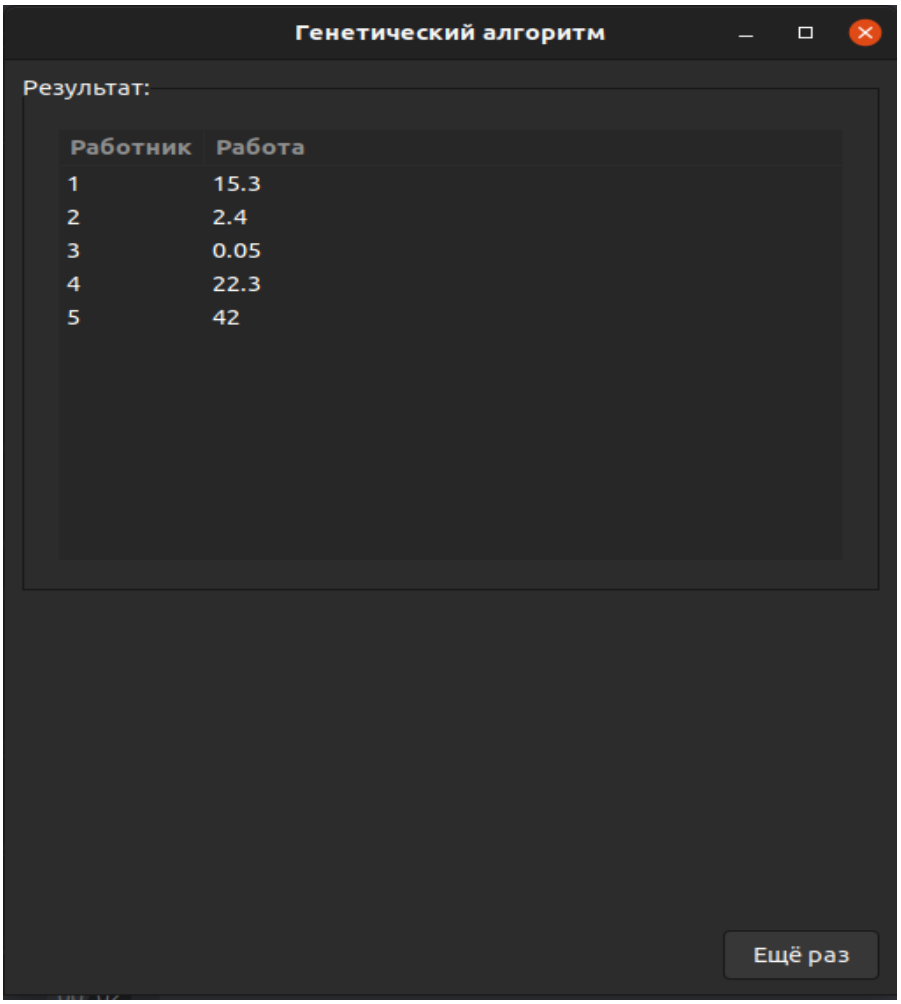


Рисунок 4. Результат работы.

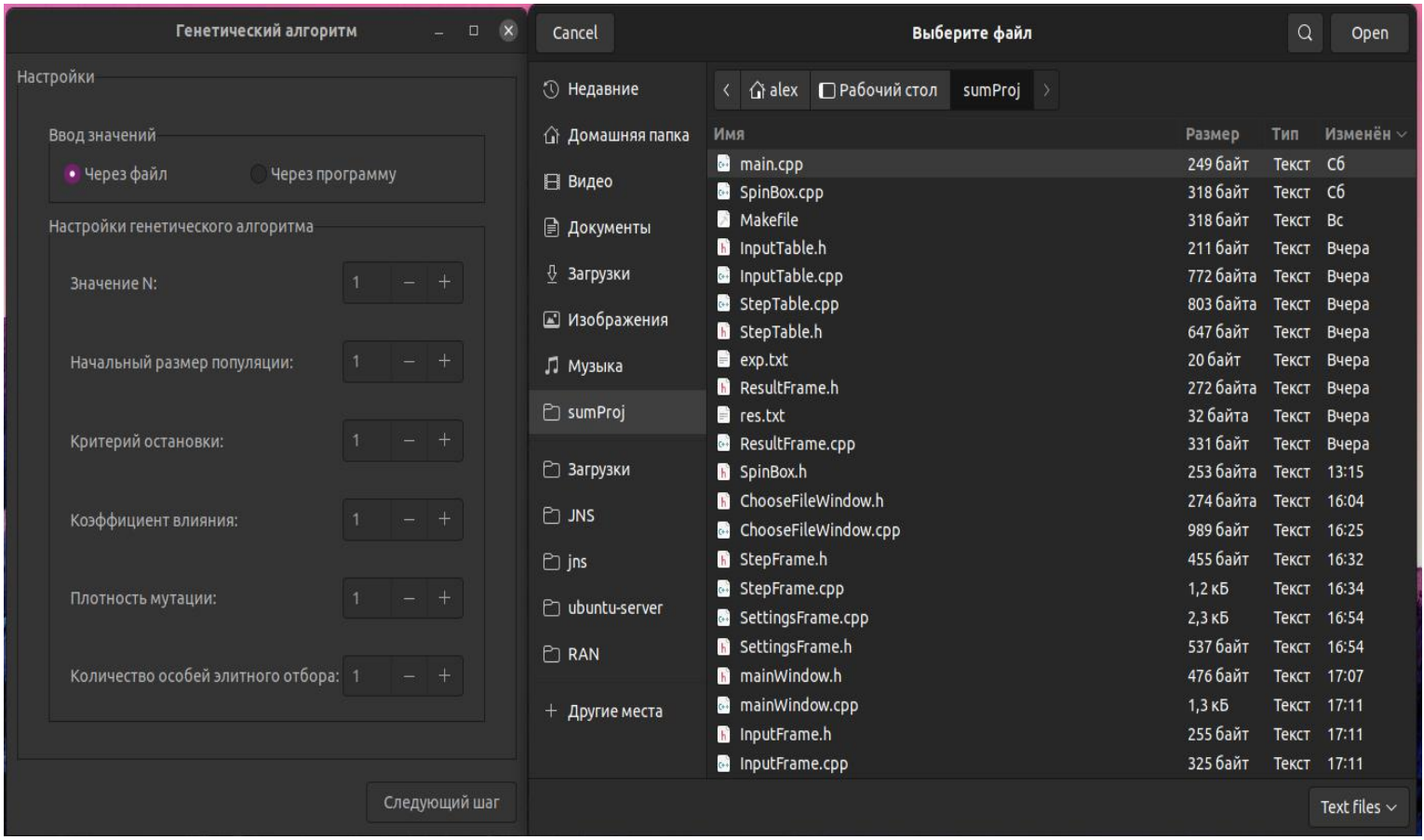


Рисунок 5. Окно для выбора файла.

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 х 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.

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.

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>, dou-
ble (*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 = par-
ent_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 = recombina-
tion_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, GARun-
ner::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> rou-
lette_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) * rou-
lette_fitness_influence + (1. / individuals.size()) * (1 - rou-
lette_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 при-
способленности на шанс хромосомы быть выбранной в методе рулетки: [0, 1]
std::pair<BoolString, BoolString> rou-
lette_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
```