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

Эволюционные вычисления являются достаточно общим термином, который содержит в себе множество похожих между собой техник. Принцип эволюционного программирования состоит в имитации естественного отбора [1]. Эволюционные алгоритмы — такие алгоритмы оптимизации, в основном использующиеся для решения сложных задач (NP - трудных) на практике. Для этих задач не существует детерминированного алгоритма, решающего задачу за полиномиальное время. В текущих реалиях, если только не $P = NP$, лучшее решение найти не представляется возможным, поэтому эволюционные алгоритмы находят достаточно хорошее решение за приемлемое время работы. За счет своей эффективности, доказанной экспериментально, они часто применяются на практике. Но, к сожалению, на данный момент мы не обладаем достаточной информацией с теоретической точки зрения о их рабочих принципах. Этот факт мешает подбирать для каждой конкретной задачи самый оптимальный алгоритм.

Цель данной работы — улучшить понимание рабочих принципов эволюционных алгоритмов на задачах с графами. Для достижения этой цели мы исследуем то, как различные эволюционные алгоритмы решают задачу о поиске максимального разреза графа, а точнее, находят приближение к нему.

Так как для произвольного графа неизвестно заранее, какое максимальное число ребер могут быть «разрезаны», то будем оценивать время работы эволюционных алгоритмов с условием, что должна быть «резрезана» хотя бы половина ребер. Для разных графов такое решение может быть по-разному близко к оптимальному. Например, для полных графов (для которых в данной работе будет приведена теоретическая оценка) мы можем разрезать максимум $\lfloor \frac{1}{2}|E|(1 + \frac{1}{n-1})\rfloor$. Исходя из этого можно сказать, что решение с половиной «разрезанных» ребер - это достаточно хорошая аппроксимация оптимального решения с точностью до множителя $1 + o(1)$. Для двудольного же графа мы можем разрезать все ребра, если положим доли по разные стороны от разреза. То есть для таких типов графов при разрезе половины ребер мы получаем решение, которое в 2 раза хуже оптимального.

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 предложен-

ного алгоритма на полных графах, а также сравнение с ожидаемым временем работы других эволюционных алгоритмов:

— Random Local Search (RLS);

— $(1+1)$-ЭА со стандартным оператором мутации;

— $(1+1)$-ЭА с выбором вероятности по степенному закону.

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 разработанный 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 обычно инициализируется случайной битовой строкой длины n, где n - размер задачи. В нашем случае особь инициализируется строкой из нулей. Затем проходят итерации, каждая из которых состоит из двух фаз: мутации и скрещивания.

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 $(1 + 1)$-ЭА в фазе мутации 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 ℓ различных позиций в $[n]$ случайным образом и создаем потомка, инвертировав битовые значения в этих позициях. Все эти потомки находятся на одинаковом расстоянии от родителя. На промежуточном этапе отбора потомок с наилучшим значением функции приспособленности выбирается победителем мутации для дальнейшего участия в фазе скрещивания. Если таких несколько, то победитель выбирается равномерно случайным образом среди претендентов. Обозначим этого потомка как x'.

Далее следует фаза скрещивания. Снова создается λ потомков. На этот раз каждый бит потомка берется из победителя мутации с вероятностью c и из родителя с вероятностью $1 - c$. Победитель скрещивания y выбирается 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 хуже $f(x)$. Псевдокод алгоритма представлен в Листинге 1.

Листинг 1 – Псевдокод $(1 + (\lambda, \lambda))$-ГА, максимизирующего 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
  $x' \leftarrow \arg\max_{z \in \{x^{(1)}, ..., x^{(\lambda)}\}} f(z)$
  for $i \in [1..\lambda]$ **do** ▷ Фаза срещивания
    $y^{(i)} \leftarrow$ каждый бит с вероятностью c берётся из x' иначе из x
  end for
  $y \leftarrow \arg\max_{z \in \{y^{(1)}, ..., y^{(\lambda)}\}} f(z)$
  if $f(y) \geqslant f(x)$ **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 скрещивания и $(1 + (\lambda, \lambda))$-ГА сводится к $(1 + \lambda)$-ЭА. В частности, для $c = 1, p = \frac{1}{n}$ и $\lambda = 1$ алгоритм является $(1 + 1)$-ЭА. Во всех остальных случаях $0 < c < 1$ результат итерации 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 используются 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 два непересекающихся подмножества V_1 и V_2 так, что число «разрезанных» ребер было 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 два 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 число «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. Следовательно, эволюционный алгоритм не может заранее знать насколько он близок к оптимальному решению, и тогда следует остановить работу, 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 эволюционных алгоритмов на данной задаче с таким условием остановки, однако нет опублико-

ванных результатов, 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 $(1 + (\lambda, \lambda))$-ГА с другими существующими эволюционными алгоритмами.

ГЛАВА 2. ТЕОРЕТИЧЕСКАЯ ОЦЕНКА ВРЕМЕНИ РАБОТЫ АЛГОРИТМА НА ПОЛНЫХ ГРАФОВ

Теоретических оценок для задачи максимального разреза графа на данный момент не существует, поэтому целесообразно начать исследования в этой области с одного класса графов. В данной главе проводится теоретическая оценка эволюционных алгоритмов RLS, $(1+1)$-ЭА со стандартным оператором мутации и с оператором мутации с выбором вероятности по степенному закону, а также генетического алгоритма $(1+(\lambda,\lambda))$ на задаче поиска максимального разреза графа для полных графов. Для этого типа графа, учитывая описанный ранее критерий остановки, алгоритм выдаст результат, очень близкий к оптимальному.

2.1. Анализ эволюционных алгоритмов, основанных на мутации

Необходимо оценить ожидаемое время, то есть число вычислений функции приспособленности, за которое разные эволюционные алгоритмы найдут разрез, в котором хотя бы половина общего числа ребер «разрезаны».

Всего ребер в полном графе $\frac{n(n-1)}{2}$, где n — число вершин в графе. Соответственно необходимо найти такое разбиение, что разрезаных ребер $\lceil \frac{n(n-1)}{4} \rceil$. Обозначим m, как число вершин по правую сторону от разреза, то есть исходя из введенных определений — это число единиц в разбиении P. Тогда число «разрезанных» ребер в такой ситуации $m(n-m)$.

Рассмотрим два случая, когда число ребер в полном графе четное и нечетное соответственно.

Пусть $\frac{n(n-1)}{2}$ четное, то есть $\lceil \frac{n(n-1)}{4} \rceil = \frac{n(n-1)}{4}$. Тогда оценим необходимое значение m :

$$m(n-m) \geqslant \frac{n(n-1)}{4}.$$

Используя методы решения квадратных неравенств получаем промежуток значений:

$$m \in \left[\frac{n-\sqrt{n}}{2}, \frac{n+\sqrt{n}}{2} \right].$$

Теперь рассмотрим, если $\frac{n(n-1)}{2}$ нечетное, следовательно $\lceil \frac{n(n-1)}{4} \rceil = \frac{\frac{n(n-1)}{2}+1}{2} = \frac{n^2-n+2}{4}$. Идентичным способом, как и для четных, оценим m:

$$m(n-m) \geqslant \frac{n^2-n+2}{4}.$$

Аналогично получим промежуток значений:

$$m \in \left[\frac{n}{2} - \frac{\sqrt{n-2}}{2}, \frac{n}{2} + \frac{\sqrt{n-2}}{2}\right].$$

Объеденим результаты этих оценок m и можем сделать вывод, что необходимо искать время, за которое алгоритм находит разрез, где

$$m \in \left[\frac{n}{2} - \frac{\sqrt{n-2}}{2}, \frac{n}{2} + \frac{\sqrt{n-2}}{2}\right].$$

Для всех ниже описанных алгоритмов считаем, что начинаем запуск с $m = 0$. Обозначим m_t — число вершин справа от разреза (с единицами в разбиении P) после t итераций алгоритма.

Для последующего сравнения теоретических оценок времени работы будем анализировать следующие алгоритмы:

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

Полученные теоретические оценки сформулируем в виде теорем.

Random Local Search — достаточно популярный эволюционный алгоритм, основанный на мутации. На каждой итерации алгоритма инвертируется один случайный бит в разбиении x, полученное разбиение x' заменяет родителя, если функция приспособленности не ухудшилось.

Для получения теоретической оценки обозначим T_i как время (число вычислений функции приспособленности), необходимое алгоритму, чтобы увеличить число вершин справа от разбиения, то есть лежащих в V_2, с i до $i+1$. Тогда общее время работы равно

$$T = \sum_{i=o}^{\frac{n}{2} - \frac{\sqrt{n-2}}{2}} T_i.$$

Теорема 1. Математическое ожидание числа вычислений функции приспособленности, которое произведет Random Local Search до нахождения оптимума в задаче поиска максимального разреза графа на полных графах равно

$n \ln 2 - o(n)$, если алгоритм начинает свою работу, когда ни одного ребра не разрезано.

Доказательство (Теорема 1). Вероятность выбрать случайно вершину слева от разреза есть $Pr[i \to i+1] = \frac{n-i}{n}$, где i равно количеству вершин справа от разбиения, т.е. число единиц в разбиении P. Отсюда следует, что T_i имеет геометрическое распределение $\mathrm{Geom}(\frac{n-i}{n})$ и $\mathbb{E}[T_i] = \frac{n}{n-i}$.

Посчитаем математическое ожидание времени работы, используя формулу Эйлера для гармонического ряда $H_n = \ln n + \gamma + O(\frac{1}{n})$, где γ — константа Эйлера-Маскерони:

$$\begin{aligned}
\mathbb{E}[T] &= \sum_{i=o}^{\frac{n}{2}-\frac{\sqrt{n-2}}{2}} \mathbb{E}[T_i] = \sum_{i=o}^{\frac{n}{2}-\frac{\sqrt{n-2}}{2}} \frac{n}{n-i} = [j = n-i] = n \sum_{j=\frac{n}{2}+\frac{\sqrt{n-2}}{2}}^{n} \frac{1}{j} \\
&= n \left(\sum_{j=1}^{n} \frac{1}{j} - \sum_{j=1}^{\frac{n}{2}+\frac{\sqrt{n-2}}{2}-1} \frac{1}{j} \right) = n \left(\ln(n) - \ln\left(\frac{n}{2} + \frac{\sqrt{n-2}}{2}\right) + O\left(\frac{1}{n}\right) \right) \\
&= n \ln \left(\frac{n}{\frac{n}{2} + \frac{\sqrt{n-2}}{2}} \right) + o(1) = n \ln \left(\frac{2}{1 + \frac{\sqrt{n-2}}{n}} \right) + o(1) \\
&= n \ln 2 - n \ln \left(1 + \frac{\sqrt{n-2}}{n} \right) + o(1) = n \ln 2 - n \cdot \Theta\left(\frac{1}{\sqrt{n}}\right) + o(1) \\
&= n \ln 2 - o(n)
\end{aligned}$$

□

Далее рассмотрим $(1+1)$-ЭА с различными типами мутации. В случае стандартной битовой мутации каждый бит в разбиении x инвертируется с вероятностью $\frac{1}{n}$. Полученный мутант заменяет родителя x, если улучшает функцию приспособленности. Этот алгоритм уже может переносить несколько вершин из одного множества в другое.

Так как число инвертированных бит имеет биномиальное распределение, то математическое ожидание числа инвертированных бит равно 1. Следовательно, данный алгоритм на этой задаче не сильно отличается от рассмотренного ранее RLS.

Теорема 2. Математическое ожидание числа вычислений функции приспособленности, которое произведет $(1+1)$-ЭА со стандартным оператором мутации до нахождения оптимума в задаче поиска максимального разреза гра-

фа на полных графах равно $en\ln 2 - o(n)$, если алгоритм начинает свою работу, когда ни одного ребра не разрезано.

Доказательство (Теорема 2). Посчитаем с какой вероятностью P_{mut} на фазе мутации число единиц в разбиении увеличится. Для этого необходимо инвертировать хотя бы $n - i$ нулевых бит, не затронув единичных, где i — число вершин справа от разреза (число единиц в разбиении). Учитывая тот факт, что $(1 - \frac{1}{n})^{n-1} \geqslant \frac{1}{e}$ получаем:

$$P_{mut} = (n-i)\frac{1}{n}\left(1 - \frac{1}{n}\right)^{n-1} \geqslant \frac{n-i}{en}.$$

Тогда $E[T_i] \leqslant \frac{en}{n-i}$ и по аналогии с приведенным способом оценки в доказательстве Теоремы 1 для RLS, посчитаем математическое ожидание числа итераций до достижения оптимума:

$$\mathbb{E}[T] \leqslant \sum_{i=o}^{\frac{n}{2}-\frac{\sqrt{n-2}}{2}} \mathbb{E}[T_i] \leqslant \sum_{i=o}^{\frac{n}{2}-\frac{\sqrt{n-2}}{2}} \frac{en}{n-i} = [j = n-i] = en \sum_{j=\frac{n}{2}+\frac{\sqrt{n-2}}{2}}^{n} \frac{1}{j}$$
$$= en\left(\ln 2 - \Theta\left(\frac{1}{\sqrt{n}}\right)\right) + o(1) = en\ln 2 - o(n).$$

□

Далее рассмотрим оператор мутации с выбором вероятности по степенному закону. На фазе мутации выбирается число $\lambda \in [1..n]$ из распределения с «тяжелым хвостом» $pow(\beta, n)$, что означает, что вероятность выбрать число из $[1..n]$ пропорциональна $\lambda^{-\beta}$, где $\beta \in (1;2)$ — константный параметр. Далее каждый символ заменяется с вероятностью $\frac{\lambda}{n}$.

Теорема 3. Математическое ожидание числа вычислений функции приспособленности, которое произведет $(1+1)$-ЭА с оператором мутации с выбором вероятности по степенному закону до нахождения оптимума в задаче поиска максимального разреза графа на полных графах равно $\Theta(n^{\beta-1})$, если алгоритм начинает свою работу, когда ни одного ребра не разрезано.

Доказательство (Теорема 3). Используем доказанный факт в [**ссылка**], что для $\beta \in (1,2)$ ожидаемое значение инвертированных бит $\lambda = \Theta(n^{2-\beta})$. Так как в наших условиях задачи число нулей больше числа единиц, то с константной вероятностью такой мутант будет лучше, чем текущая особь на $\Theta(n^{2-\beta})$.

Следовательно, так как за одну итерацию ответ улучшается на $\Theta(n^{2-\beta})$, то всего для достижения оптимума нужно:

$$\mathbb{E}[T] = \frac{\frac{n}{2} - \frac{\sqrt{n-2}}{2}}{\Theta(n^{2-\beta})} = \Theta(n^{\beta-1})$$

□

Такой алгоритм работает значительно лучше на многих сложных задачах. В нашем случае, когда больше половины бит в разбиении нули, выгодно инвертировать их в большом количестве и двигаться к оптимуму шагами больше чем 1.

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

Результат анализа времени работы генетического алгоритма $(1 + (\lambda, \lambda))$ будет сформулирован следующей теоремой:

Теорема 4. Математическое ожидание числа оценок функции приспособленности, которое произведёт $(1 + (\lambda, \lambda))$-ГА до нахождения оптимума в задаче поиска максимального разреза графа на полных графах, удовлетворяет $\mathbb{E}[T] = O(\frac{n\lambda \ln \lambda}{\ln \ln \lambda})$, если алгоритм начинает свою работу, когда ни одного ребра не разрезано.

Обозначим $B' = \{i \in [n] \mid x_i = 0 \cap x'_i = 1\}$ — набор битов из победителя мутации, которые стали единицами из нулей, ℓ - сила мутации. Далее введем две вспомогательные леммы.

Лемма 5. С вероятностью $1 - o(1)$ хотя бы $\frac{\lambda}{8}$ инвертированных бит во время мутации в каждой особи — это 0, инвертированные в 1.

Доказательство. Рассмотрим одну итерацию фазы мутации. x' — особь победителя. 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 $d = d(x)$ — число нулей в разбиении x. Проанализируем генерацию одного из λ потомков. $B_1 = \{i \in [n] \mid x_i = 0 \cap x_i^{(1)} = 1\}$ — набор индексов, на которых биты из нулей инвертировались в единицы. 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 6. Вероятность, что в победителе скрещивания будет хотя бы $\frac{\ln \lambda}{\ln \ln \lambda}$ правильных бит из победителя мутации равна $\Omega(1)$.

Доказательство. Рассмотрим одну из λ операций скрещивания, результатом которой получается особь y^i, где $i \in [\lambda]$.

Пусть δ такое, что $Cut(y^i) \geqslant Cut(x) + \delta$. Необходимо оценить вероятность, обозначим ее $p_{cross,\delta}$, фаза скрещивания выберет δ «хороших» битов (то есть из B') и ни одного из «плохих» (те, что инвертировались в процессе мутации, но не принадлежат B'). В процессе вычислений воспользуемся утверждением, что $\ell \leqslant 2\lambda - 2$ (по границам Чернова с константной вероятностью) и $|B'| \geqslant \frac{\lambda}{k}$, где k — константа, а также неравенством $(1 - \frac{1}{n})^{n-1} \geqslant \frac{1}{e}$

$$p_{cross,\delta} \geqslant C^{\delta}_{|B'|} \left(\frac{1}{\lambda}\right)^{\delta} \left(1 - \frac{1}{\lambda}\right)^{\ell-\delta} \geqslant \left(\frac{|B'|}{\delta}\right)^{\delta} \left(\frac{1}{\lambda}\right)^{\delta} \left(1 - \frac{1}{\lambda}\right)^{2(\lambda-1)}$$
$$\geqslant \left(\frac{|B'|}{\delta\lambda}\right)^{\delta} \left(\frac{1}{e^2}\right) \geqslant \left(\frac{1}{k\delta}\right)^{\delta} \left(\frac{1}{e^2}\right).$$

Для $\delta = \lfloor \frac{\frac{1}{2}\ln\lambda - 1}{\ln\ln\lambda + \ln k} \rfloor$ мы имеем $p_{cross,\delta} \geqslant \frac{1}{\lambda}$.

Тогда вероятность того, что оператор скрещивания выберет δ «хороших» битов и ни одного из «плохих» хотя бы в одной особи после скрещивания равна $1 - (1 - \frac{1}{\lambda})^{\lambda} \geqslant 1 - \frac{1}{e}$.

□

Доказательство (Теорема 4). Используя Леммы 5 и 6 можно показать, что одна итерация алгоритма с вероятностью не менее $1 - \frac{1}{e} - o(1)$ дает решение $Cut(y) \geqslant Cut(x) + \delta$, где $\delta = \Omega(\frac{\ln\lambda}{\ln\ln\lambda})$. Для получения оценки количества итераций применим аддитивную теорему о сносе [**ссылка**] и получим

$$\mathbb{E}[T] \leqslant \frac{\frac{n}{2} - \frac{\sqrt{n-2}}{2}}{\Omega(\frac{\ln\lambda}{\ln\ln\lambda})} = O\left(\frac{n \ln\ln\lambda}{\ln\lambda}\right).$$

Так как на каждой итерации алгоритма происходит еще λ вычислений функции приспособленности на фазе мутации и фазе скрещивания, то общее время работы равно $O\left(\frac{\lambda n \ln\ln\lambda}{\ln\lambda}\right)$.

□

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 $\mathbb{E}[T] = O\left(\frac{\lambda n \ln \ln \lambda}{\ln \lambda}\right)$. 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 НА ВСЕХ ТИПАХ 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 число «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.

Случайные графы генерировались по биномиальной модели Эрдёша-Реньи с вероятностью $\frac{1}{2}$.

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

Для каждой конфигурации алгоритмы запускались по 80 раз для более точного анализа ожидаемого времени работы. Время работы считалось, как число вычислений функции приспособленности. Конечным результатом времени работы алгоритма для каждого типа графа считалось среднее арифмети-

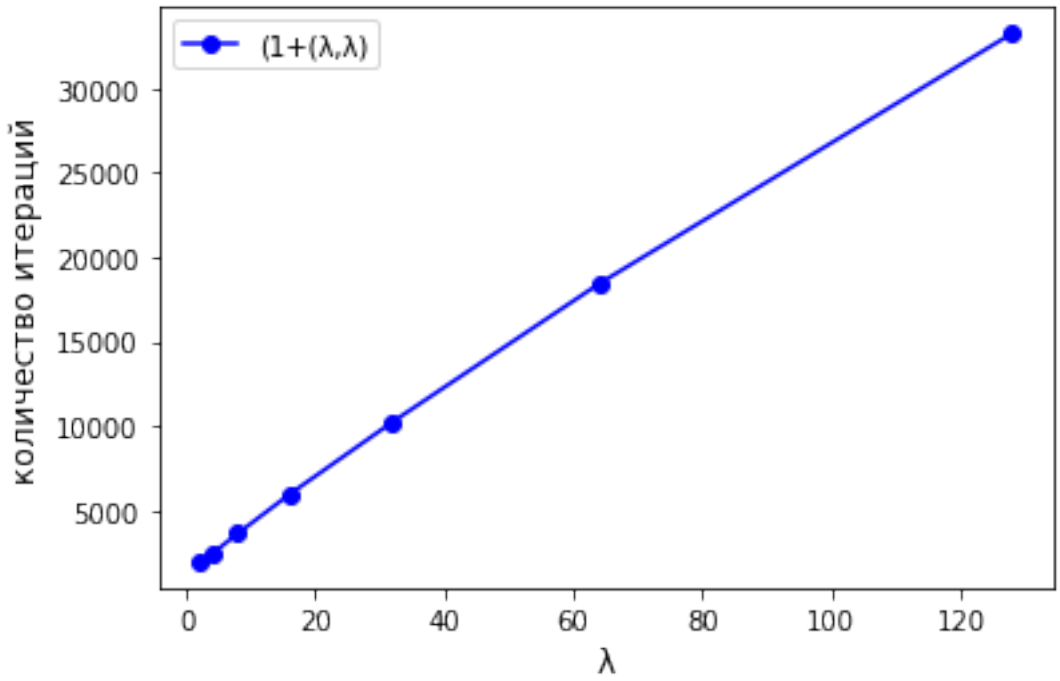

Рисунок 1 – Производительность $(1 + (\lambda, \lambda))$-ГА для различных значений λ на полных графах с числом вершин 2^{10}

ческое результатов времени работы всех 80 запусков. Для отображения полученных значений на графиках таже учитывались отклонения от среднего.

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 числом 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;

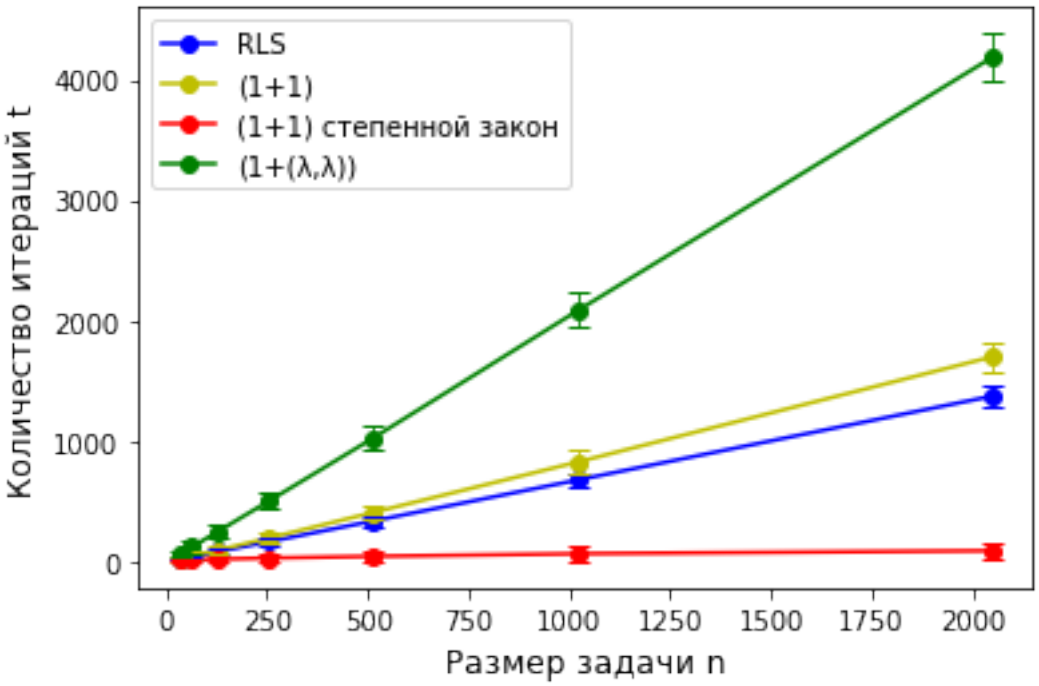

Рисунок 2 – Графики зависимости времени работы эволюционных алгоритмов на полных графах от размера задачи

— значение f для победителя мутации;
— значение f для победителя скрещивания (для $(1 + (\lambda, \lambda))$-ГА).

Такой сбор информации позволяет более детально увидеть и проанализировать работу алгоритма на практике. Также после завершения каждого алгоритма фиксировалось число вычислений функции приспособленности. Реализация всех описанных выше алгоритмов предоставлена в виде программы на языке C++ в Приложении А.1

3.2. Сравнение эмпирической и теоретической оценок на полных графах

Рассмотрим полученные результаты для полных графов и посмотрим как разные эволюционные алгоритмы решают задачу о максимальном разрезе графа на практике. Для них в предыдущей главе уже получена теоретическая оценка. Напомним, что для RLS оценка количества вычислений функций приспособленности до достижения оптимума $nln2 - o(n)$, для $(1 + 1)$-ЭА со стандартным оператором мутации — $enln2 - o(n)$, $(1 + 1)$-ЭА с оператором мутации с выбором мутации по степенному закону — $\Theta(n^{\beta-1})$, но так как для экспериментов использовалась β равная 1,5, ожидаемое время $\Theta(\sqrt{n})$, и для $(1 + (\lambda, \lambda))$-ГА— $O\left(\frac{\lambda n \ln\ln\lambda}{\ln\lambda}\right)$.

Сравним полученные теоретическую и эмпирическую оценку. На Рисунке 2 изображен график зависимости количества вычислений функции приспособленности до достижения оптимума от числа вершин. Сразу можно заметить, что алгоритмы Random Local Search и $(1+1)$-ЭА работают схожим образом. Это связано с тем, что RLS на каждой своей итерации инвертирует один бит и ожидаемое число инвертированных битов в (1+1) так же равно единице. В реальности еще существуют итерации, создающие точную копию родительской особи, что ухудшает работу алгоритма. Алгоритм (1+1) с распределением с «тяжелым хвостом» показал наилучший результат, как и ожидалось. Такой алгоритм движется к оптимуму шагами больше 1, так как может заменить много символов с гораздо большей вероятностью. В нашей задаче больше половины битов нули, поэтому выгодно инвертировать много битов, соответственно этот алгоритм работает значительно лучше чем RLS и (1+1) со стандартным типом мутации. Судя по графику эмпирическая оценка близка к константной.

Что касаемо генетического алгоритма $(1+(\lambda,\lambda))$, он показал свою неэффективность на данной задаче. Согласно полученным результатам $(1+(\lambda,\lambda))$-ГА не дает приимущество над тривиальными алгоритмами. Как минимум из-за дополнительных 2λ вычислений функции приспособленности на каждой итерации алгоритма. Отклонение времени работы $(1+(\lambda,\lambda))$-ГА от среднего составляет около 10%, что также изображено на графике.

3.3. Результаты экспериментов на разных типах графов

Проанализируем как алгоритмы находят решение, где половина ребер «разрезана», на полных двудольных и случайных графах. Для полных двудольных графов такое условие остановки алгоритма дает результат в два раза хуже, чем наилучший, так как в таких графах можно разрезать все ребра. Для случайно сгенерированных графов мы не можем знать заранее на сколько полученный ответ близок к оптимальному. И в отличие от описанного ранее анализа алгоритмов на полных графах, не можем оценить ожидаемое число единиц в искомом разбиении.

На Рисунках 3 и 4 изображены графики зависимости числа вычислений функции приспособленности от размера задачи для полных двудольных и случайных графов соответственно.

Результаты экспериментов схожи с результатами на полных графах. Генетический алгоритм $(1+(\lambda,\lambda))$ все еще не показал свою эффективность

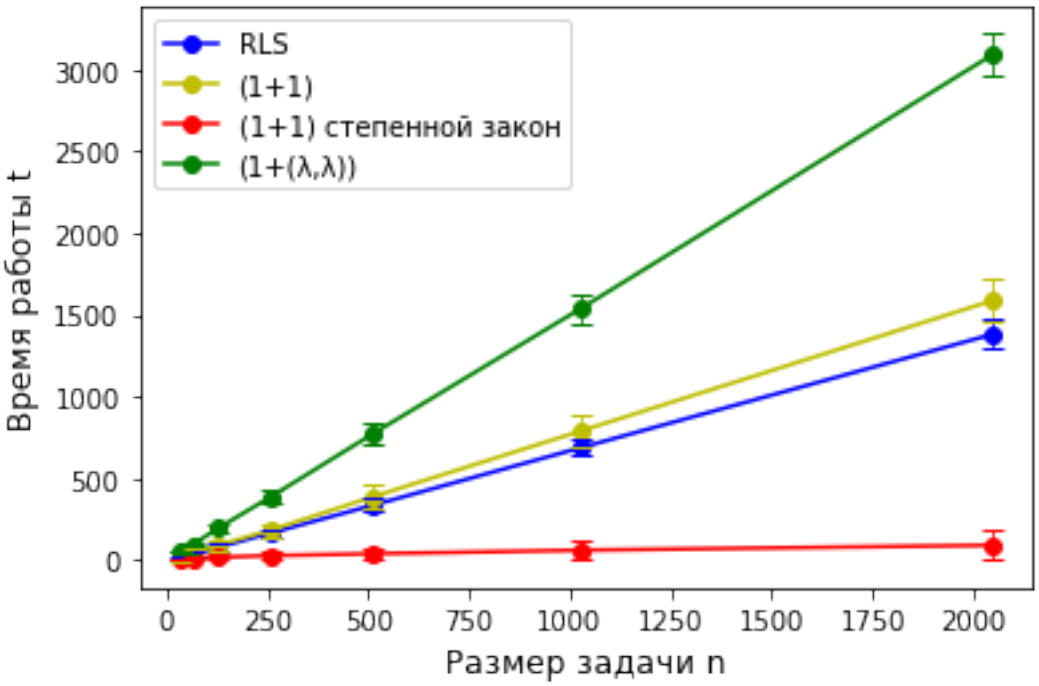

Рисунок 3 – Графики зависимости времени работы эволюционных алгоритмов на полных двудольных графах от размера задачи

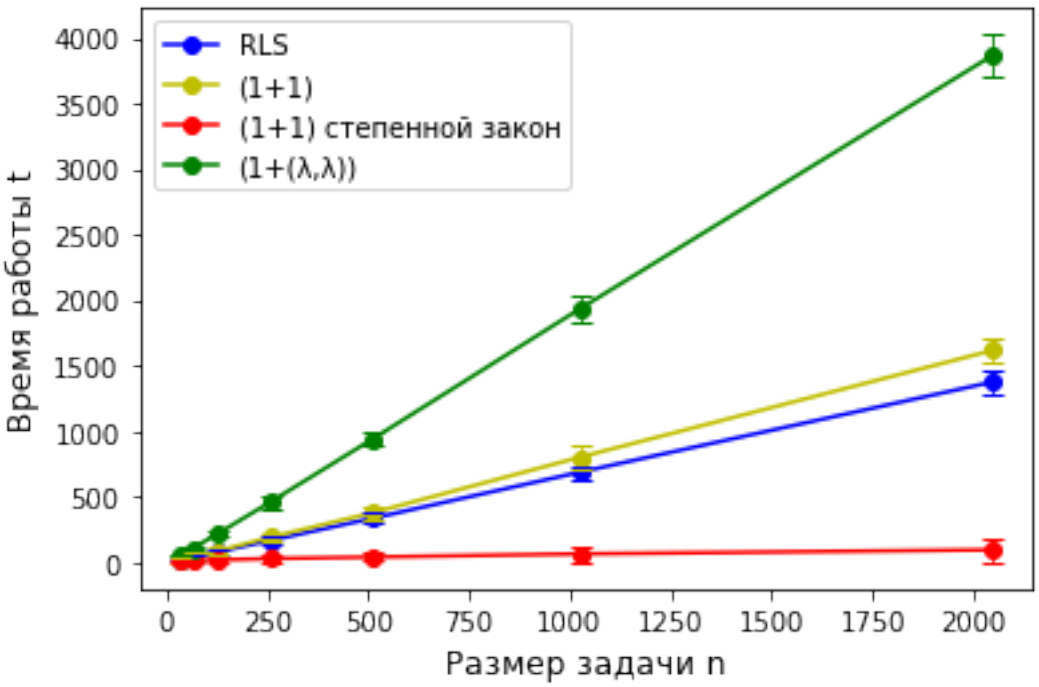

Рисунок 4 – Графики зависимости времени работы эволюционных алгоритмов на случайных графах от размера задачи

по сравнению с другими эволюционными алгоритмами. Заметим, что алгоритм $(1+1)$-ЭА с оператором мутации с выбором вероятности по степенному закону показал наилучший результат и на других типах графов. Исходя из

проведенных экспериментов, можно сделать вывод, что алгоритмы работают схожим образом, как и на полных графах, и предполагаемое время работы $(1 + (\lambda, \lambda))$-ГА на задаче максимального разреза графа на всех типах графах — $O\left(\frac{\lambda n \ln \ln \lambda}{\ln \lambda}\right)$.

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 $O\left(\frac{\lambda n \ln \ln \lambda}{\ln \lambda}\right)$.

ЗАКЛЮЧЕНИЕ

В данной работе была проанализирована работа генетического алгоритма $(1 + (\lambda, \lambda))$ на задаче поиска максимального разреза графа. Для полных графов была получена теоретическая оценка ожидаемого числа вычислений функции приспособленности, равная $O\left(\frac{\lambda n \ln \ln \lambda}{\ln \lambda}\right)$, а также проведено сравнение с другими эволюционными алгоритмами. Проанализировано время работы алгоритмов на задаче максимального разреза графа на практике. Из полученных в данной работе результатов можно сделать вывод о неэффективности выбора $(1 + (\lambda, \lambda))$-ГА на данной задаче.

Для продолжения работы по этой теме и дополнительных оценок генетического алгоритма $(1 + (\lambda, \lambda))$ на задаче поиска максимального разреза графа можно провести исследования в области подбора параметров, которые улучшат эффективность алгоритма, а также различные модификации операторов мутации и скрещивания.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1 *Скобцов Ю. А.* Основы эволюционных вычислений. — 2008.

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(C:/Users/Viktoriya/CLionProjects/evol/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;
}


#include <vector>

#ifndef EVOL_GRAPH_H
#define EVOL_GRAPH_H

using namespace std;

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();
};

#endif //EVOL_GRAPH_H

#include <vector>
#include <iostream>
#include <stdlib.h>
#include <fstream>
#include <ctime>
#include <random>
```

```
#include <chrono>
#include "graph.h"

using namespace std;

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("C:/Users/Viktoriya/CLionProjects/evol/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 << "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);
    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 l + 1;
}

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

    out << "flip " << pos << '\n';

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

    long long newD = countD(newE);

    vector<int> newd = countd(newE);

    out << "старый потенциал = " << getD() << "\nослеn 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";

    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() << "\nослеn 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() << "\nослеn 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() << "\nослеп 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;
        }
    }
```