Министерство науки и высшего образования Российской Федерации

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА

АНАЛИЗ ГЕНЕТИЧЕСКОГО АЛГОРИТМА (1 + (ЛЯМБДА, ЛЯМБДА)) НА ЗАДАЧЕ МАКСИМАЛЬНОГО РАЗРЕЗА ГРАФА

Автор: Черноокая Виктория Александровна_____

Направление подготовки: 01.03.02 Прикладная математика и информатика

Квалификация: Бакалавр

Руководитель ВКР: Антипов Д.С., PhD _____

Санкт-Петербург, 2022 г.

Обучающийся Черноокая Виктория Александровна
Группа М3435 Факультет ИТиП

Направленность (профиль), специализация
Информатика и программирование

ВКР принята «_____» _____ 20___ г.

Оригинальность ВКР _____%

ВКР выполнена с оценкой _____

Дата защиты «_____» _____ 20___ г.

Секретарь ГЭК Павлова О.Н. _____

Листов хранения _____

Демонстрационных материалов/Чертежей хранения _____

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В Главе 1 подробно описывается область исследования, которая включает в себя описание алгоритма $(1 + (\lambda, \lambda))$-ГА, задачи поиска максимального разреза графа. Далее, в Главе 2 приводится теоретическая оценка предложенного алгоритма на полных графах, а также сравнение с ожидаемым временем работы других эволюционных алгоритмов. В Главе 3 описываются результаты выполнения известных эволюционных алгоритмов, включая $(1 + (\lambda, \lambda))$-ГА на полных, полных двудольных и случайных графах, а также их сравнение.

ГЛАВА 1. ПРИМЕНЕНИЕ ГЕНЕТИЧЕСКОГО АЛГОРИТМА $(1 + (\lambda, \lambda))$ НА ЗАДАЧЕ ПОИСКА МАКСИМАЛЬНОГО РАЗРЕЗА ГРАФА

В данной главе описывается алгоритм $(1 + (\lambda, \lambda))$ и постановка исследуемой задачи, а именно поиска максимального разреза графа. Также сформулированны условия для получения теоретической и эмпирической оценки применения алгоритма $(1 + (\lambda, \lambda))$ на задаче максимального разреза графа.

1.1. Генетический алгоритм $(1 + (\lambda, \lambda))$

Генетический алгоритм $(1 + (\lambda, \lambda))$ — относительно недавно сформулированный эволюционный алгоритм, минимизирующий функцию приспособленности $f(x) \ : \ \{0, 1\}^n \to \mathbb{R}$ и содержайщий в себе 3 главных параметра:

— размер популяции $\lambda \in \mathbb{N}$;

— вероятность мутации $p \in [0, 1]$;

— смещённость скрещивания $c \in [0, 1]$.

$(1 + (\lambda, \lambda))$-ГА работает с одной родительской особью x, которая инициализируется случайной битовой строкой длины n, где n - размер проблемы. Затем проходят итерации, каждая из которых состоит из двух фаз: мутации и скрещивания(или кроссовера). На этапе мутации алгоритм сначала выбирает силу мутации ℓ из биномиального распределения $\mathrm{Bin}(n, p)$ с n испытаниями и вероятностью успеха p. В алгоритме в фазе мутации $(1 + 1)$-ЭА мы имеем $p = \frac{1}{n}$, но поскольку мы стремимся к более быстрому результату, то обычно рассматривают вероятность p, превышающую $\frac{1}{n}$. После этого создается λ потомков, каждый путем инвертирования ℓ случайных бит родителя. То есть выбираем набор из ell различных позиций в $[n]$ случайным образом и создаем потомка, перевернув битовые значения в этих позициях. Все эти потомки находятся на одинаковом расстоянии от родителя. На промежуточном этапе отбора потомок с наименьшим значением функции приспособленности выбирается победителем мутации для дальнейшего участия в фазе скрещивания. Если таких несколько, то победитель выбирается равномерно случайным образом. Обозначим этого потомка как x'. Далее следует фаза скрещивания. Снова создается λ потомков. На это раз каждый бит потомка берется из победителя мутации с вероятностью c и из родителя с вероятностью $1-c$. Победитель скрещивания y получается тем же способом, как и на фазе мутации. В конце итерации происходит отбор или фаза выбора, где происходит сравнение значений функции

приспособленности родителя x и победителя скрещивания y. Родитель заменяется особью-победителем y фазы скрещивания, если значение $f(y)$ не больше $f(x)$. Псевдокод алгоритма представлен в Листинге 1.

Листинг 1 – Псевдокод $(1 + (\lambda, \lambda))$-ГА, минимизирующего f

$x \leftarrow$ случайная последовательность бит длины n
for $t \leftarrow [1, 2, 3...]$ **do**
  Выбрать ℓ из Bin (n, p) ▷ Фаза мутации
  for $i \in [1..\lambda]$ **do**
    $x^{(i)} \leftarrow$ копия x с инвертированными ℓ битами, взятыми из равномерного распределения
  end for
  $x' \leftarrow \arg\min_{z \in \{x^{(1)}, \ldots, x^{(\lambda)}\}} f(z)$
  for $i \in [1..\lambda]$ **do** ▷ Фаза срещивания
    $y^{(i)} \leftarrow$ каждый бит с вероятностью c берётся из x' иначе из x
  end for
  $y \leftarrow \arg\min_{z \in \{y^{(1)}, \ldots, y^{(\lambda)}\}} f(z)$
  if $f(y) \leqslant f(x)$ **then** ▷ Отбор
    $x \leftarrow y$
  end if
end for

Алгоритм зависит от размера популяции $\lambda \in \mathbb{N}$, и вероятностей мутации и скрещивания $p, c \in [0, 1]$. Без доказательства заметим, что алгоритм не сходится к оптимальному решению, когда $p = 0$ или $c = 0$, или $p = c = 1$. Для всех остальных случаев он в конце концов находит (и сохраняет) оптимум. При $c = 1$ на фазе скрещивания получается победитель мутации, что исключает влияние фазы кроссовера и $(1 + (\lambda, \lambda))$-ГА сводится к $(1 + \lambda)$-ЭА. В частности, для $c = 1, p = \frac{1}{n}\lambda = 1$ алгоритм является уже упомянутым $(1 + 1)$-ЭА. Во всех остальных случаях $0 < c < 1$ результат алгоритма $(1 + (\lambda, \lambda))$-ГА зависит от фазы скрещивания. Поскольку для многих эволюционных алгоритмов, основанных на мутациях, вероятность мутации $\frac{1}{n}$ является рекомендуемым выбором [**ссылка**] (и иногда доказуемо оптимальным [**ссылка**]), то следует использовать алгоритм $(1 + (\lambda, \lambda))$ с p, c, удовлетворяющим $pc = \frac{1}{n}$. Из интуитивных соображений в [**ссылка**] было предложено использовать такое соотношение параметров размера задачи n и размера популяции λ:

— $p = \frac{\lambda}{n}$;
— $c = \frac{1}{\lambda}$.

Также такое соотношение параметров показало свою эффективность и было оптимальным на других анализируемых задачах, таких как OneMax [**ссылка**], LeadingOnes [**ссылка**] и MAX-3SAT [**ссылка**]). В данной работе также будем исспользовать предложенные соотношения параметров.

1.2. Задача о максимальном разрезе графа

Смысл задачи о максимальном разрезе графа - для заданного неориентированного графа без петель и паралельных ребер $G = (V, E)$ с множеством вершин V и ребер E разбить множество вершин на 2 непересекающихся подмножества V_1 и V_2 так, что количество «разрезанных» ребер максимально и $V_1 \cup V_2 = V$. Ребро $(v, u) \in E$ назвается «разрезанным», если инцидентные ему вершины находятся в разных подмножествах V_1 и V_2, то есть $v \in V_1 \cap u \in V_2$ или $v \in V_2 \cap u \in V_1$.

P - разбиение множества V на 2 подмножества V_1 и V_2. Представим его в виде битовой строки, где i-ый бит равен нулю, если $v_i \in V_1$, и единице, если $v_i \in V_2$. Определим функцию Cut от разбиения P :

$$Cut(P) = |\{e = (v_1, v_2) \in E \ : \ v_1 \in V_1 \cap v_2 \in V_2\}|.$$

Неформально говоря, это количество «разрезанных» ребер.

Задача является NP-трудной, что в текущих реалиях означает, что не существует детерминированного алгоритма, решающего задачу за полиномиальное время. Поэтому используются эволюционные алгоритмы. Обычно это нетривиальная задача, так как мы не можем заранее знать сколько ребем мы можем «разрезать». Например, для полного графа K_n с четным n оптимальным разбиение является то, которое разбивает вершины на два подмножества размером $\frac{n}{2}$. В этом случае мы разрезаем $\frac{n^2}{4}$ ребер, что чуть больше половины всех $\frac{n(n-1)}{2}$ ребер.

В рамках этой работы функцию приспособленности определим как

$$f(x) = E - 2Cut(x),$$

где x - разбиение, и будем считать, что разрез найден, если разрезана хотя бы половина ребер. Это позволит нам получить хоть какую-то оценку эволюционных алгоритмов, а также для многих частных случаев является приближенным значением. Ранее уже проводились исследования работы других эволюцион-

ных алгоритмов на данной задаче с таким условием остановки, но для алгоритма $(1 + (\lambda, \lambda))$-ГА результатов ожидаемого времени работы еще получено не было.

Выводы по главе 1

В данной главе был сформулирован алгоритм $(1 + (\lambda, \lambda))$, а также задача используемая для анализа времени работы на нем. Сформулированны и предложены оптимальные параметры для генетического алгоритма, а также затронуто сравнение реализации $(1 + (\lambda, \lambda))$ с другими существующими эволюционными алгоритмами.

ГЛАВА 2. ТЕОРЕТИЧЕСКАЯ ОЦЕНКА ВРЕМЕНИ РАБОТЫ АЛГОРИТМА

2.1. Анализ существующих алгоритмов

2.2. Анализ алгоритма $(1 + (\lambda, \lambda))$ на полных графах

Лемма 1. С вероятностью $1 - o(1)$ хотя бы $\frac{\lambda}{8}$ инвертированных бит в каждой особи - это 0, инвертированные в 1.

Доказательство. Рассмотрим одну итерацию фазы мутации. ℓ - сила мутации, x' - особь победителя, $B' = \{i \in [n] \mid x_i = 0 \cap x'_i = 1\}$ - набор битов, которые стали единицами из нулей. Так как $\lambda = \omega(1)$ и ℓ из Bin $\left(n, \frac{\lambda}{n}\right)$, то простое применение границ Чернова [**ссылка**] говорит, что $|\ell - \lambda| \leqslant \frac{\lambda}{2}$ с вероятностью $1 - o(1)$, то есть $\ell \in [\frac{\lambda}{2}, \frac{3\lambda}{2}]$.

Определим $d = d(x)$ - количество нулей в разбиении x. Проанализируем генерацию одного из λ потомств. $B_1 = \{i \in [n] \mid x_i = 0 \cap x_i^{(1)} = 1\}$. Тогда снова используя границы Чернова, но для гипергеометрического распределения $HG(d, n, \ell)$ [**ссылка**], получим $Pr[|B_1| \geqslant \frac{d\ell}{2n}] = 1 - o(1)$. Так как все потомки имеют одинаковое расстояние Хэмминга от x, а победитель мутации всегда особь с максимальным B_1, то $|B'| \geqslant |B_1| \geqslant \frac{d\ell}{2n} \geqslant \frac{\ell}{4}$. Учитывая ограничения, полученные для ℓ, $Pr[|B'| \geqslant \frac{\lambda}{8}] = 1 - o(1)$.

□

Лемма 2. Вероятность, что из победителя мутации будет взято хотя бы $\frac{c\lambda}{log\lambda}$ бит равна $\Omega(1)$

Доказательство.

Выводы по главе 2

В этой главе была получена верхняя оценка математического ожидания времени работы алгоритма $(1 + (\lambda, \lambda))$ на задаче разреза половины ребер на полных графах $\mathbb{E}[T] = O(nlog(\lambda))$. Также было проведено сравнение теоретического результата ожидаемого времени работы $(1 + (\lambda, \lambda))$-ГА с другими эволюционными алгоритмами на задаче максимального разреза графа. Исходя из полученных данных для этой задачи, можно сделать вывод, что применение этого алгоритма менее эффективно, чем RLS и $(1 + 1)$-ЭА.

ГЛАВА 3. ЭМПИРИЧЕСКАЯ ОЦЕНКА ВРЕМЕНИ РАБОТЫ АЛГОРИТМА ДЛЯ ВСЕХ ТИПОВ ГРАФОВ

В этой главе рассматривается эмпирическая оценка времени работы алгоритма $(1 + (\lambda, \lambda))$ на задаче максимального разреза графа для разных типов графов, описываются конфигурации тестовых запусков. В заключении прилагаются результаты экспериментов в виде графиков, а также анализ полученных данных.

3.1. Конфигурации тестовых запусков

Эксперименты, а также последующее сравнение результатов проводились для следующих алгоритмов:

— $(1 + (\lambda, \lambda))$-ГА;
— $(1 + 1)$-ЭА со стандартным операттором мутации;
— $(1 + 1)$-ЭА с выбором вероятности по степенному закону;
— Random Local Search.

Алгоритмы запускались и начинали свою работу на графах, у которых разбиение было представлено битовой строкой только из нулей, то есть на первой итерации всех алгоритмов количество «разрезанных» ребер было равно нулю и все вершины находились в подмножестве V_1, согласно определению разбиения. Алгоритмы останавливали свою работу когда функция приспособленности достигала значения меньше либо равное нулю. То есть хотя бы половина ребер разрезана. Эксперименты проводились на различных типах графов без петель и параллельных ребер:

— полные графы;
— полные двудольные графы;
— случайно сгенерированные.

Также тестовые запуски проводились на графах с различным количеством вершин, равным степеням двойки: 2^5, 2^6, 2^7, 2^8, 2^9, 2^{10}, 2^{11}. В двудольных графах количество ребер в долях было одинаковым.

Для каждой конфигурации алгоритмы запускались по 80 раз для более точного анализа ожидаемого времени работы. Время работы считалось, как количество вычислений функции приспособленности. Конечным результатом времени работы алгоритма для каждого типа графа считалось среднее арифметическое результатов времени работы всех 80 запусков. Для отображения полученных значений на графиках таже учитывались отклонения от среднего.

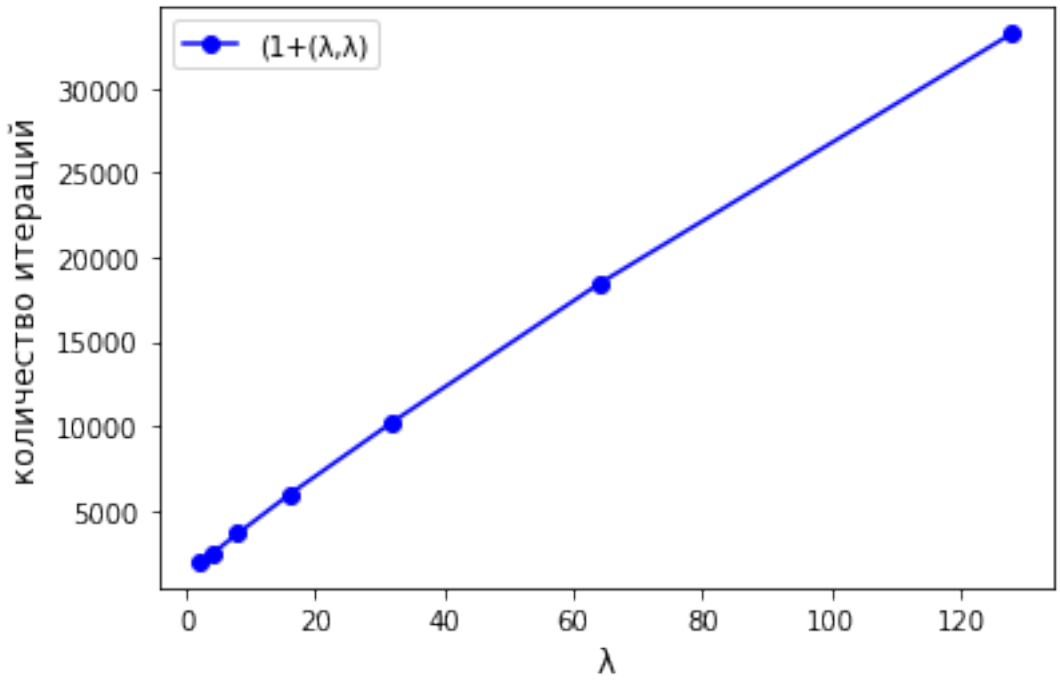

Рисунок 1 – Производительность $(1 + (\lambda, \lambda))$-ГА для различных значений λ на полных графах с количеством вершин 2^{10}

Для алгоритма $(1 + 1)$-ЭА с выбором вероятности по степенному закону константный параметр β равен 1,5. Для $(1 + (\lambda, \lambda))$ используем константное значение $\lambda = 10$. Решение об использовании именно этой константы было принято на основании дополнительных экспериментов проведенных на полных графов с количеством вершин - 1024. Для исследования рассматривались λ равные 2, 4, 8, 16, 32, 64, 128. Для каждого значения λ алгоритм запускался также по 80 раз.

На Рисунке 1 изображен график зависимости числа вычислений функции приспособленности от значий λ. Исходя из графика можно сделать вывод, что брать большую λ не оптимально, но и в случае слишком маленького значения, например равной 1, алгоритм не будет отличаться от $(1 + 1)$-ЭА. Тогда, чтоб была возможность оценить эффективность данного генетического алгоритма выберем $\lambda = 10$. Для всех запусков $(1 + (\lambda, \lambda))$-ГА использовалась именно эта константа.

Для каждого запуска на каждой итерации собиралась и записывалась информация:

— текущее значение функции приспособленности f;
— значение f для победителя мутации;
— значение f для победителя кроссовера (для $(1 + (\lambda, \lambda))$-ГА).

Такой сбор информации позволяет более детально увидеть и проанализировать работу алгоритма на практике. Также после завершения каждого алгоритма фиксировалось количество вычислений функции приспособленности.

3.2. Результаты экспериментов

Выводы по главе 3

В этой главе были проанализированы результаты запусков эволюционных алгоритмов на задаче поиска максимального разреза графа для различных типов графов. Полученная эмпирическая оценка соответствует ожиданиям, полученным во 2 главе. Также на основании полученных результатов проведенных экспериментов можно предполагать, что ожидаемое время работы генетического алгоритма $(1 + (\lambda, \lambda))$ для разных типов графов такое же как и для полных и равно $O(nlog(\lambda))$.

ЗАКЛЮЧЕНИЕ

В данном разделе размещается заключение.

ПРИЛОЖЕНИЕ А. ИСХОДНЫЙ КОД

Листинг А.1 – Реализация $(1 + (\lambda, \lambda))$-ГА со сбором статистики

```
#include <iostream>
#include <fstream>
#include <ctime>
#include <stdlib.h>
#include <valarray>
#include "graph.h"

using namespace std;

enum TypeAlgo {
    RLS,
    ONEPLUSONE,
    LAMBDA,
    ONEPLUSONE_POWERLAW
};

const int REPEAT = 80;
const double BETTA = 1.5; // for power law

vector<double> powerLawDistribution (int n) {
    vector<double> distribution (n, 0.0);
    double sumDist = 0.0;

    for (int i = 1; i <= n; i++) {
        double cur = pow(i, -BETTA);
        sumDist += cur;
    }

    distribution[0] = 1.0 / sumDist;

    for (int i = 1; i < n; i++) {
        distribution[i] = pow(i + 1 , -BETTA) / sumDist +
            distribution[i-1];
    }

    return distribution;
}
```

```
void run (TypeGraph typeGraph, int size, vector<TypeAlgo>
    typesAlgo) {
    vector<vector<int>> result = vector<vector<int>> (typesAlgo.
        size(), vector<int>(REPEAT, 0));
    vector<int> sum = vector<int>(typesAlgo.size(), 0);
    vector<int> max = vector<int>(typesAlgo.size(), 0);
    vector<int> min = vector<int>(typesAlgo.size(), 100000000);

    for (int i = 0; i < REPEAT; i++) {
        auto *graph = new Graph(size, typeGraph);

        for (int j = 0; j < typesAlgo.size(); j++) {
            if (typesAlgo[j] == RLS) result[j][i] = graph->RLS(1);
            else if (typesAlgo[j] == ONEPLUSONE) result[j][i] =
                graph->onePlusOneAlgorithm(1);
            else if (typesAlgo[j] == ONEPLUSONE_POWERLAW) {
                vector<double> dist = powerLawDistribution(size);
                result[j][i] = graph->
                    onePlusOneHeavyTailedAlgorithm(1, dist);
            }
            else if (typesAlgo[j] == LAMBDA) result[j][i] = graph
                ->lambdaAlgorithm(1);

            graph->reset();

            sum[j] += result[j][i];
            if (result[j][i] > max[j]) max[j] = result[j][i];
            if (result[j][i] < min[j]) min[j] = result[j][i];
        }

    }

    vector<double> average = vector<double>(typesAlgo.size(), 0.0)
        ;

    for (int j = 0; j < typesAlgo.size(); j++) {
        average[j] = (double) sum[j] / REPEAT;
    }

    ofstream out("out", ios_base::app);
    out << "результаты:\n";
```

```
        out << "размер графа - " << size << '\n';
        out << "тип графа - ";

        if (typeGraph == KN) {
            out << "KN\n";
        } else if (typeGraph == KNN) {
            out << "KNN\n";
        } else if (typeGraph == RANDOM) {
            out << "Random\n";
        }

        for (int i = 0; i < typesAlgo.size(); i++) {
            out << "алгоритм - ";

            if (typesAlgo[i] == RLS) {
                out << "RLS\n";
            } else if (typesAlgo[i] == ONEPLUSONE) {
                out << "(1 + 1)\n";
            } else if (typesAlgo[i] == ONEPLUSONE_POWERLAW) {
            out << "(1 + 1) Heavy-tailed\n betta = " << BETTA << "\n";
            } else if (typesAlgo[i] == LAMBDA) {
                out << "(1 + лямбда(, лямбда))\n";
            }

            out << "время работы: ";

            for (int j = 0; j < REPEAT; j++) {
                out << result[i][j] << ' ';
            }
            out << '\n';
            out << "границы - [" << min[i] << ", " << max[i] << "]\n";

            out << "среднее времяработы  - " << average[i] << '\n';
            out << "нормированное поразмерузадачи   - " << (double) average
               [i] / size << '\n';

        }
        out << "-------------------------\n";

}
```

```
int main() {
    srand(static_cast<unsigned int>(time(0)));
    ofstream out;
    out.open("out");
    out << "";

    vector<int> sizeGraph {1024}; //{32, 64, 128, 256, 512, 1024,
       2048}
    vector<TypeAlgo> typesAlgo = { LAMBDA }; //{RLS, ONEPLUSONE,
       ONEPLUSONE_POWERLAW, LAMBDA}
    vector<TypeGraph> typesGraph = { KN };  //{KNN, KN, RANDOM}


    for (int i = 0; i < typesGraph.size(); i++) {
            for (int j = 0; j < sizeGraph.size(); j++) {
                run(typesGraph[i], sizeGraph[j], typesAlgo);
            }
    }
    return 0;
}

enum TypeGraph {
    KN,
    KNN,
    RANDOM
};

class Graph {
private:
    vector<int> d;
    long long D;
    vector<int> e;
public:
    int n;
    long long numberEdges;
    vector< vector<int> > matrix;
    TypeGraph type;

    Graph(int size, TypeGraph t);

    void setVecD(vector<int> nd);
```

```
    void setD(long long nD);
    void setE(vector<int> ne);

    vector<int> getVecD();
    long long getD();
    vector<int> getE();

    vector<int> countd(vector<int> e);
    long long countD(vector<int> e);
    void createKn();
    void createKnn();
    void createRandom();

    void printMatrix();
    void printVec(vector<int> vec);

    int RLS(int iteration);
    int lambdaAlgorithm(int iteration);
    int onePlusOneAlgorithm(int iteration);
    int onePlusOneHeavyTailedAlgorithm(int iteration, vector<
       double> dist);

    void reset();
};

const int LAMBDA = 128; //1 2 4 8 16 32 64 128

Graph::Graph(int size, TypeGraph t) {
    type = t;
    n = size;
    numberEdges = 0;
    e = vector<int> (size, 0);
    d = vector<int> (size, 0);

    if (type == KN) {
        createKn();
    } else if (type == KNN) {
        createKnn();
    } else if (type == RANDOM) {
        createRandom();
    }
```

```
};

void Graph::reset() {
    vector<int> ne = vector<int> (n, 0);
    setE(ne);
    setVecD(countd(getE()));
    setD(countD(getE()));
};

void Graph::setVecD(vector<int> nd) {
    d = move(nd);
};
void Graph::setD(long long nD) {
    D = nD;
};
void Graph::setE(vector<int> ne) {
    e = move(ne);
};

vector<int> Graph::getVecD() {
    return d;
};
long long Graph::getD(){
    return D;
};
vector<int> Graph::getE(){
    return e;
};

void Graph::printVec(vector<int> vec) {
    ofstream out("out", ios_base::app);
    for (int i : vec) {
        out << i << " ";
    }
    out << '\n';
    out.close();
}

void Graph::printMatrix() {
    ofstream out("out", ios_base::app);
    out << "количество вершин " << matrix.size() << '\n';
```

```
    out << "матрица смежности matrix:" << '\n';
    for (auto & i : matrix) {
        for (int j = 0; j < matrix.size(); j++) {
            out << i[j] << " ";
        }
        out << '\n';
    }
    //out << "--------------------------" << '\n';
    out << "количество ребер: " << numberEdges << '\n';
    out << "--------------------------" << '\n';
    out.close();
}

vector<int> Graph::countd (vector<int> newE) {
    vector<int> ans(n, 0);
    for(int i = 0; i < matrix.size(); i++) {
        for(int j = 0; j < matrix.size(); j++) {
            if(matrix[i][j] == 1) {
                if(newE[i] == newE[j]) {
                    ans[i]++;
                    ans[j]++;
                } else {
                    ans[i]--;
                    ans[j]--;
                }
            }
        }
    }
    for (int & i : ans) {
        i /= 2;
    }
    return ans;
}

long long Graph::countD (vector<int> newE) {
    vector<int> newd;
    long long ans = 0;
    newd = countd(newE);

    for (int i : newd) {
        ans += i;
```

```
    }
    return ans;
}

void Graph::createKn () {
    ofstream out("out", ios_base::app);
    out << "KN graph\n";
    matrix = vector< vector <int> >(n, vector<int> (n, 0));

    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (i != j) {
                matrix[i][j] = 1;
                numberEdges++;
            }
        }
    }
    numberEdges /= 2;
    setVecD(countd(getE()));
    setD(countD(getE()));

    printMatrix();
};

void Graph::createKnn () {
    ofstream out("out", ios_base::app);
    out << "KNN graph\n";
    matrix = vector< vector <int> >(n, vector<int> (n, 0));

    for (int i = 0; i < n/2; i++) {
        for (int j = n/2; j < n; j++) {
            matrix[i][j] = 1;
            matrix[j][i] = 1;
            numberEdges++;
        }
    }

    setVecD(countd(getE()));
    setD(countD(getE()));
    printMatrix();
};
```

```
void Graph::createRandom () {
    ofstream out("out", ios_base::app);
    out << "RANDOM graph\n";
    matrix = vector< vector <int> >(n, vector<int> (n, 0));

    for (int i = 0; i < n; i++) {
        for (int j = i+1; j < n; j++) {
                int randEdge = rand() % 2;
                matrix[i][j] = randEdge;
                matrix[j][i] = randEdge;
                if (randEdge == 1) numberEdges++;
        }
    }

    setVecD(countd(getE()));
    setD(countD(getE()));
    printMatrix();
}



int binSearch (double x, vector<double> dist) {
    ofstream out("out", ios_base::app);
    //out << "X    " << x <<'\n';
    int l = 0;
    int r = dist.size();

    while (r > l) {
        int m = (l + r) / 2;

        if (dist[m] - x < 1e-12) {
            l = m + 1;
        } else if (dist[m] - x >= 1e-12) {
            r = m - 1;
        } else {
            return m;
        }
    }
    if (l == dist.size()) return l;
```

```
    return 1 + 1;
}

int Graph::RLS (int iteration) {
    ofstream out("out", ios_base::app);
    out << "--------------------------\n";
    //out << "iteration " << iteration << '\n';
    int pos = rand() % this->n;

    out << "flip " << pos << '\n';
    /*out << "prev e: ";
    for (int i : getE()) {
        out << i << " ";
    }
    out << '\n';*/

    vector<int> newE = getE();
    newE[pos] = (newE[pos] + 1) % 2;

    //out << "new e:  ";
    /*for (int i : newE) {
        out << i << " ";
    }
    out << '\n';*/

    long long newD = countD(newE);

    vector<int> newd = countd(newE);

    out << "старый потенциал = " << getD() << "\послеn флипа = " <<
       newD << '\n';

    if (newD <= getD()) {
        setE(newE);
        setVecD(countd(newE));
        setD(countD(newE));
        out << "good mutation\n";
    }

    if (getD() > 0) {
        iteration++;
```

```
        out.close();
        return RLS(iteration);
    } else {
        out << "----------------------------\n";
        out << "(" << this->n << ", ";
        if (this->type == KN) {
            out << "KN";
        } else if (this->type == KNN) {
            out << "KNN";
        } else {
            out << "Random";
        }
        out << ")\n";
        out << "разрезана половинареберза    " << iteration << "
           итераций\n";
        out << "----------------------------\n";
        return iteration;
    }
}

int Graph::onePlusOneAlgorithm(int iteration){
    ofstream out("out", ios_base::app);
    out << "----------------------------\n";
    //out << "iteration " << iteration << '\n';
    //int l = rand() % this->n;
    unsigned seed = chrono::system_clock::now().time_since_epoch()
       .count();
    default_random_engine generator(seed);
    binomial_distribution<int> distribution(this->n,(double) 1/
       this->n);

    int l = distribution(generator);

    out << "flip " << l << " bits\n";
    vector<int> newE = getE();

    vector<int> flipArr(this->n, 0);

    for (int i = 0; i < l; i++) {
        int index;
```

```
        while(true) {
            index = rand() % this->n;

            if (flipArr[index] != 1) {
                flipArr[index] = 1;
                newE[index] = (newE[index] + 1) % 2;
                break;
            }
        }
    }

    long long newD = countD(newE);

    out << "старый потенциал = " << getD() << "\послеn флипа = " <<
       newD << '\n';

    if (newD <= getD()) {
        setE(newE);
        setVecD(countd(newE));
        setD(countD(newE));
        out << "good mutation\n";
    }

    if (getD() > 0) {
        iteration++;
        out.close();
        return onePlusOneAlgorithm(iteration);
    } else {
        out << "--------------------------\n";
        out << "(" << this->n << ", " ;
        if (this->type == KN) {
            out << "KN";
        } else if (this->type == KNN) {
            out << "KNN";
        } else {
            out << "Random";
        }
        out << ")\n";
        out << "разрезана половинареберза    " << iteration << "
           итераций\n";
        out << "--------------------------\n";
```

```
            return iteration;
        }
    }

    int Graph::onePlusOneHeavyTailedAlgorithm(int iteration, vector<
        double> dist) {
        ofstream out("out", ios_base::app);
        out << "--------------------------\n";

        unsigned seed = chrono::system_clock::now().time_since_epoch()
            .count();
        default_random_engine generator(seed);
        uniform_real_distribution<> distribution(0.0, 1.0);

        double y =  distribution(generator);
        int l = binSearch(y, dist);

        out << "flip " << l << " bits\n";
        vector<int> newE = getE();

        vector<int> flipArr(this->n, 0);

        for (int i = 0; i < l; i++) {
            int index;

            while(true) {
                index = rand() % this->n;

                if (flipArr[index] != 1) {
                    flipArr[index] = 1;
                    newE[index] = (newE[index] + 1) % 2;
                    break;
                }
            }
        }

        long long newD = countD(newE);

        out << "старый потенциал = " << getD() << "\послеn флипа = " <<
            newD << '\n';
```

```
    if (newD <= getD()) {
        setE(newE);
        setVecD(countd(newE));
        setD(countD(newE));
        out << "good mutation\n";
    }

    if (getD() > 0) {
        iteration++;
        out.close();
        return onePlusOneHeavyTailedAlgorithm(iteration, dist);
    } else {
        out << "--------------------------\n";
        out << "(" << this->n << ", " ;
        if (this->type == KN) {
            out << "KN";
        } else if (this->type == KNN) {
            out << "KNN";
        } else {
            out << "Random";
        }
        out << ")\n";
        out << "разрезана половинареберза    " << iteration << "
           итераций\n";
        out << "--------------------------\n";
        return iteration;
    }
}



int Graph::lambdaAlgorithm(int iteration) {
    ofstream out("out", ios_base::app);
    out << "--------------------------\n";
    out << "iteration " << iteration << '\n';

    unsigned seed = chrono::system_clock::now().time_since_epoch()
       .count();
    default_random_engine generator(seed);
    binomial_distribution<int> distribution(this->n,(double)
      LAMBDA/this->n);
```

```
int l = distribution(generator);
vector<int> x = getE();
out << "flip " << l << " bits\n";

if (l == 0) {
    iteration++;
    out.close();
    return lambdaAlgorithm(iteration);
}

vector<vector<int>> mutationArr(LAMBDA, x);

for (int i = 0; i < LAMBDA; i++) {
    vector<int> flipArr(this->n, 0);

    for (int j = 0; j < l; j++) {
        int index;

        while(true) {
            index = rand() % this->n;

            if (flipArr[index] != 1) {
                flipArr[index] = 1;
                mutationArr[i][index] = (mutationArr[i][index]
                    + 1) % 2;
                break;
            }
        }
    }
}

int mutationWinIndex = 0;
int bestMutD = countD(mutationArr[mutationWinIndex]);

for (int i = 1; i < LAMBDA; i++) {
    int curD = countD(mutationArr[i]);

    if (curD < bestMutD) {
        mutationWinIndex = i;
        bestMutD = curD;
```

```
    }
}

out << "победитель мутации:   ";
for (int i : mutationArr[mutationWinIndex]) {
    out << i << " ";
}
out << '\n';

out << "старый потенциал = " << getD() << "\nослеn флипа = " <<
   bestMutD << '\n';

vector<vector<int>> y(LAMBDA, x);

for (int i = 0; i < LAMBDA; i++) {
    for (int j = 0; j < y[i].size(); j++) {
            if ((double)rand() / (double)RAND_MAX < (double)1/
               LAMBDA) {
                y[i][j] = mutationArr[mutationWinIndex][j];
            }
    }
}

int crossoverWinIndex = 0;
int bestCrossD = countD(y[crossoverWinIndex]);

for (int i = 1; i < LAMBDA; i++) {
    int curD = countD(y[i]);

    if (curD < bestCrossD) {
        crossoverWinIndex = i;
        bestCrossD = curD;
    }
}

out << "победитель кроссовера:   ";
for (int i : y[crossoverWinIndex]) {
    out << i << " ";
}
out << '\n';
```

```
        out << "старый потенциал = " << getD() << "\nослен кроссовера = "
            << bestCrossD << '\n';

        if (getD() > bestCrossD) {
            setE(y[crossoverWinIndex]);
            setVecD(countd(y[crossoverWinIndex]));
            setD(bestCrossD);
            out << "good mutation\n";
        }

        if (getD() > 0) {
            iteration++;
            out.close();
            return lambdaAlgorithm(iteration);
        } else {
            out << "--------------------------\n";
            out << "(" << this->n << ", ";
            if (this->type == KN) {
                out << "KN";
            } else if (this->type == KNN) {
                out << "KNN";
            } else {
                out << "Random";
            }
            out << ")\n";
            out << "разрезана половинареберза    " << iteration*2*LAMBDA <<
                " итераций\n";
            out << "LAMBDA " << LAMBDA << "\n";
            out << "--------------------------\n";
            return iteration*2*LAMBDA;
        }
    }
```