

НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»

Факультет Компьютерных Наук

Департамент Программной Инженерии

Контрольное Домашнее Задание

по дисциплине «Алгоритмы и Структуры Данных»

ОТЧЕТ

Выполнил:
Студент 2 курса группы БПИ151
Куприянов Кирилл Игоревич

Москва 2016

Содержание

1	Постановка задачи	3
2	Описание алгоритмов и использованных СД	5
2.1	Для решения задачи алгоритмом Хаффмана	5
2.1.1	Структуры данных	5
2.1.2	Алгоритмы и функции	5
2.2	Для решения задачи алгоритмом Шеннона-Фано	6
2.2.1	Структуры данных	6
2.2.2	Алгоритмы и функции	6
3	Описание плана эксперимента	8
4	Результаты эксперимента - таблицы и графики	9
4.1	Таблицы	9
4.2	Графики	10
4.2.1	Первый набор графиков	11
4.2.2	Второй набор графиков	15
4.2.3	Третий набор графиков	21
4.3	Дополнительные графики	23
5	Сравнительный анализ методов	25
6	Заключение	25
7	Список используемых источников	26

т.е. Всего минимум $144 \times 2 = 288$). Для повышения достоверности результатов каждый эксперимент можно повторить несколько (5-10) раз на файлах (с одним возможным набором символов) одного размера с последующим усреднением результата.

Подготовить отчет по итогам работы, содержащий постановку задачи, описание алгоритмов и задействованных структур данных, описание реализации, обобщенные результаты измерения эффективности алгоритмов, описание использованных инструментов (например, если использовались скрипты автоматизации), выводы о соответствии результатов экспериментальной проверки с теоретическими оценками эффективности исследуемых алгоритмов. Отчет также должен содержать измерения качества архивации (степень сжатия = отношение размеров выходного и входного файлов), оценку связи между степенью сжатия для различных входных файлов (как влияют объем, язык, набор символов, их разнообразие?) и временем работы (количеством операций) для каждого алгоритма.

Было выполнено: Всё. + построены дополнительные графики

2 Описание алгоритмов и использованных СД

2.1 Для решения задачи алгоритмом Хаффмана

2.1.1 Структуры данных

Для реализации архивирования - деархивирования алгоритмом Хаффмана я использовал следующие **структуры данных**:

- **Бинарное дерево** с узлами - указателями на объекты класса Node. Для их хранения использовался двусвязный список.
- **map <char, int>** - для хранения таблицы частот
- **map <char, vector<bool> >** - для хранения таблицы вида «символ - его код»

2.1.2 Алгоритмы и функции

Описание использованных **алгоритмов**

Использовались следующие функции:

- **vector<char> getSymbols(string)** - для заполнения вектора символов символами из файла. По указанному пути файла создает поток и считывает все символы в вектор. Затем удаляет (делает `push_back`) последний элемент - константу EOF, поскольку она лишняя. Сложность - $O(n)$
- **map<char, int> getFreq(vector<char>)** - для составления таблицы частот. По данному вектору символов составляет `map<char, int>` - где каждому символу ставится в соответствие его частота появления в данном вектре
- **void buildTree(map<char, int>)** - для построения дерева. Создает `list<Node*>` для содержания узлов деревьев. Изначально каждый Node в списке имеет значение `c = char` из входной `map`, `n = частоте появления` (значение `int` во входной `map`). Указатели на левых и правых детей равны `nullptr`. Процедура построения дерева происходит следующим образом: Список сортируется по возрастанию частоты появления символов (для этого использую функцию `sort` и структуру `Compare`, где перегружаю оператор `()`). Затем берутся первые 2 элемента списка, они становятся детьми нового узла, который кладется в начало списка, а 2 прошлых - удаляются. Эта процедура происходит рекурсивно до тех пор, пока не останется в списке только 1 элемент - корень. Сложность - $O(n)$
- **void buildTable(Node*)** - для построения таблицы кодов, используя дерево. Начиная с корня, идем по дереву налево. Если левый ребенок не равен `nullptr`, добавляем во временный вектор `code 0` и вызываем эту функцию, передавая в качестве аргумента указатель на этого ребенка. Аналогично с правым ребенком, только добавляем в вектор `code не 0, а 1`. Если же и левый и правый дети - `nullptr`, то мы считаем, что вектор `code` представляет собой код символа, который находится в текущем узле и записываем это соответствие в `map <char, vector <bool> >`
- **void encodeHuff(string, string)** - для «архивации» файла. Архивация происходит следующим образом. Открываю потоки ввода и вывода. Для потока вывода ставлю флаги - `std::ios::binary`, и `std::ios::out`, поскольку я буду писать в бинарный файл. Считываю все символы из входного файла в `vector<char>`, с помощью указанной выше функции. Записываю в выходной файл первым делом длину того вектра - количество символов в исходном файле. Это понадобится дальше при разархивировании. Затем составляю

таблицу частот и вторым байтом записываю длину таблицы, другими словами, кол-во уникальных символов. Если эта длина равна единице, значит, что исходный файл заполнен одним конкретным символом "первый байт" количество раз. В таком случае, мы его считываем и пишем его столько раз в выходной файл. При разархивировании это учитываем. Если же количество уникальных символов не равно 1, то продолжаем. Записываем таблицу соответствий сюда «символ - его частота». Затем строим дерево, таблицу кодов, проходимся по исходному файлу и выводим в архив код каждого символа следующим образом:

Поскольку писать побитово нельзя, пишем побайтово. Аккумулируем биты в переменной `buf` и считаем сколько бит мы уже записали. Как только это количество станет равно 8, пишу `buf` в файл и обнуляем его. В конце, если у нас кол-во битов оказалось не кратным 8, пишу то, что осталось для того, чтобы заполнить недостающие биты. НЕ считать лишнего при разархивировании нам поможет первый байт - кол-во символов в исходном сообщении.

- **`void decodeHuff(string , string)`** - для «деархивации» файла. Открываю потоки, считываю первый байт - кол-во символов. Если оно равно 1, то считываю второй байт - сколько раз повторяется этот конкретный символ. Затем считываю этот символ и пишу в аутпут его такое количество раз. Если же первый байт не единица, то идем дальше. Считываем длину таблицы частот и такое количество раз считываем следующие байты, попутно инициализируя таблицу частот. На основе этой таблицы строится дерево. Затем декодирую само сообщение. Считываю байт информации в `char byte` и смотрю на него побитово, если бит равен 1, то иду по дереву направо, если 0, то налево. Когда дошел до листа, я дописываю в результирующую строку символ в листе. Когда просмотрел 8 битов, считываю новый. Затем, после того, как прочитал весь файл, записываю в аутпут результирующую строку, но не всю, а только такое количество символов, сколько я считал в начале.

2.2 Для решения задачи алгоритмом Шеннона-Фано

2.2.1 Структуры данных

Для реализации архивирования - деархивирования алгоритмом Шеннона-Фано я использовал следующие **структуры данных**:

- **`struct node`** - основная структура для работы с алгоритмом Шеннона-Фано с двумя полями - `char ch` (символ) и `float p` (его вероятность появления в тексте)
- **`node *freqTable`** - таблица вероятностей, представляет собой динамический массив из элементов `node`
- **`map<char, vector<bool> > table`** - аналогичная (таблице кодов Хаффмана) таблица, которой каждому символу поставлен в соответствие его код

2.2.2 Алгоритмы и функции

Описание использованных **алгоритмов**

Использовались следующие функции:

- **`void shannonFano(int, int)`** - функция, которая, собственно, выполняет процедуру Шеннона-Фано. Строит таблицу «символ-код». Это рекурсивная функция. База: `return`, если в группе всего лишь 1 символ. Если 2 символа, первому добавляем к коду 0, а ко второму - 1. Для остальных вариантов: проходим по (отсортированной!) таблице, постепенно инкрементируя текущую сумму вероятностей до момента пока она не станет

больше либо равной половине полной вероятности, а пока она меньше половины, мы на каждом шаге приписываем к коду текущего символа 0. Как только этот момент настал, добавляем к кодам тех элементов, что ниже 1. Рекурсивно вызываем функцию для первой и второй групп. Получили построенную таблицу кодов по алгоритму Шеннона-Фано.

- **void encodeSF(string, string)** - функция, идентичная аналогичной для кода Хаффмана. Длина сообщения записывается первым битом, кол-во уникальных символов - вторым, затем таблица соответствий символ - вероятность, затем само сообщение, так же, как и для алгоритма Хаффмана. Со случаем, когда длина таблицы = 1 разбираемся так же, как и там. Различие - в моей реализации задачи алгоритмом Шеннона-Фано *не используется* дерево, используется лишь таблица кодов.
- **void decodeSF(string, string)** - функция, идентичная аналогичной для кода Хаффмана. После получения таблицы вероятностей, вызываю функцию shannonFano для построения таблицы кодов. По ней и декодируется сообщение, лежащее в последних байтах архива. Как только получаем новый битик, добавляем его во временный vector<bool> и по мере добавления, проверяю, есть ли он в таблице кодов с помощью функции isInTable. Если он там есть, ищу символ, соответствующий этому коду при помощи функции getchar.
- **char getchar(vector<bool>)** - вспомогательная функция. Возвращает символ, которому в таблице кодов соответствует данный вектор или 0, если такого нет.

3 Описание плана эксперимента

План эксперимента состоит в следующем:

- Сгенерировать набор тестовых файлов
- Запустить программу на них, подсчитав количество операций, и получив таблицу
- По полученной таблице построить графики
- Проанализировать результаты эксперимента

Для начала нужно сгенерировать тестовый набор из файлов разных размеров (20, 40, 60, 80, 100 Кб, 1, 2, 3 Мб) и разных наборов символов, каждого по три штуки, итого $8 \times 3 \times 3 = 72$ файла. Для генерации тестового набора файлов будет написан скрипт на языке Python.

Для подсчета количества операций при архивировании - деархивировании двумя алгоритмами этих 72х файлов, я будет сделано следующее. Будет глобальная переменная, изначально равная 0 **long numberOfOperations** и она будет инкрементироваться там, где необходимо подсчитать действие за операцию. Вызов библиотечных функций будет считаться за одну операцию. В конце выполнения программы это число выведется в отдельный файл.

Затем для подсчета операций при всех упомянутых выше действий, будет написан еще один скрипт на языке Python. Он попутно формировать csv таблицу, записывая в нее данные о размере файлов, количестве операций, используемом алгоритме, и т.д. (табл. 1)

name	old_size	new_size	operations	algo	set	action	num	size
------	----------	----------	------------	------	-----	--------	-----	------

Таблица 1.

Для более удобного построения графиков нужно будет отфильтровать таблицу, усреднив количества операций для трех дубликатов. Отфильтровка будет производиться в тетрадке jupyter notebook на языке Python, при помощи библиотек pandas и numpy.

Построение графиков будет проводиться в той же среде, при помощи библиотеки matplotlib.

4 Результаты эксперимента - таблицы и графики

4.1 Таблицы

Ниже приведена часть отфильтрованной таблицы. Полная таблица представлена в файле filtered.csv.

name	old_size	new_size	operations	algo	set	action	num	size
set_100_1_1.haff	77611	106509	1427272	haff	1	d	1	100
set_100_2_1.haff	76829	106512	1435777	haff	2	d	1	100
set_100_3_1.haff	83701	106510	1570388	haff	3	d	1	100
set_1_1_1.haff	732537	1007739	13462131	haff	1	d	1	1
set_1_2_1.haff	724456	1007782	13341756	haff	2	d	1	1
set_1_3_1.haff	785493	1007785	14406953	haff	3	d	1	1
set_20_1_1.haff	18098	24579	333718	haff	1	d	1	20
set_20_2_1.haff	18202	24582	356596	haff	2	d	1	20
set_20_3_1.haff	19873	24579	404576	haff	3	d	1	20
set_2_1_1.haff	1458962	2007285	26810733	haff	1	d	1	2
set_2_2_1.haff	1441710	2007381	26543748	haff	2	d	1	2
set_2_3_1.haff	1564668	2007370	28647416	haff	3	d	1	2
set_3_1_1.haff	2185326	3006831	40159281	haff	1	d	1	3
set_3_2_1.haff	2159711	3006951	39751799	haff	2	d	1	3
set_3_3_1.haff	2342968	3006948	42881756	haff	3	d	1	3
set_40_1_1.haff	30004	40965	552385	haff	1	d	1	40
set_40_2_1.haff	29949	40966	572777	haff	2	d	1	40
set_40_3_1.haff	32695	40967	638421	haff	3	d	1	40
set_60_1_1.haff	47860	65544	880456	haff	1	d	1	60
set_60_2_1.haff	47573	65547	896903	haff	2	d	1	60
set_60_3_1.haff	51786	65548	988454	haff	3	d	1	60
set_80_1_1.haff	59766	81930	1099230	haff	1	d	1	80
set_80_2_1.haff	59301	81931	1113272	haff	2	d	1	80
set_80_3_1.haff	64449	81932	1221124	haff	3	d	1	80
set_100_1_1.shan	77676	106509	1636177	shan	1	d	1	100
set_100_2_1.shan	77126	106512	1623705	shan	2	d	1	100
set_100_3_1.shan	83778	106510	1733023	shan	3	d	1	100
set_1_1_1.shan	732700	1007739	15475307	shan	1	d	1	1
set_1_2_1.shan	726084	1007782	15358870	shan	2	d	1	1
set_1_3_1.shan	785787	1007785	16375760	shan	3	d	1	1
set_20_1_1.shan	18130	24579	378189	shan	1	d	1	20
set_20_2_1.shan	18275	24582	376902	shan	2	d	1	20
set_20_3_1.shan	19900	24579	402449	shan	3	d	1	20
set_2_1_1.shan	1459177	2007285	30824264	shan	1	d	1	2
set_2_2_1.shan	1444790	2007381	30581392	shan	2	d	1	2
set_2_3_1.shan	1565095	2007370	32617358	shan	3	d	1	2
set_3_1_1.shan	2185690	3006831	46173351	shan	1	d	1	3
set_3_2_1.shan	2164260	3006951	45813148	shan	2	d	1	3
set_3_3_1.shan	2343503	3006948	48852992	shan	3	d	1	3
set_40_1_1.shan	30037	40965	629726	shan	1	d	1	40
set_40_2_1.shan	30069	40966	626671	shan	2	d	1	40
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

Описание признаков:

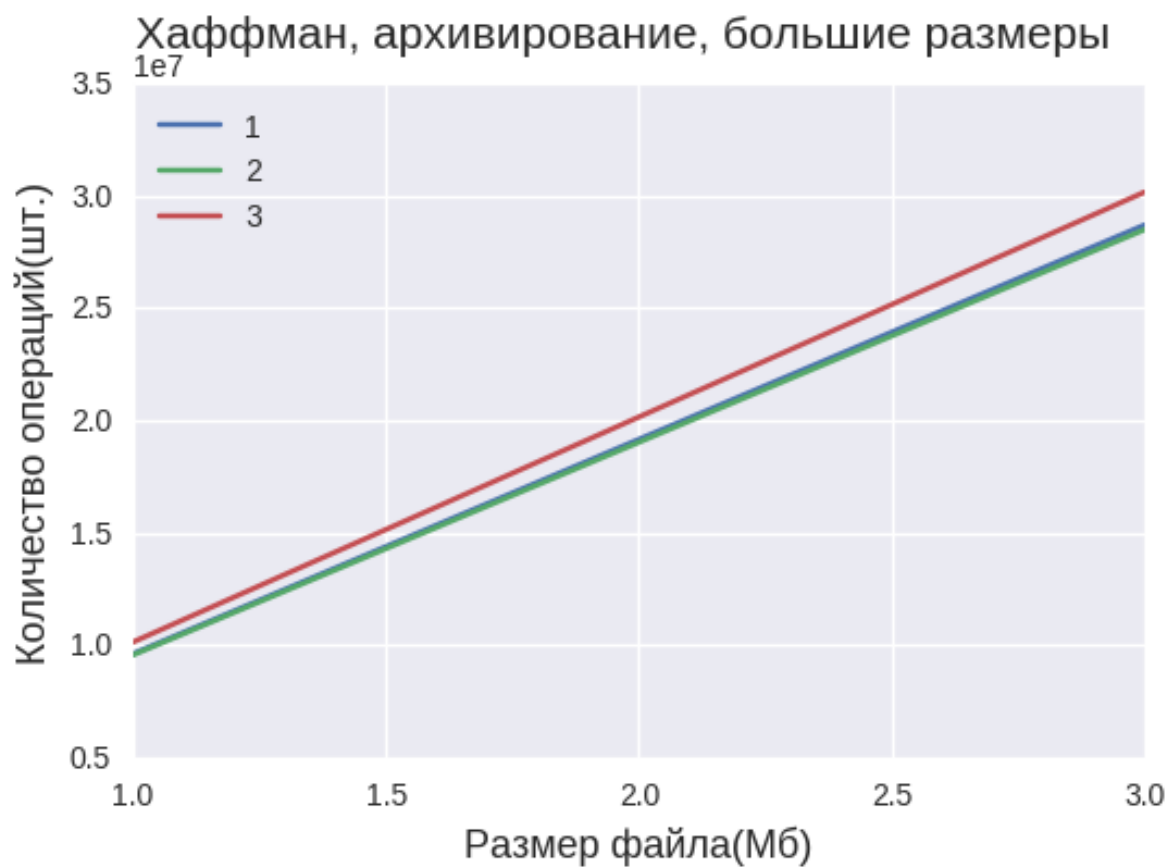
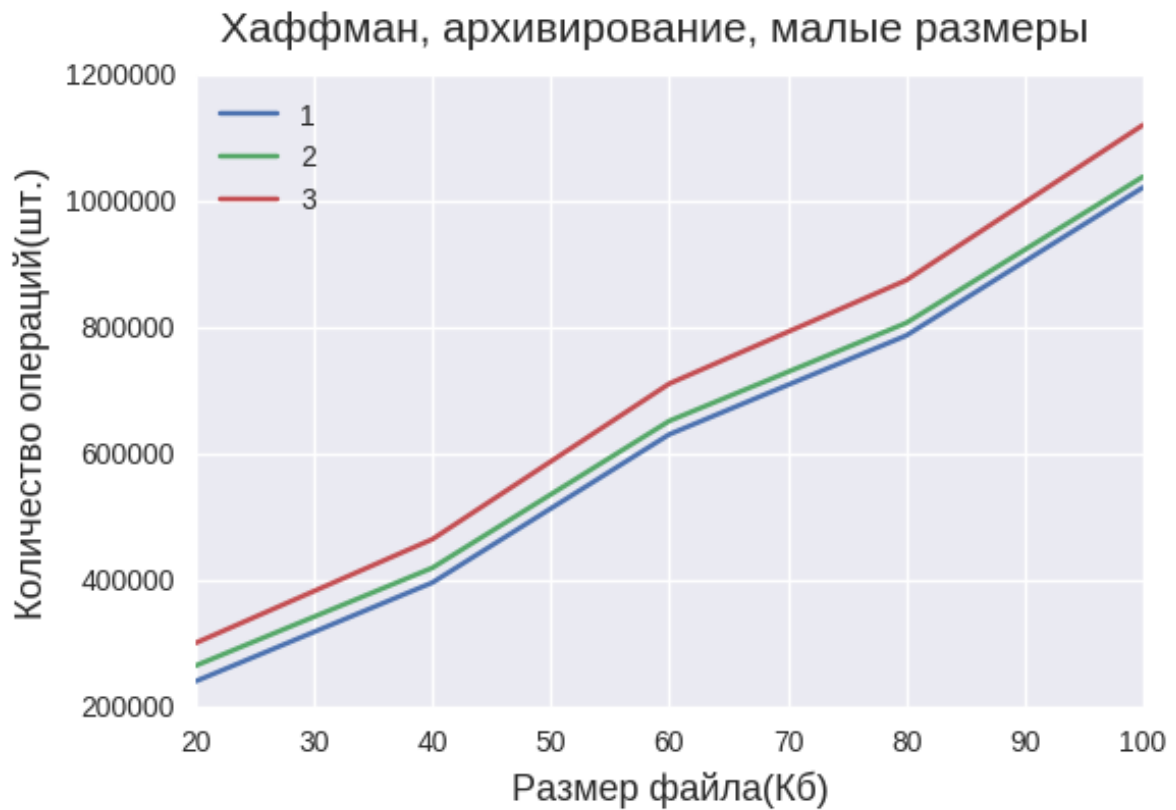
- **name:** имя файла, над которым будет производиться работа. Первая цифра - размер, вторая - набор символов, третья - номер дубликата, но, поскольку таблица уже отфильтрована, везде на месте третьей цифры стоит 1
- **old_size:** старый размер файла, до того, как над ним было совершено действие
- **new_size:** новый размер файла, после того, как над ним было совершено действие
- **operations:** количество операций, произошедших при совершении действия
- **algo:** алгоритм, при помощи которого выполнялось действие
- **set:** номер набора символов, взят из названия
- **action:** действие, которое и совершалось над файлом (с - compress, d - decompress)
- **num:** номер дубликата, взят из названия
- **size:** размер файла, взят из названия

4.2 Графики

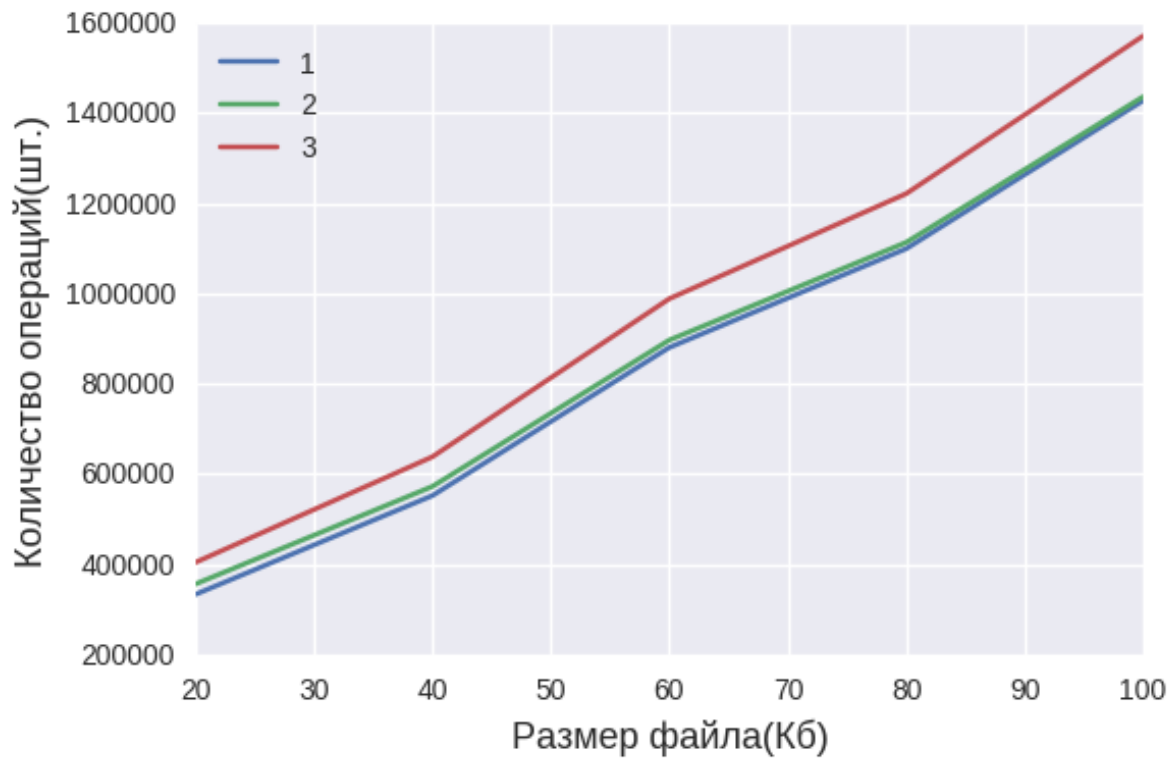
По приведенной таблицы были построены следующие графики (всего $8 + 12 + 2 = 22$ шт.):

1. Зависимость количества операций (ось ОУ) от размера файла (ось ОХ) и размера набора символов (цвет линии) для каждого алгоритма архивирования / разархивирования. Должно быть 2 (алгоритма) $\times 2$ (арх/разарх) $\times 2$ (малые и большие файлы отдельно) набора графиков, на каждом 3 кривые (для каждого набора символов свой цвет)
2. Зависимость количества операций (ось ОУ) от размера файла (ось ОХ), и используемого алгоритма (цвет линии) для каждого набора символов. Должно быть 3 (набора символов) $\times 2$ (арх/разарх) $\times 2$ (малые и большие файлы отдельно) набора графиков, на каждом 2 кривые (для каждого алгоритма)
3. Зависимость количества операций (ось ОУ) от размера набора символов (ось ОХ) на файлах максимального размера (3 Мб) для каждого алгоритма архивирования / разархивирования (цвет линии). Должно быть 2 (арх/разарх) набора графиков, на каждом 2 кривые (для каждого алгоритма)

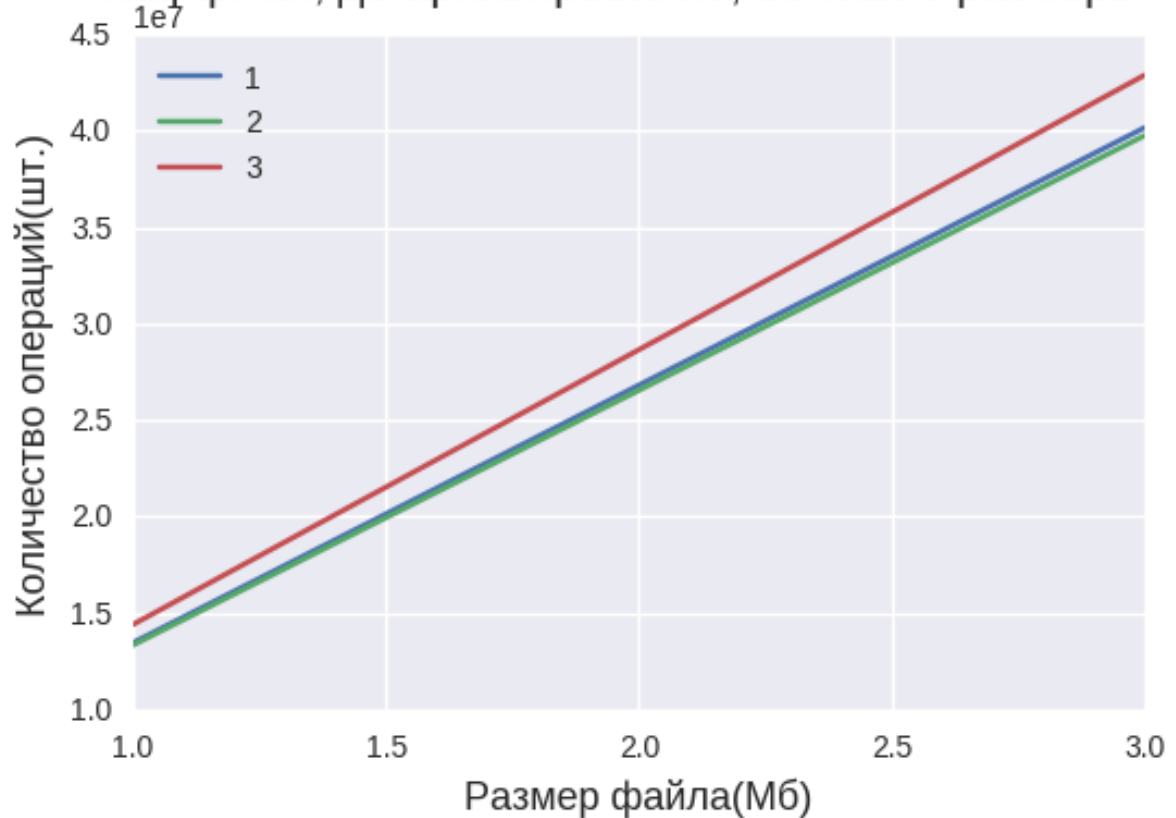
4.2.1 Первый набор графиков



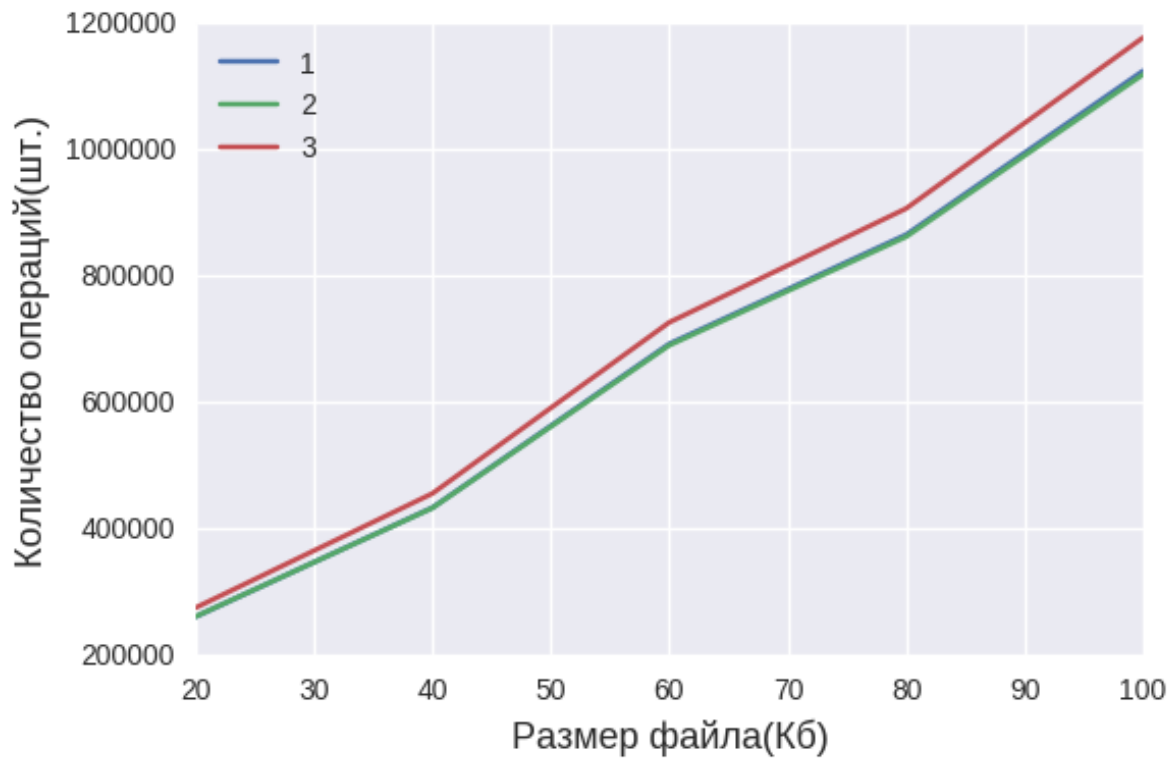
Хаффман, де-архивирование, малые размеры



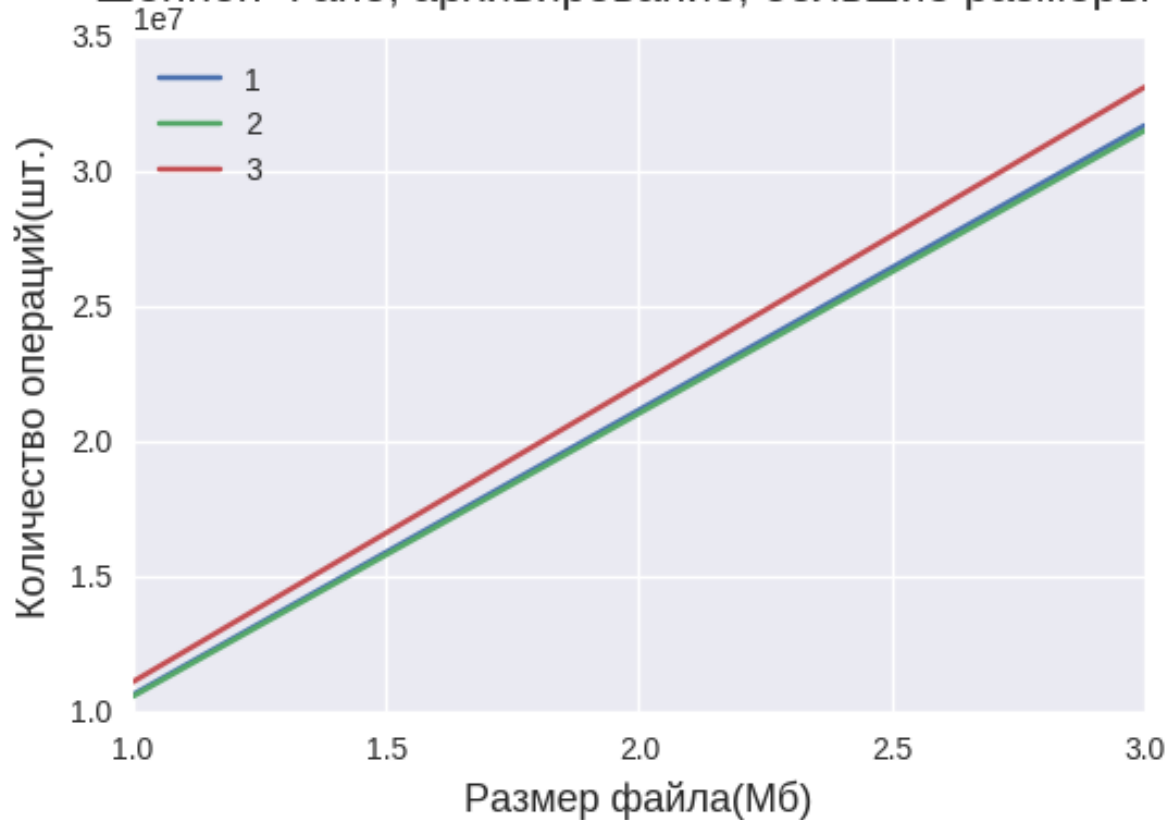
Хаффман, де-архивирование, большие размеры



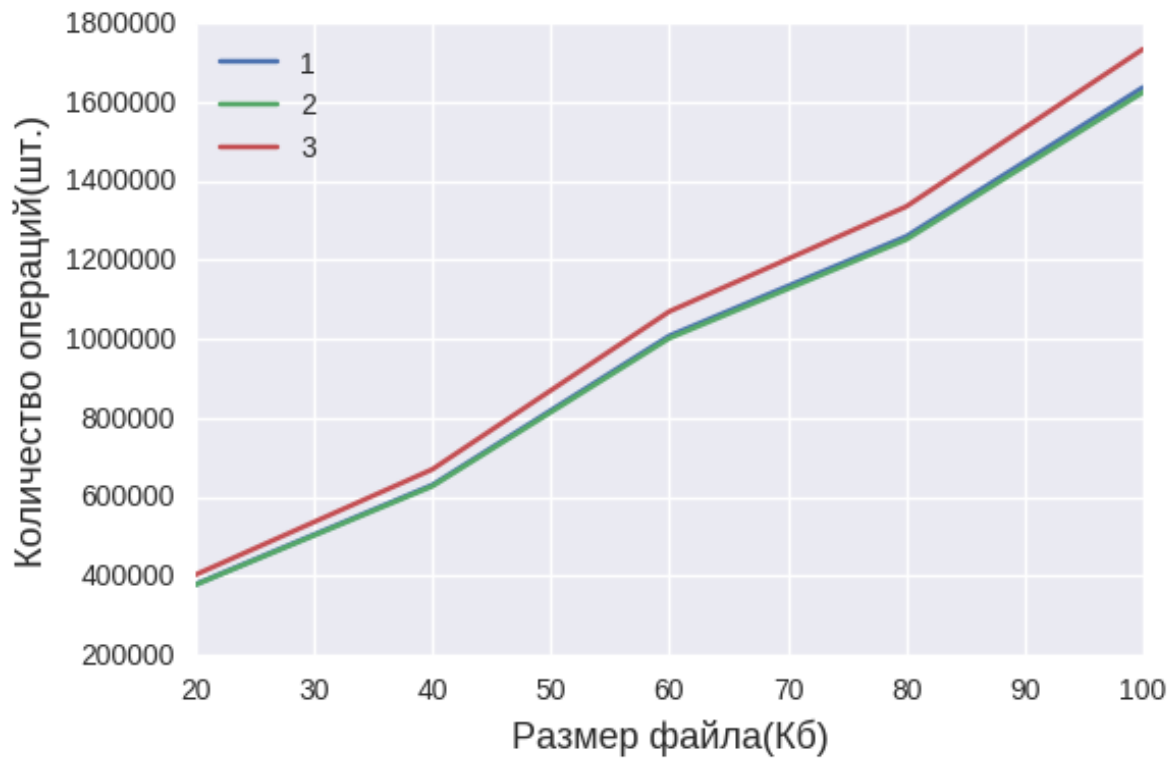
Шеннон_Фано, архивирование, малые размеры



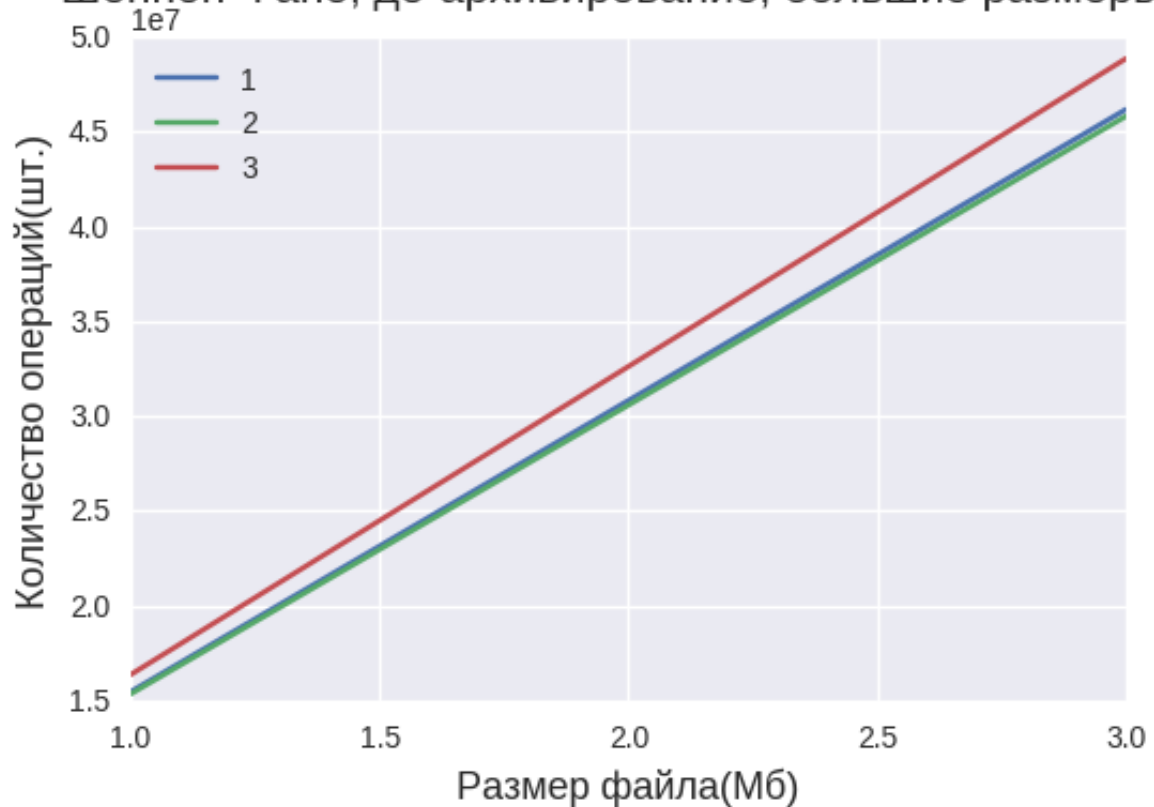
Шеннон-Фано, архивирование, большие размеры



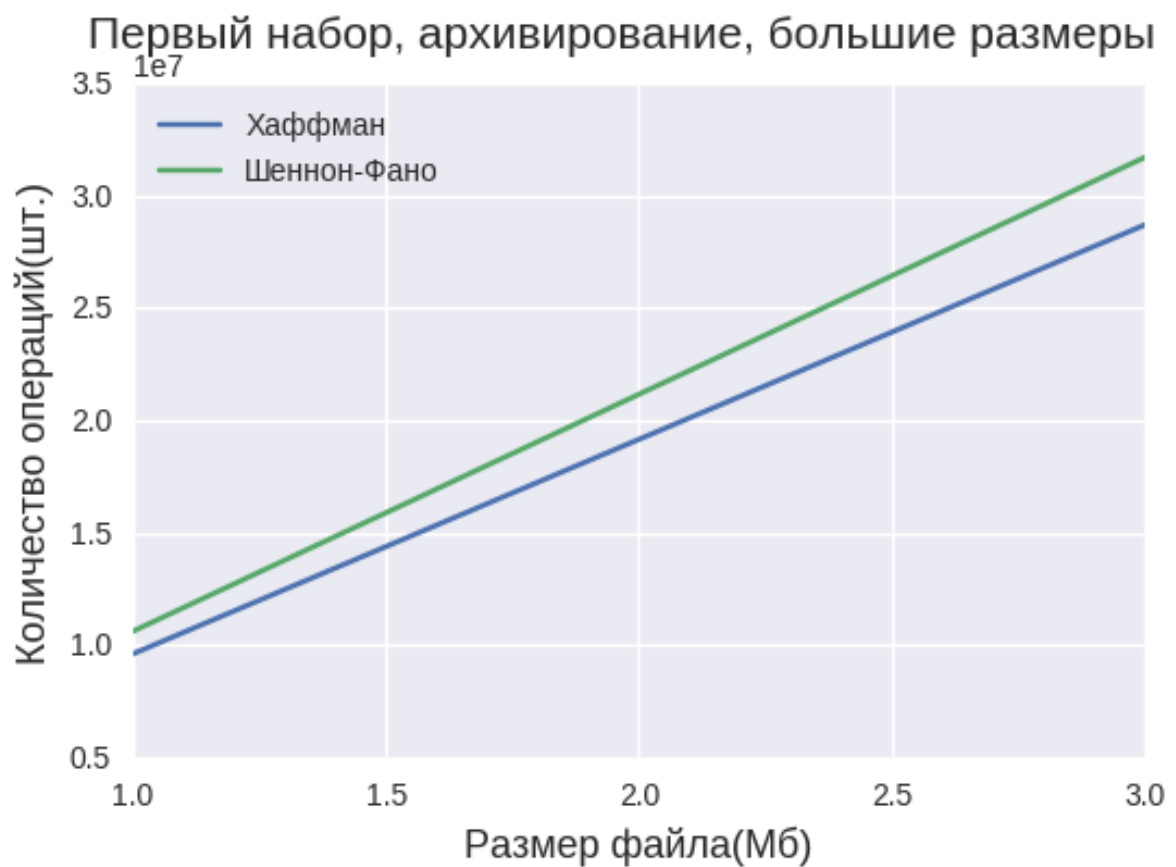
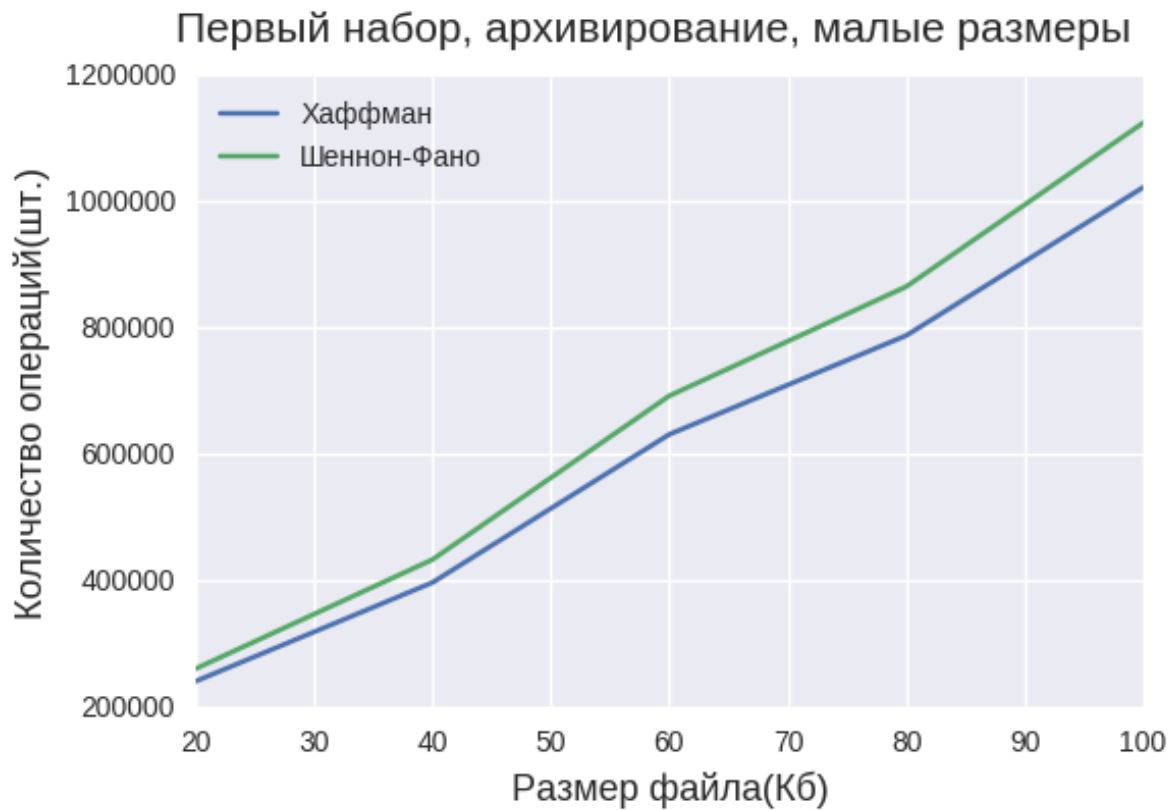
Шеннон-Фано, де-архивирование, малые размеры



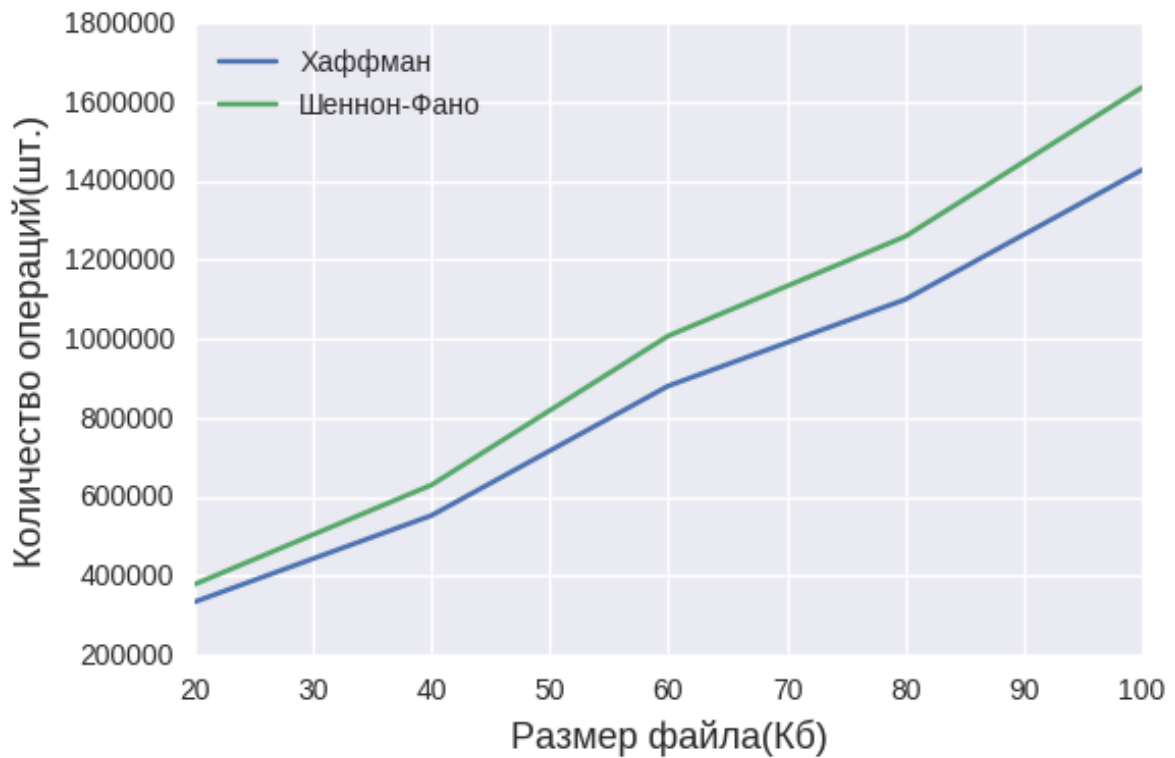
Шеннон-Фано, де-архивирование, большие размеры



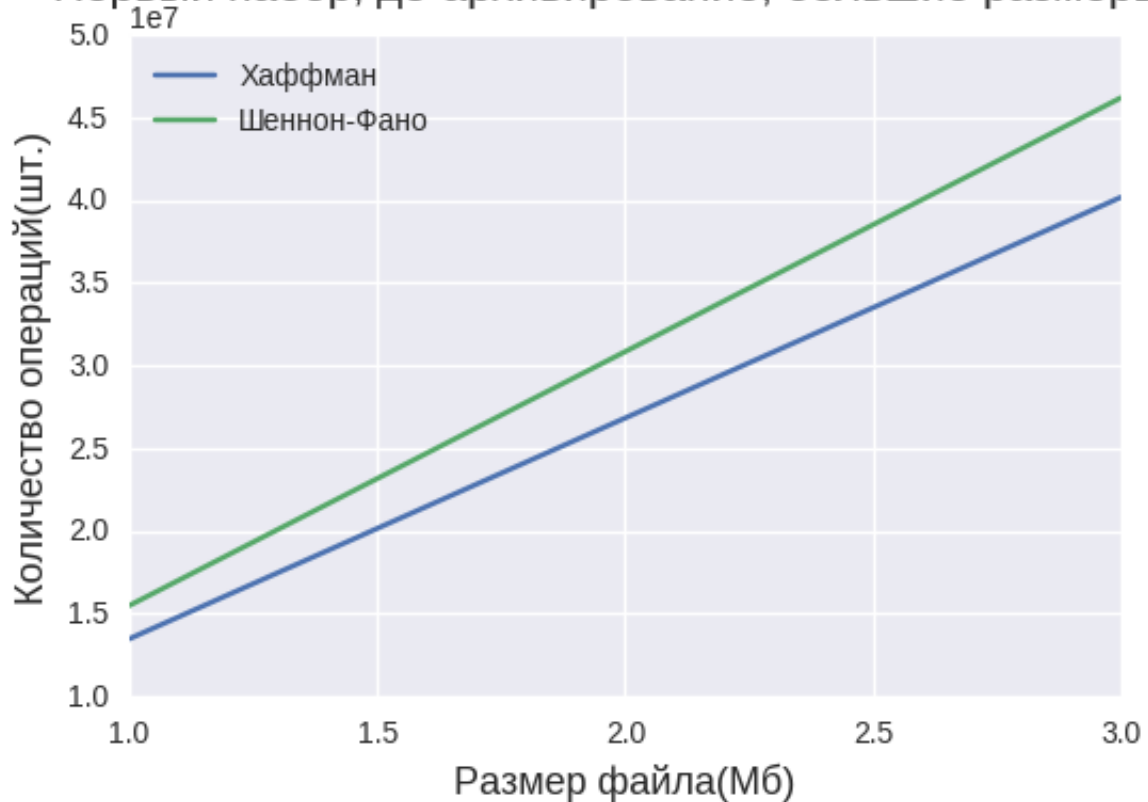
4.2.2 Второй набор графиков



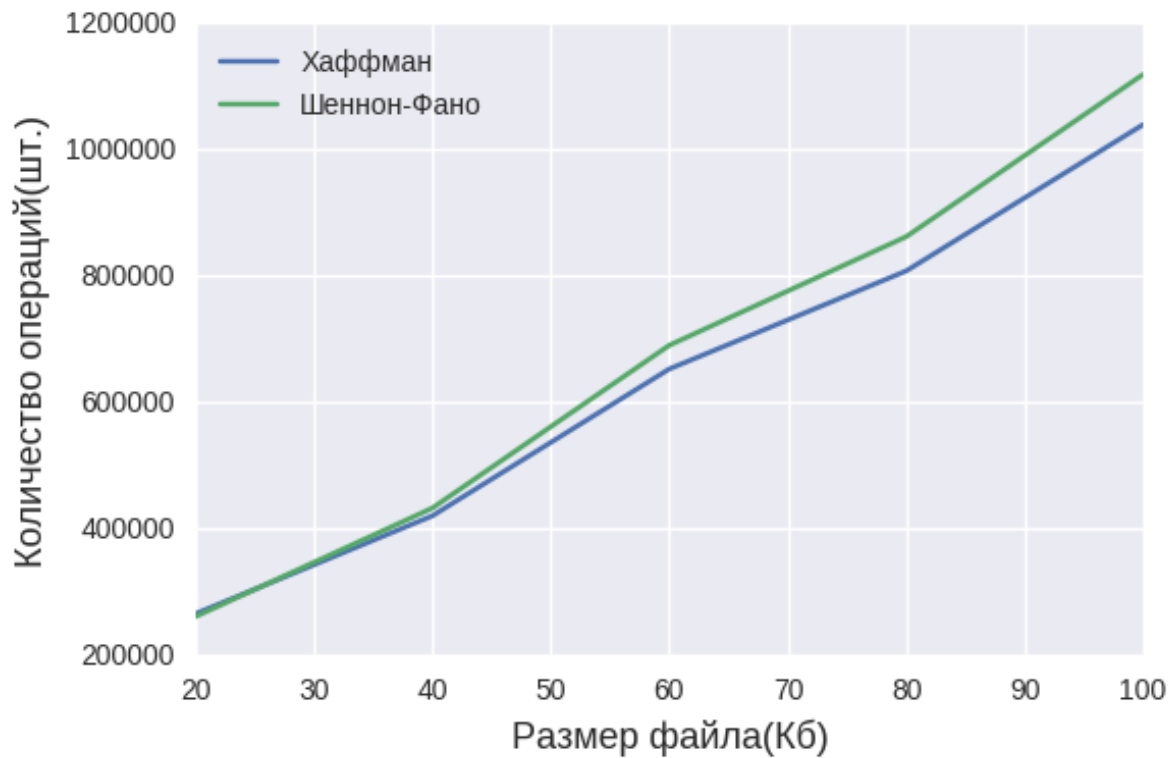
Первый набор, де-архивирование, малые размеры



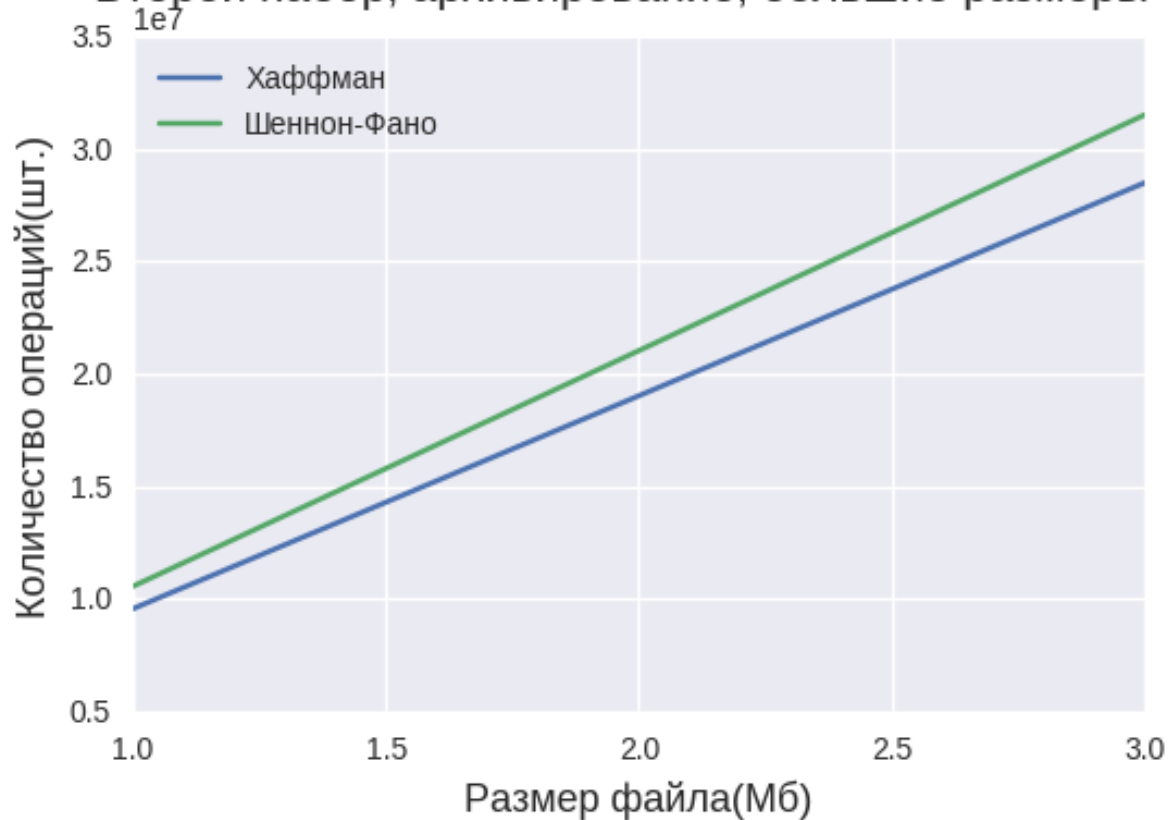
Первый набор, де-архивирование, большие размеры



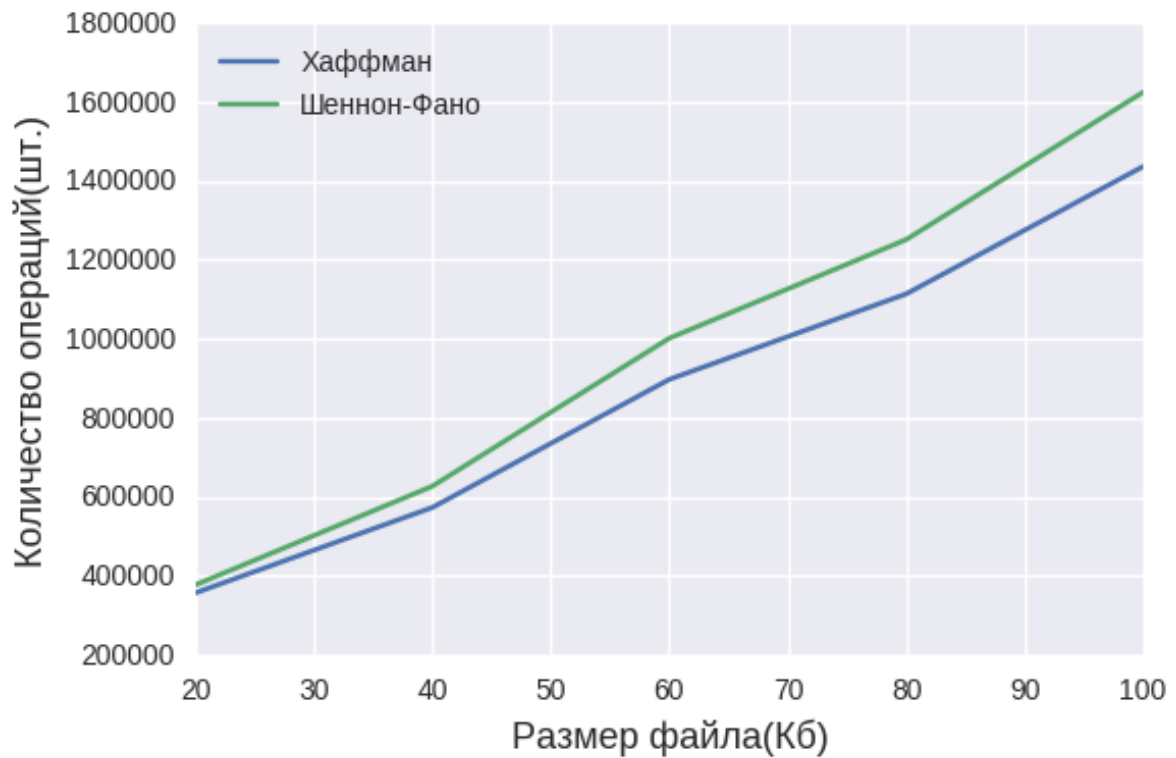
Второй набор, архивирование, малые размеры



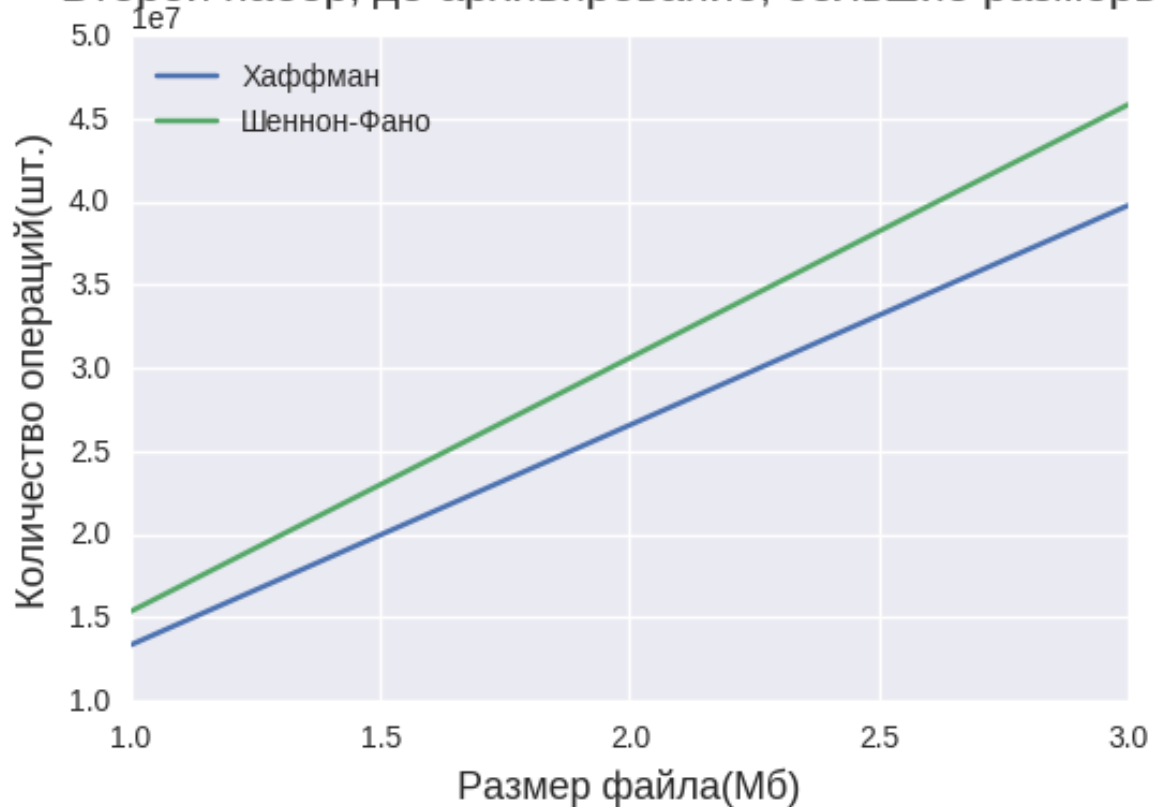
Второй набор, архивирование, большие размеры



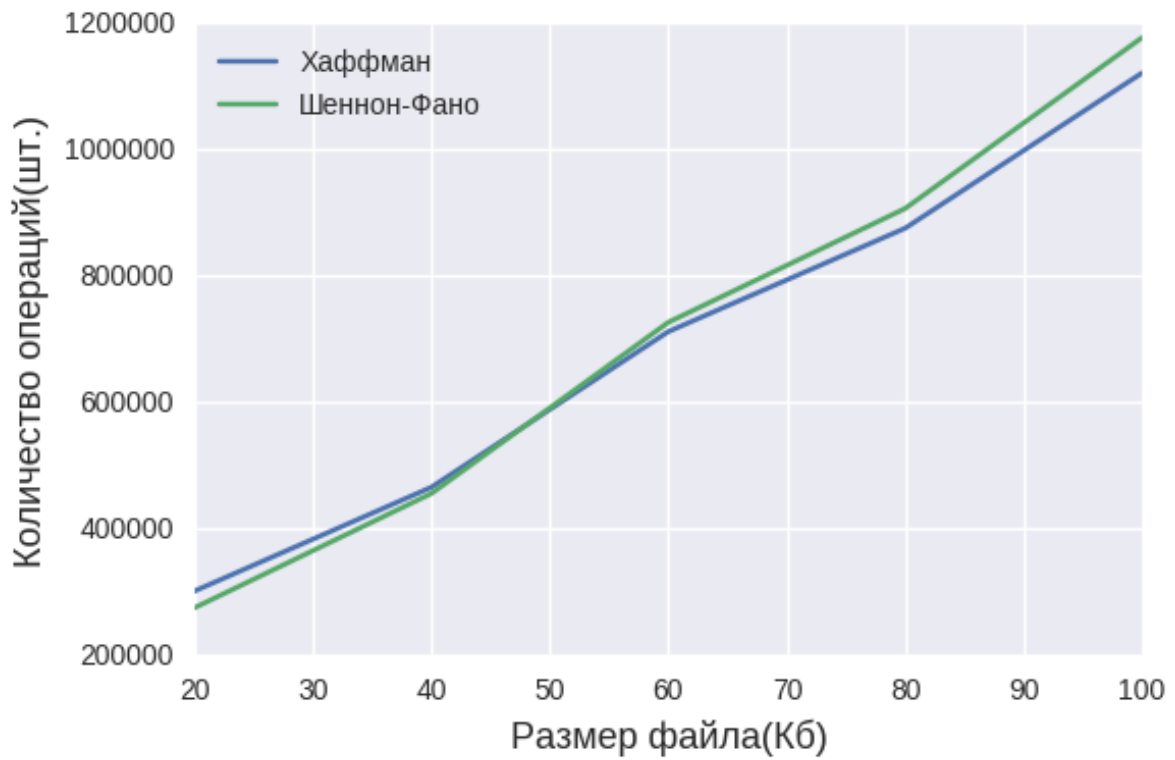
Второй набор, де-архивирование, малые размеры



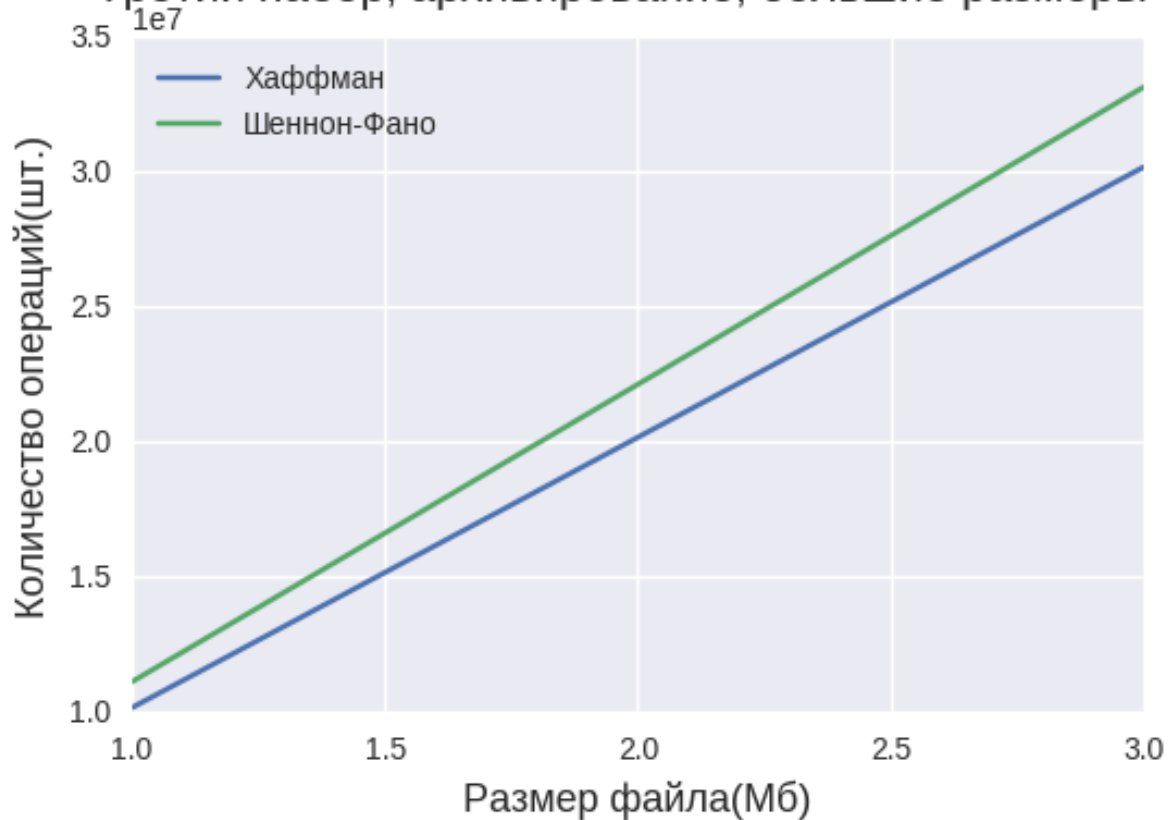
Второй набор, де-архивирование, большие размеры



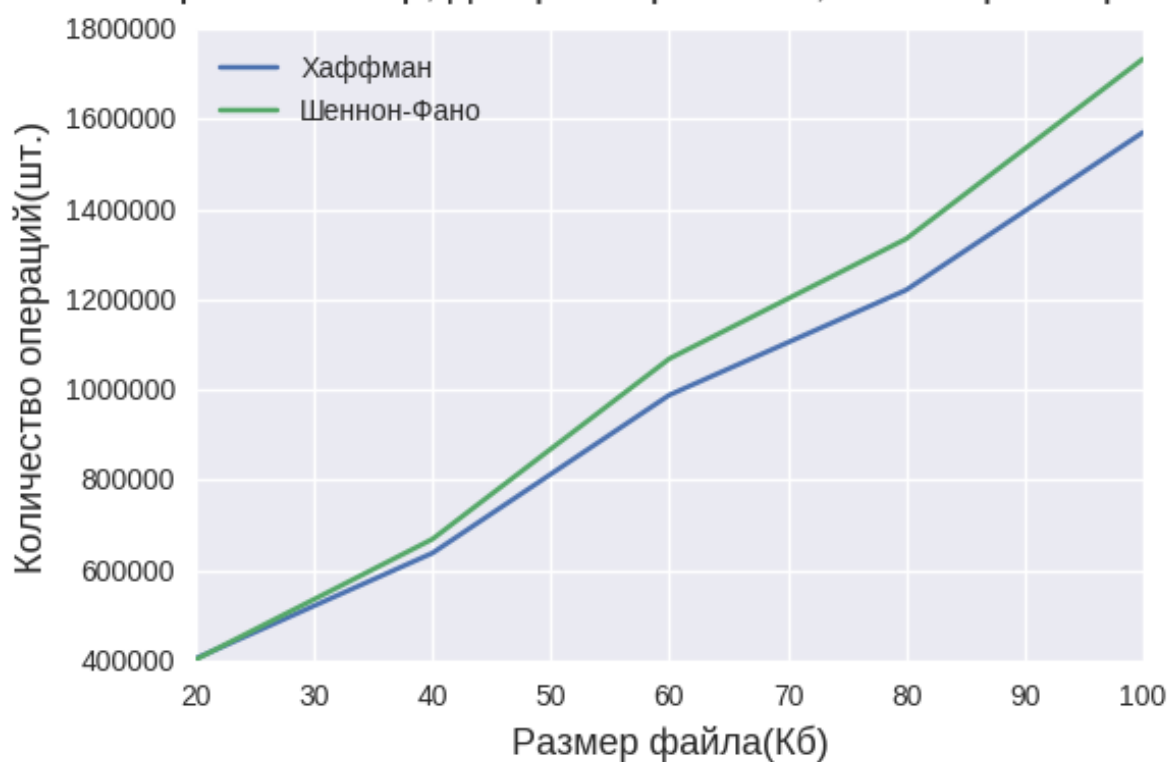
Третий набор, архивирование, малые размеры



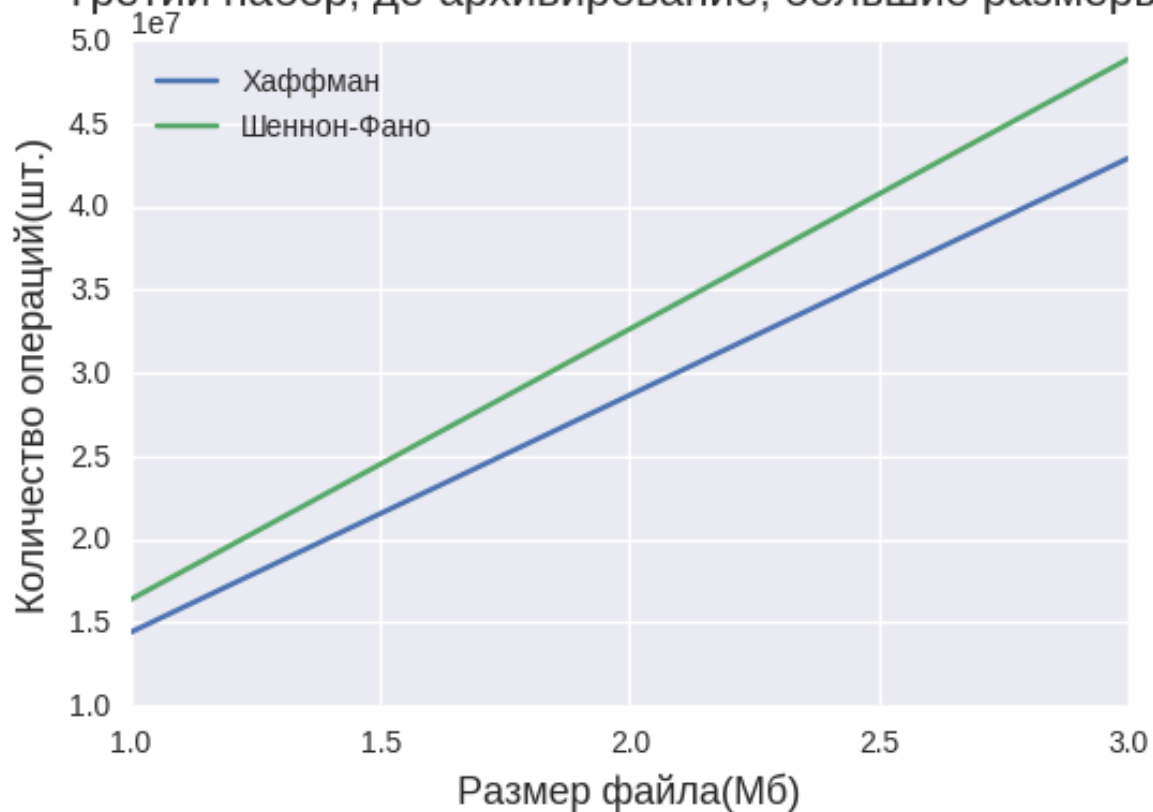
Третий набор, архивирование, большие размеры



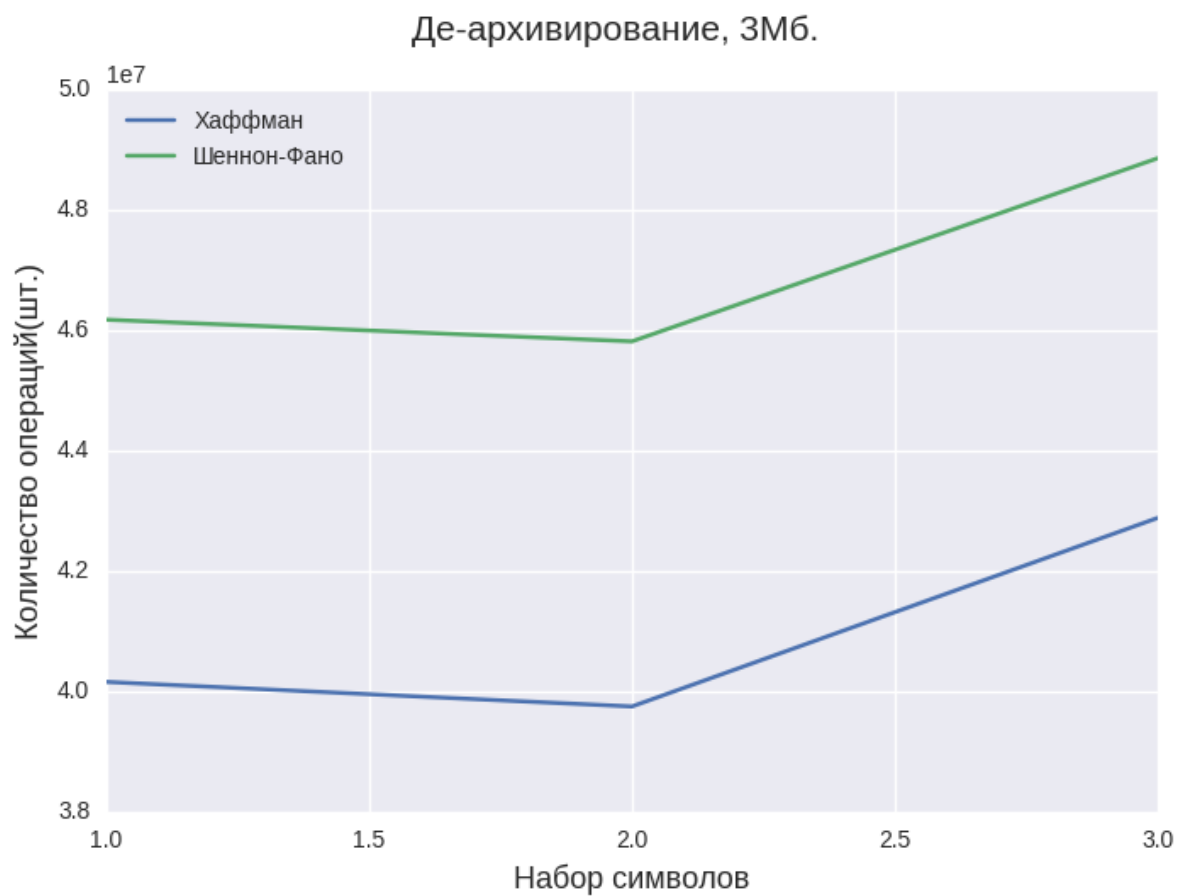
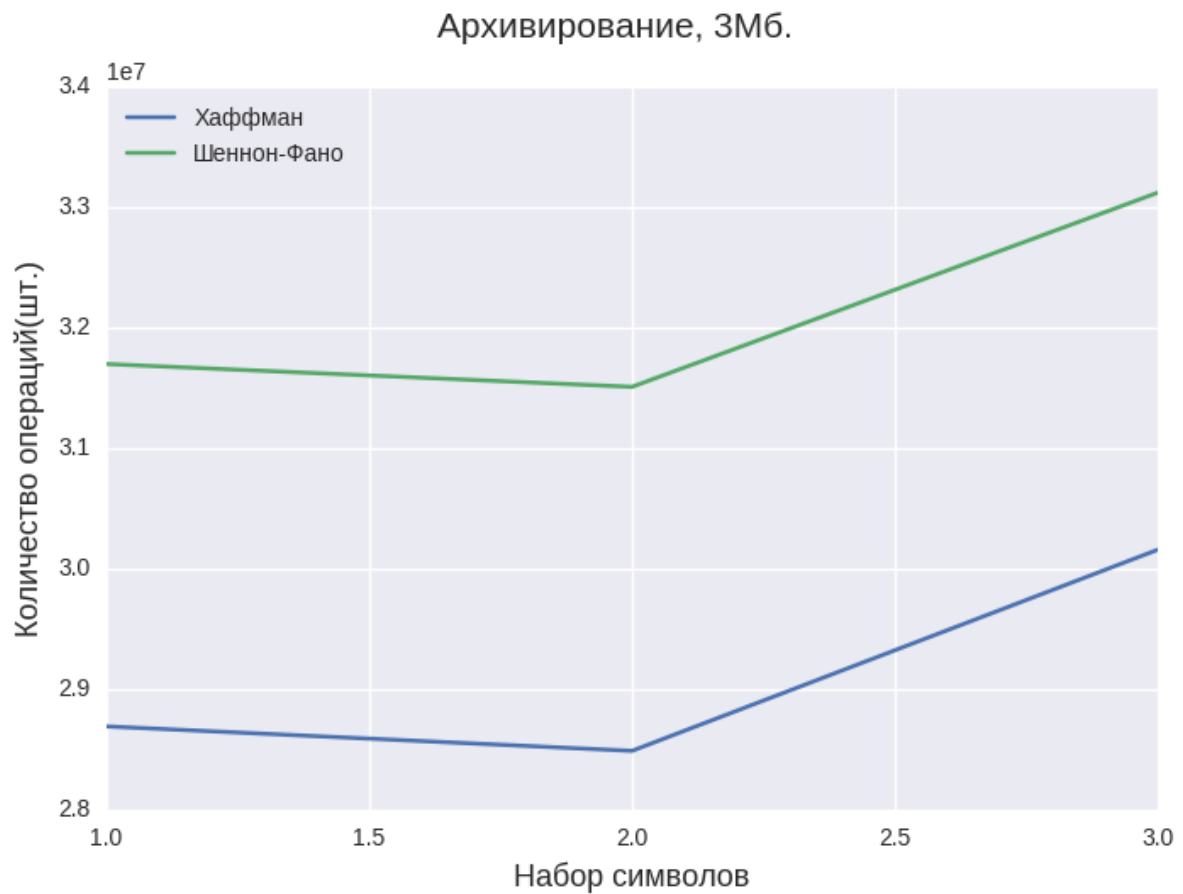
Третий набор, де-архивирование, малые размеры



Третий набор, де-архивирование, большие размеры



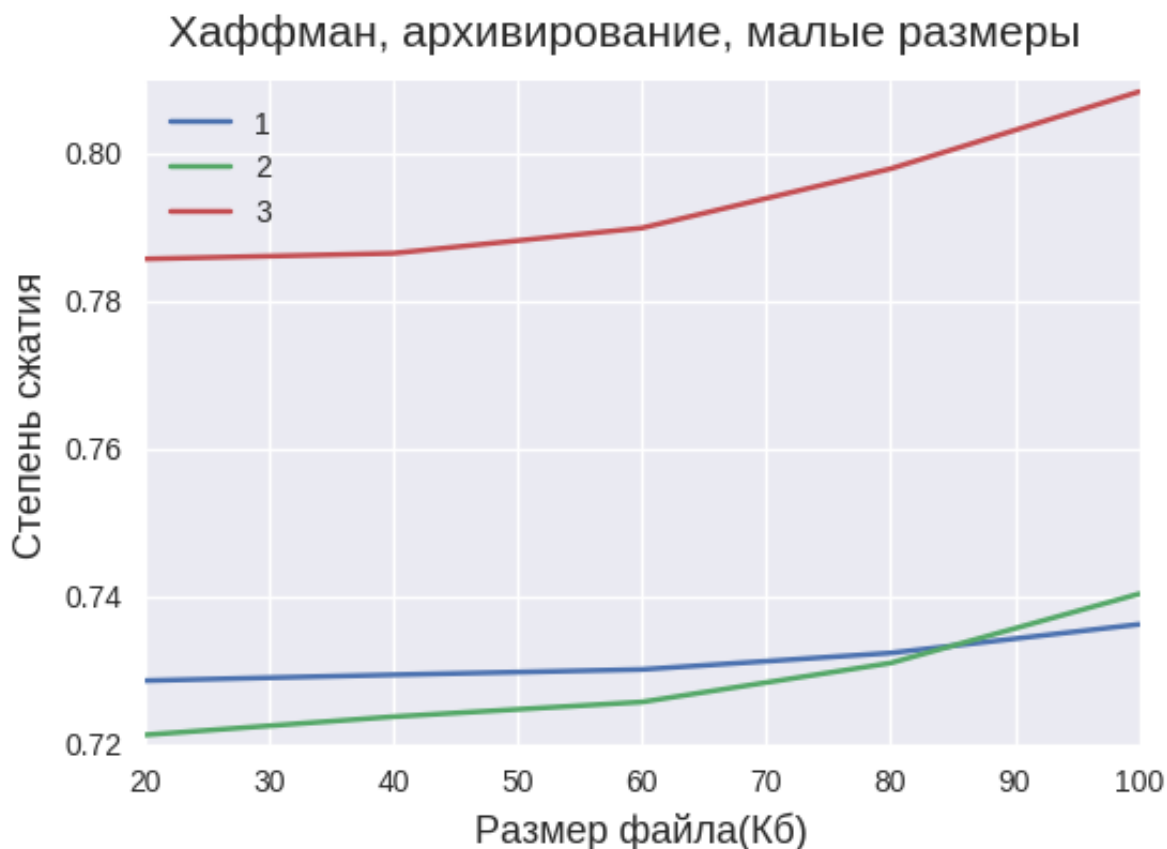
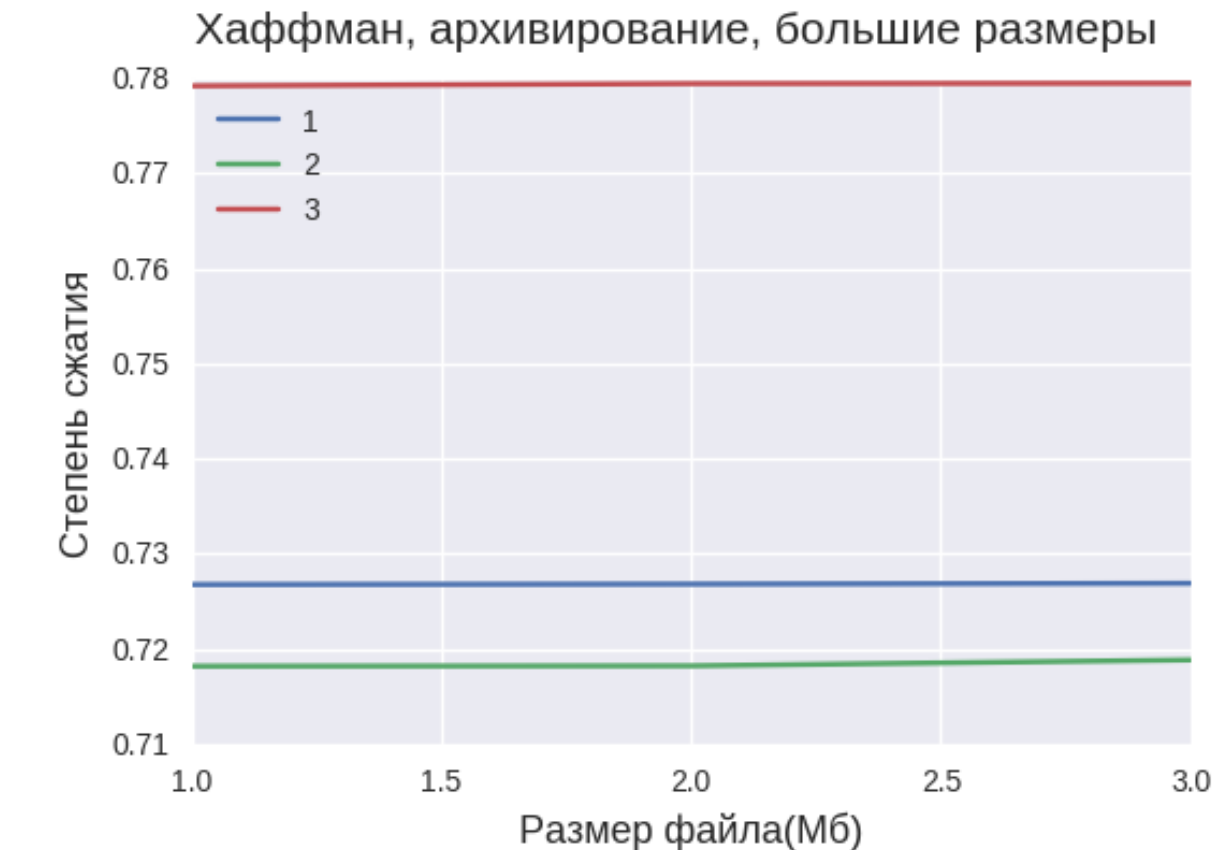
4.2.3 Третий набор графиков

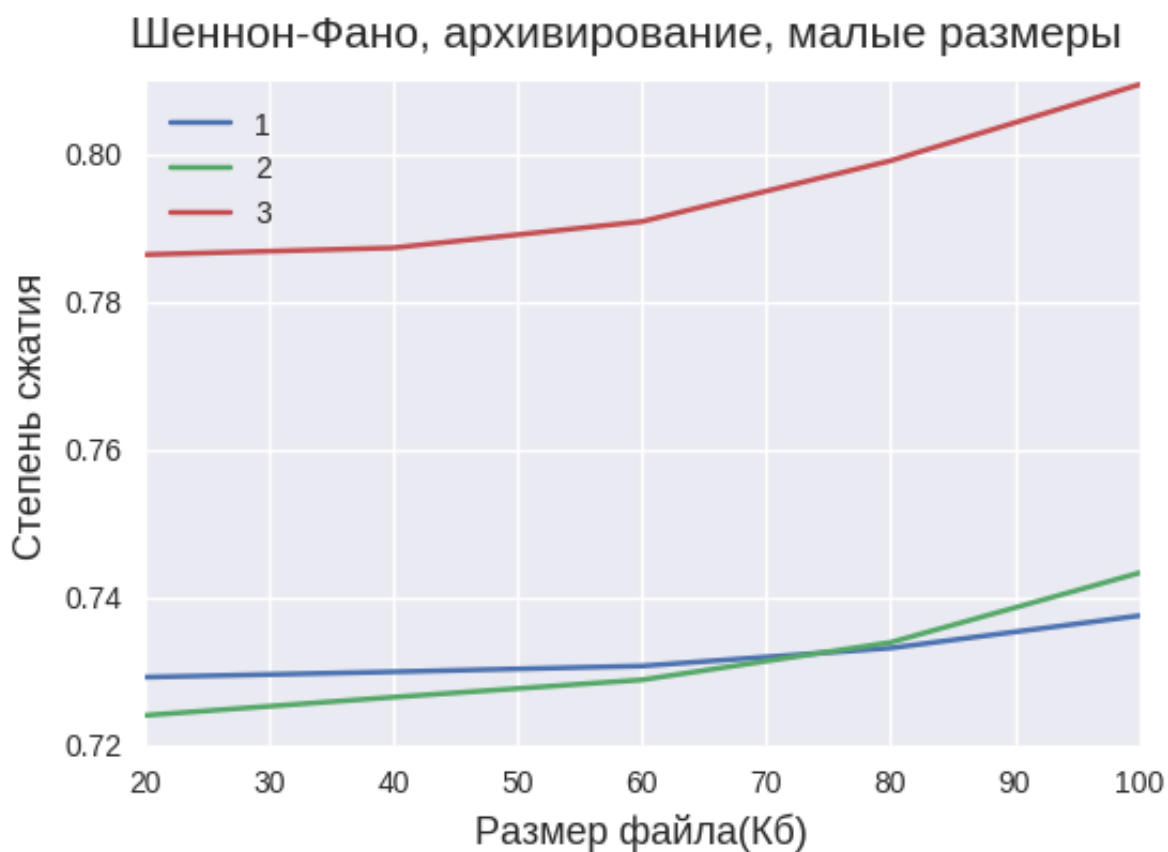
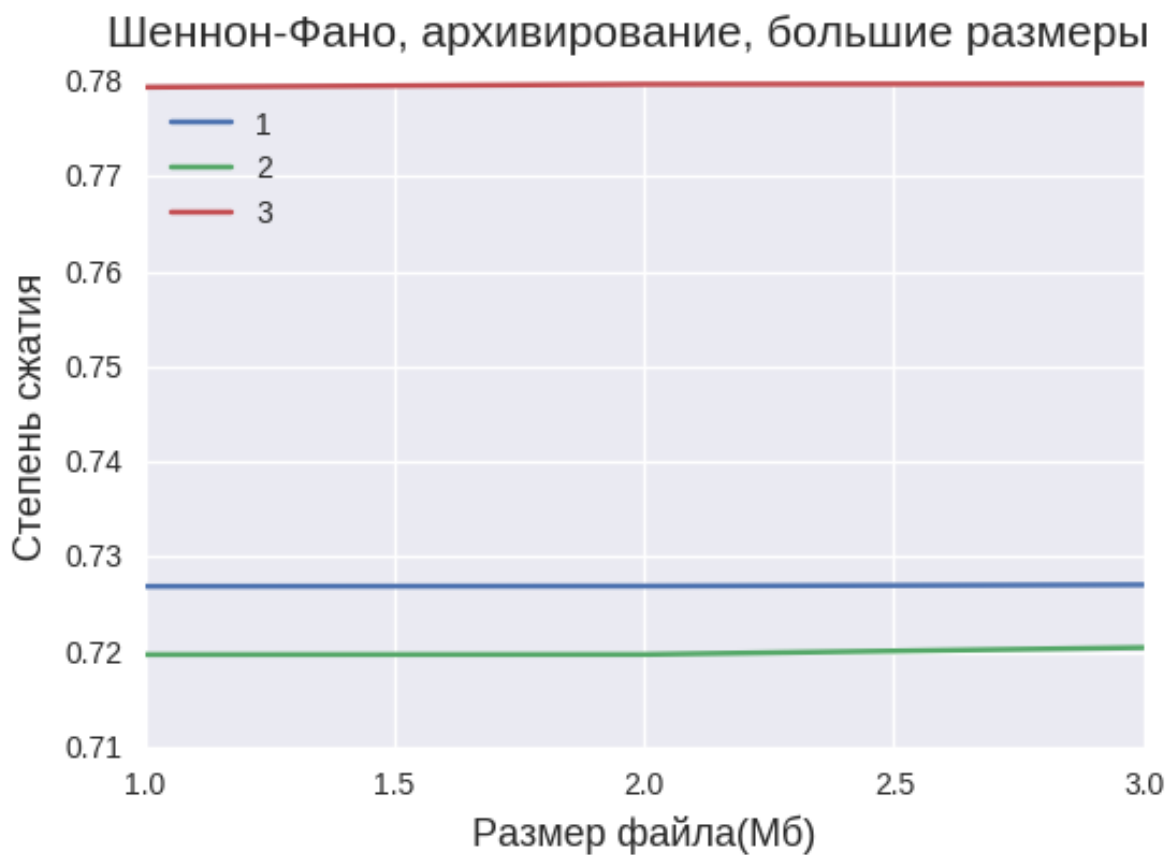


По приведенным графикам видно, что архивирование - разархивирование алгоритмом Хаффмана всегда занимает меньше операций, чем алгоритмом Шеннона-Фано. Еще видно, что третий набор символов (со знаками пунктуаций, скобками, и т.д.) занимает всегда больше операций, чем остальные, причем распределены они соответственно. То есть наименьшее кол-во операций требует архивация - деархивация файлов с английскими буквами, пробелами и новыми строками, затем тот же набор, но с русскими буквами, и на третьем месте с символами. Единственное исключение - на третьем наборе графиков, когда на втором наборе кол-во операций меньше, чем на первом.

4.3 Дополнительные графики

Здесь приведены дополнительные графики, а именно, графики, иллюстрирующие степень сжатия файлов в зависимости от используемого алгоритма, набора символов (цвет линии) и размера файлов.





Видно, что чем больше файл - тем больше степень сжатия. Так же, файлы с третьим набором символов имеют наибольшую степень сжатия. Интересно заметить, что при маленьких размерах файлов, файлы со вторым набором символов и размером до 100 Кб имели меньшую степень сжатия алгоритмом Хаффмана, чем файлы с первым набором символов. Аналогично с алгоритмом Шеннона-Фано, но только для файлов до 80 Кб.

5 Сравнительный анализ методов

Общее сравнение:

В моей реализации алгоритма Шеннона - Фано не использовалось дерево, в отличие от реализации алгоритма Хаффмана. Следовательно, сложность его выше. Если не смотреть конкретно на мою реализацию, то сложность будет примерно одинаковая. Оба алгоритма используют рекурсивные функции для построения дерева, которые работают за $O(n)$.

Сравнение двух алгоритмов:

Шеннон-Фано	Хаффман
Дерево строится сверху-вниз	Дерево строится снизу-вверх
Длина кодов в среднем большая	Длина кодов в среднем маленькая
Легко реализуется	Немного сложнее реализуется
Используемые СД: деревья, связные списки	Используемые СД: очереди с приоритетом, связные списки
Нигде не используется	Используется при сжатии JPEG и mp3 файлов

6 Заключение

В ходе работы над проектом была выполнена поставленная в начале задача. С использованием языка C++ была написана программа для архивации и деархивации текстовых файлов. Были сгенерированы тестовые текстовые файлы и проведены необходимые замеры. Составлена таблица, по ней построены и проанализированы графики. Приведу основные выводы, к которым я пришел:

- Степень и скорость сжатия лучше у алгоритма Хаффмана
- Алгоритм Хаффмана гарантирует построение кодов наименьшей длины и наименьшим средним числом символов на букву при данном распределении вероятностей, в то время как алгоритм Шеннона - Фано не всегда приводит к однозначному построению кода. Дело в том, что на первой итерации функции ShannonFano, большей по сумме вероятностей может оказаться как первая, так и вторая половина набора. В результате среднее число символов на букву окажется другим. Таким образом, построенный код может оказаться не самым лучшим
- Алгоритм Хаффмана имеет повсеместное применение, используется при сжатии изображений в формате JPEG/JPG, а так же для сжатия звука в формат mp3. Существует так же «адаптивный алгоритм Хаффмана», который, считается лучше в некоторых случаях. Он работает в реальном времени, адаптируясь под текущую вероятность символов в кодируемом файле
- Для очень маленьких файлов степень сжатия может быть > 1 , поскольку таблица частот, которая запишется в архив, может иметь такой размер, что в сумме с закодированным сообщением будет весить больше исходного текстового файла, не содержащего таблицу

7 Список используемых источников

Литературные источники:

[1] Кормен, Томас Х., Лейзерсон, Чарльз И., Ривест, Рональд Л., Штайн, Клиффорд. Алгоритмы: построение и анализ, 2-е издание.: Пер. с англ. М.: Издательский дом "Вильямс 2005. ISBN 5-8459-0857-4. стр. 219, стр. 459, стр. 216

Интернет - источники:

[2] Видеоурок "Алгоритм Хаффмана на C++":

<https://www.youtube.com/watch?v=KNVPFVG49Oc> Дата обращения: 27.11.2016

[3] Статья на википедии https://en.wikipedia.org/wiki/Huffman_coding. Дата обращения: 30.11.2012

[4] Статья на википедии <https://en.wikipedia.org/wiki/Shannon>