КДЗ-1 по дисциплине
“Алгоритмы и структуры данных”

Архивирование и разархивирование текстовых файлов с помощью алгоритмов Хаффмана и Шеннона — Фано

Выполнил Абрамов А. М.
Студент группы БПИ151

Table of Contents

Постановка задачи....2
Описание алгоритмов и использованных структур данных....3
Эксперимент....6
Результаты эксперимента – Графики....7
Результаты эксперимента - Таблицы....9
Анализ методов архивирования....17
Заключение....18
Инструменты использованные для разработки / отладки....19
Использованные источники....19

Постановка задачи

(1) Реализовать с использованием языка C++ программы для архивирования и разархивирования текстовых файлов. При этом использовать два известных алгоритма кодирования информации:

1.Хаффмана (простой)
2. Шеннона-Фано.

Обе реализации поместить в одном файле main.cpp, содержащем соответствующие методы:

1. Метод архивирования, использующий алгоритм Хаффмана,
вход: текстовый файл <name>.txt (кодировка UTF-8)
выход: архивированный файл <name>.haff

2. Метод разархивирования, использующий алгоритм Хаффмана,
вход: архивированный файл <name>.haff
выход: разархивированный файл <name>-unz-h.txt (кодировка UTF-8)

3. Метод архивирования, использующий алгоритм Шеннона-Фано,
вход: текстовый файл <name>.txt (кодировка UTF-8)
выход: архивированный файл <name>.shan

4. Метод разархивирования, использующий алгоритм Шеннона-Фано.
вход: архивированный файл <name>.shan
выход: разархивированный файл <name>-unz-s.txt (кодировка UTF-8)

Выбор алгоритма осуществляется с помощью флага командной строки.

(2) Провести вычислительный эксперимент с целью оценки реализованных алгоритмов архивации / разархивации. Измерить (экспериментально) количество операций (в рамках модели RAM), выполняемых за время работы (архивирования, разархивирования) каждого алгоритма на

файле для каждого размера входного файла и набора символов. Для повышения достоверности результатов каждый эксперимент повторить 5 раз на различных файлах (с одним возможным набором символов и одного размера) с последующим усреднением результата.

Для этого следует подготовить тестовый набор из нескольких текстовых файлов разного объёма (20, 40, 60, 80, 100 Кб; 1, 2, 3 Мб — всего 8 файлов) на разных языках (ru, en - кодировка UTF-8) с разным набором символов в каждом файле, а именно:

0. набор номер 0: символы латинского алфавита и пробел
1. набор номер 1: символы из первого набора + символы русского алфавита
2. набор номер 2: символы из второго набора + следующие знаки и спецсимволы: знаки арифметики [+ - * / =], знаки препинания [. , ; : ? !], символы [% @ # $ & ~] скобки разных типов [( ) [ ] { } < >] , кавычки [" ']),

(3) Подготовить отчёт по итогам работы, содержащий постановку задачи, описание алгоритмов и задействованных структур данных, описание реализации, обобщённые результаты измерения эффективности алгоритмов, описание использованных инструментов (например, если использовались скрипты автоматизации), выводы о соответствии результатов экспериментальной проверки с теоретическими оценками эффективности исследуемых алгоритмов.

Проделанная работа
В рамках выполнения домашнего здания полностью выполнены все выше перечисленные пункты.

Описание алгоритмов и использованных структур данных

Описание работы программы
Для обоих алгоритмов программа работает в два прохода. Сначала строится таблица частот встречаемости символов в конкретном файле. Затем разными способами строится кодовое дерево. Кодовое дерево записывается в выходящий файл. В выходящий файл записывается именно дерево а не таблица перевода символов, так как дерево занимает гораздо меньше места для любого файла состоящего из более чем двух видов символов. По кодовому дереву строится таблиц перевода символов в их битовые коды. Далее осуществляется проход по входящему файлу и каждый прочитанный символ переводиться в битовый код и записывается в выходной файл.

Для разархивирования алгоритм считывает кодовое дерево записанное в начале файла, и одновременно со считыванием строит таблицу перевода символов. Далее осуществляется проход по входящему (архивированному) файлу и как только прочитанные биты образуют код соответствующей записи в таблице перевода символов, по битам находиться изначальный символ который и записывается в выходной (разархивированный) файл.

Построение кодового дерева
Код Шеннона-Фано строится с помощью дерева. Построение этого дерева начинается от корня. Всё множество кодируемых элементов соответствует корню дерева (вершине первого уровня).

Оно разбивается на два подмножества с примерно одинаковыми суммарными вероятностями. Эти подмножества соответствуют двум вершинам второго уровня, которые соединяются с корнем. Далее каждое из этих подмножеств разбивается на два подмножества с примерно одинаковыми суммарными вероятностями. Им соответствуют вершины третьего уровня. Такое последовательное разбиение дерева описывается с помощью рекурсии. Если подмножество содержит единственный элемент, то ему соответствует концевая вершина кодового дерева. Подобным образом поступаем до тех пор, пока не получим все концевые вершины. Ветви кодового дерева размечаем символами 1 и 0.

Код Хаффмана также строится с помощью дерева, как и в случае кода Шеннона. Сначала по каждому входному символу строятся листья дерева. Каждый лист имеет вес, который равен количеству вхождений символа в сообщение, которое необходимо сжать. Далее:

1. Выбираются два свободных узла дерева с наименьшими весами.
2. Создаётся их родитель с весом, равным их суммарному весу.
3. Родитель добавляется в список свободных узлов, а два его потомка удаляются из этого списка.
4. Одной дуге, выходящей из родителя, ставится в соответствие бит 1, другой - бит 0.
5. Шаги, начиная со второго, повторяются до тех пор, пока в списке свободных узлов не останется только один свободный узел. Он и будет считаться корнем дерева.

Краткое описание созданных классов / структур данных под алгоритмы
Кодовое дерево описывается с помощью его узлов. Для описания узлов используется класс INode.

От класса Inode наследуются классы LeafNode & InternalNode. Класс LeafNode используется для представления листа кодового дерева (символ и его частота). Класс InternalNode используется для хранения информации об узлах кодового дерева (левый и правый потомки, и их суммарная частота).

В базовом классе IEncoder собранны основные функции необходимые для архифирования, которые описаны в пункте “Описание работы программы”.

От класса IEncoder наследуются классы EncodeHuffman & EncodeShannon реализуюшие каждый свою вариацию функции для построения кодового дерева.

Краткое описание вспомогательных созданных классов / структур данных
Класс NodeCmp реализует оператор сравнения для узлов дерева и используется для сортировки вектора из узлов дерева.

Есть четыре класса для чтения/записи в файл. Классы bit_ifstream & bit_ofstream написаны по образу классов basic_ifstream & basic_ofstream из библиотеки STL. Однако они работают на уровне по-битового чтения / записи. Классы ucs4_ifstream & ucs4_ofstream также наследуют от стандартных файловых потоков, они нужны для автоматизации перевода UTF-8 в UCS4 и обратно при чтении и записи соответственно.

Класс ArgParse написан для работы с аргументами командной строки.

Более подробное описание классов для реализации алгоритмов
Для описания кодового дерева используется класс Inode. Этот класс имеет одно поле для записи частоты встречаемости символа. Также в нем реализован оператор сравнения для сортировки массива из узлов кодового дерева.

От него наследуются классы LeafNode & InternalNode. Класс LeafNode содержит поля для символа и частоты его встречаемости. Это лист кодового дерева. Класс InternalNode используется как узел кодового дерева. Он содержит своего левого и правого потомка (ветви дерева) и их суммарную частоту. Таким образом он характеризует частоту некоторого множества символов. Соответственно корень дерева содержит частоту всех символов (то есть общее количество символов в архивируемом файле).

Создан базовый класс IEncoder в котором собранны основные функции необходимые для архивирования. В нем содержаться функции для построй таблицы частот находящихся в файле символов. Функции принимающая кодовое дерево и строящее по нему таблицу перевода символов. Таблица перевода символов сопоставляет каждому входному символу его битовый код. Ещё в классе содержится функция для побитовой записи кодового дерева в файл (что нужно для того чтобы восстановить таблицу перевода символов при разархивирования). Далее в нем реализована функция для поочерёдного перевода всех входных символов из входящего текстового потока в их битовый код (с помощью таблицы перевода символов) и записи в “исходящий” битовый поток. В нем же содержится и собственно сама функция Encode, которая принимает текстовый и байтовый потоки и осуществляет архивацию.

На основе Iencoder создано два класса EncoderHuffman & EncoderShannon. Эти классы реализуют виртуальную функцию BuildTree заданную в IEncoder, так как отличие между двумя алгоритма состоит в способе постройке дерева кодов.

EncoderHuffman состоит всего лишь из одной этой функции, так как алгоритм Хаффмана достаточно прост и не требует вспомогательных методов/полей.

EncoderShannon состоит из трёх функций. Функция BuildTree запускает рекурсивное построение дерева, функция InnerBuildTree является рекурсивной функцией которая запускается на каждом подразбиении вектора частот. Функция FindBreakingIndex ищет следующий индекс для разбиения вектора частот с учётом того, что для построения наиболее эффективного кодового дерева надо минимизировать разницу в частотах между левыми и правыми ветками кодового дерева.

Для разархивирования был создан класс Decoder. Не было необходимости создавать отдельные реализации для разархиватора отдельно Хаффмана и отдельно Шеннона, в силу того что оба эти алгоритма используют одну и ту же структуру кодового дерева для указания того как кодировать / раскодировать каждый символ. Поэтому в классе Decoder реализованна одна функция для чтения кодового дерева которое одинаково записывается в файл при любом типе архивации.

Более подробно о вспомогательных классах
Класс NodeCmp реализует оператор сравнения для узлов дерева и используется для сортировки вектора из узлов дерева.

Есть четыре класса для чтения/записи в файл. Классы bit_ifstream & bit_ofstream написаны по образу классов basic_ifstream & basic_ofstream из библиотеки STL. Однако они работают на уровне по-битового чтения / записи. Запись и чтение производятся через байтовый буфер.

Когда используется побитовая запись в файл, может случиться что последний байт файла будет использован не полностью и часть его будет содержать мусор который может повлиять на разархивацию. Для решения этой проблемы, после архивации файла, в его первый байт записывается количество незадействованных битов последнего файла. Таким образом bit_ifstream всегда знает когда ему стоит перестать считывать последний файл и выставить флаг EOF. Альтернативой такому способу является добавление так называемого псевдо-EOF символа в входной поток на стадии архивации. Минусам такого подхода является то, что такой символ должен наравне с обычными символами занимать место в кодовом дереве и более того он может совпасть с одним из символов в файле. Так как я посчитал нужным поддерживать набор символов больше чем заданный в рамках данного домашнего задания, (программа обрабатывает все символы внутри unicode-BMP а также символы вне её [которые не влезают в 32-битный wchar_t предоставляемый в операционной среде Windows]) поэтому вариант с добавлением особого символа для псевдо-EOF не подходил.

При записи и чтении из архивированных файлов, символы для кодового дерева содержаться в формате utf-8 и после из побитового извлечения преобразуются в широкие символы. Таким образом в классе bit_ifstream пришлось реализовать мини парсер для того чтобы определять длину в байтах для символа записанного с помощью utf-8.

В классе bit_ofstream также присутствует код для перевода char32_t в utf-8 перед записью в архивный файл. Проблема с прямой записью 4 битных типов наподобие char32_t в файл напрямую, заключается в наличии архитектур разных типов: Little-endian и Big-endian. Для типа состоящего из нескольких байтов (как например char32_t) при прямой записи его в файл на одной архитектуре и при прямом чтении его из файла на другой архитектуре будет прочитан совершенно другой символ (так как байты из файла после загрузки будут расположены в обратном порядке). Поэтому для обеспечения переносимости архивного файла между разными системами необходимо использовать кодировку utf-8 где есть ведущий байт и следующие байты.

Классы ucs4_ifstream & ucs4_ofstream также наследуют от стандартных файловых потоков, они нужны для автоматизации перевода UTF-8 в UCS4 и обратно при чтении и записи соответственно. Перевод из char32_t в utf-8 и обратно осуществляется стандартными средствами предоставляемыми библиотекой STL (в частности функцией std::basic_ifstream::imbue() и классами std::locale и std::codecvt_utf8)

Класс ArgParse написан для работы с аргументами командной строки. Было непонятно если можно подключать сторонние библиотеки, поэтому я не смог воспользоваться уже отлаженными библиотеками как GetOpt или Boost в качестве парсера для опций командной строки.

Эксперимент

Описание плана эксперимента
Заводиться глобальная переменная для подсчёта количества операций: экземпляр класса OpsCounter. Далее в код программы вставляются счётчики для подсчёта количества операций.

С помощью скрипта программа запускается на файле и после окончания работы создаёт файл с именем исходного, но с добавлением постфикса “.ops”. В данный файл записывается количество подчитанных операций.

Набор файлов для эксперимента также создаётся скриптом. Необходимо взять файлы размеров 20, 40, 60, 80, 100 Кб; 1, 2, 3 Мб - всего 8 размеров, каждому размеру файла следует сопоставить 3 набора для выбора символов. Для каждого размера и набора символов создаётся 5 файлов. Символы из набора встречаются в каждом из десяти файлов в случайном порядке. При архивации каждого из десяти файлов собираются данные о количестве операций и в конце они усредняются. При разархивации подсчёт операций происходит так же, через прогон программы на 10 файлах для каждого фиксированного размера и фиксированного набора символов и через усреднение результата.

Результаты эксперимента – Графики

Отчёт содержит следующие графики (всего 8 + 12 + 2):

1. Зависимость количества операций (ось OY) от размера файла (ось OX) и размера набора символов (цвет линии) для каждого алгоритма архивирования / разархивирования. Должно быть 2 (алгоритма) * 2 (арх/разарх) * 2 (малые и большие файлы отдельно) набора графиков, на каждом 3 кривые (для каждого набора символов свой цвет).

2. Зависимость количества операций (ось OY) от размера файла (ось OX), и используемого алгоритма (цвет линии) для каждого набора символов. Должно быть 3 (набора символов) * 2 (арх/разарх) * 2 (малые и большие файлы отдельно) набора графиков, на каждом 2 кривые (для каждого алгоритма).

3. Зависимость количества операций (ось OY) от размера набора символов (ось OX) 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).

Результаты эксперимента - Таблицы
Графики построены по следующим данным. Данные сложно привести в таблицу так как на каждом графике присутствуют несколько переменных, поэтому они приведены в том виде в каком они находились непосредственно перед передачей в функцию для отрисовки графика.

Данные для графика: Архивирование алгоритм haff большие файлы
Компонента X: Размер в байтах
Компонента У: Количество операций

```
{0: [(1048576, 35156426),
     (2097152, 70310179),
     (3145728, 105464702)],
 1: [(1048576, 41389893),
     (2097152, 82762313),
     (3145728, 124132122)],
 2: [(1048576, 43338407),
     (2097152, 86677712),
     (3145728, 130009660)]}
```

Данные для графика: Архивирование алгоритм haff небольшие файлы
Компонента X: Размер в байтах
Компонента У: Количество операций

```
{0: [(20480, 692655),
     (40960, 1378754),
     (61440, 2065506),
     (81920, 2751213),
     (102400, 3436133)],
 1: [(20480, 831519),
     (40960, 1638747),
     (61440, 2446641),
     (81920, 3255345),
     (102400, 4062936)],
 2: [(20480, 870987),
     (40960, 1715359),
     (61440, 2561173),
     (81920, 3407187),
     (102400, 4251587)]}
```

Данные для графика: Архивирование алгоритм shan большие файлы
Компонента X: Размер в байтах
Компонента У: Количество операций

```
{0: [(1048576, 35155640),
     (2097152, 70307837),
     (3145728, 105460364)],
 1: [(1048576, 41377479),
     (2097152, 82741537),
     (3145728, 124103478)],
 2: [(1048576, 43348568),
     (2097152, 86685870),
     (3145728, 130020071)]}
```

Данные для графика: Архивирование алгоритм shan небольшие файлы
Компонента X: Размер в байтах
Компонента У: Количество операций

```
{0: [(20480, 694697),
     (40960, 1380772),
     (61440, 2066743),
     (81920, 2752887),
     (102400, 3437959)],
 1: [(20480, 838107),
     (40960, 1642014),
     (61440, 2449674),
     (81920, 3257548),
     (102400, 4064831)],
 2: [(20480, 879874),
     (40960, 1725633),
     (61440, 2570773),
     (81920, 3416908),
     (102400, 4261546)]}
```

Данные для графика: Архивирование набор символов 0 большие файлы
Компонента X: Размер в байтах
Компонента У: Количество операций

```
{'haff': [(1048576, 35156426),
          (2097152, 70310179),
          (3145728, 105464702)],
 'shan': [(1048576, 35155640),
          (2097152, 70307837),
          (3145728, 105460364)]}
```

Данные для графика: Архивирование набор символов 0 небольшие файлы
Компонента X: Размер в байтах
Компонента У: Количество операций

{'haff': [(20480, 692655),
(40960, 1378754),
(61440, 2065506),
(81920, 2751213),
(102400, 3436133)],
'shan': [(20480, 694697),
(40960, 1380772),
(61440, 2066743),
(81920, 2752887),
(102400, 3437959)]}

Данные для графика: Архивирование набор символов 1 большие файлы
Компонента X: Размер в байтах
Компонента У: Количество операций

{'haff': [(1048576, 41389893),
(2097152, 82762313),
(3145728, 124132122)],
'shan': [(1048576, 41377479),
(2097152, 82741537),
(3145728, 124103478)]}

Данные для графика: Архивирование набор символов 1 небольшие файлы
Компонента X: Размер в байтах
Компонента У: Количество операций

{'haff': [(20480, 831519),
(40960, 1638747),
(61440, 2446641),
(81920, 3255345),
(102400, 4062936)],
'shan': [(20480, 838107),
(40960, 1642014),
(61440, 2449674),
(81920, 3257548),
(102400, 4064831)]}

Данные для графика: Архивирование набор символов 2 большие файлы
Компонента X: Размер в байтах
Компонента У: Количество операций

{'haff': [(1048576, 43338407),
(2097152, 86677712),
(3145728, 130009660)],
'shan': [(1048576, 43348568),
(2097152, 86685870),
(3145728, 130020071)]}

Данные для графика: Архивирование набор символов 2 небольшие файлы
Компонента X: Размер в байтах
Компонента У: Количество операций

{'haff': [(20480, 870987),
(40960, 1715359),
(61440, 2561173),
(81920, 3407187),
(102400, 4251587)],
'shan': [(20480, 879874),
(40960, 1725633),
(61440, 2570773),
(81920, 3416908),
(102400, 4261546)]}

Данные для графика: Архивирование размер файла 1048576 байт
Компонента X: Набор символов
Компонента У: Количество операций

{'haff': [(0, 35156426),
(1, 41389893),
(2, 43338407)],
'shan': [(0, 35155640),
(1, 41377479),
(2, 43348568)]}

Данные для графика: Архивирование размер файла 2097152 байт
Компонента X: Набор символов
Компонента У: Количество операций

{'haff': [(0, 70310179),

(1, 82762313),
(2, 86677712)],
'shan': [(0, 70307837),
(1, 82741537),
(2, 86685870)]}

Данные для графика: Архивирование размер файла 3145728 байт
Компонента X: Набор символов
Компонента У: Количество операций

{'haff': [(0, 105464702),
(1, 124132122),
(2, 130009660)],
'shan': [(0, 105460364),
(1, 124103478),
(2, 130020071)]}

Данные для графика: Разархивирование алгоритм haff большие файлы
Компонента X: Размер в байтах
Компонента У: Количество операций

{0: [(1048576, 54006411),
(2097152, 108013293),
(3145728, 162020022)],
1: [(1048576, 63964208),
(2097152, 127916468),
(3145728, 191865792)],
2: [(1048576, 66773449),
(2097152, 133552864),
(3145728, 200325119)]}

Данные для графика: Разархивирование алгоритм haff небольшие файлы
Компонента X: Размер в байтах
Компонента У: Количество операций

{0: [(20480, 1059915),
(40960, 2114066),
(61440, 3168742),
(81920, 4222519),
(102400, 5275468)],
1: [(20480, 1272093),
(40960, 2519595),
(61440, 3768381),

(81920, 5017668),
(102400, 6265859)],
2: [(20480, 1328363),
(40960, 2630087),
(61440, 3933353),
(81920, 5236886),
(102400, 6538764)]}

Данные для графика: Разархивирование алгоритм shan большие файлы
Компонента X: Размер в байтах
Компонента У: Количество операций

{0: [(1048576, 54017036),
(2097152, 108029456),
(3145728, 162040602)],
1: [(1048576, 63975098),
(2097152, 127929265),
(3145728, 191882207)],
2: [(1048576, 66805609),
(2097152, 133589636),
(3145728, 200372760)]}

Данные для графика: Разархивирование алгоритм shan небольшие файлы
Компонента X: Размер в байтах
Компонента У: Количество операций

{0: [(20480, 1061602),
(40960, 2116557),
(61440, 3171272),
(81920, 4226125),
(102400, 5280450)],
1: [(20480, 1273674),
(40960, 2521455),
(61440, 3770340),
(81920, 5019678),
(102400, 6268662)],
2: [(20480, 1330448),
(40960, 2634001),
(61440, 3938425),
(81920, 5243255),
(102400, 6546938)]}

Данные для графика: Разархивирование набор символов 0 большие файлы

Компонента X: Размер в байтах
Компонента У: Количество операций

{'haff': [(1048576, 66773449),
(2097152, 133552864),
(3145728, 200325119)],
'shan': [(1048576, 66805609),
(2097152, 133589636),
(3145728, 200372760)]}

Данные для графика: Разархивирование набор символов 0 небольшие файлы
Компонента X: Размер в байтах
Компонента У: Количество операций

{'haff': [(20480, 1328363),
(40960, 2630087),
(61440, 3933353),
(81920, 5236886),
(102400, 6538764)],
'shan': [(20480, 1330448),
(40960, 2634001),
(61440, 3938425),
(81920, 5243255),
(102400, 6546938)]}

Данные для графика: Разархивирование набор символов 1 большие файлы
Компонента X: Размер в байтах
Компонента У: Количество операций

{'haff': [(1048576, 66773449),
(2097152, 133552864),
(3145728, 200325119)],
'shan': [(1048576, 66805609),
(2097152, 133589636),
(3145728, 200372760)]}

Данные для графика: Разархивирование набор символов 1 небольшие файлы
Компонента X: Размер в байтах
Компонента У: Количество операций

{'haff': [(20480, 1328363),
(40960, 2630087),
(61440, 3933353),
(81920, 5236886),

(102400, 6538764)],
'shan': [(20480, 1330448),
(40960, 2634001),
(61440, 3938425),
(81920, 5243255),
(102400, 6546938)]}

Данные для графика: Разархивирование набор символов 2 большие файлы
Компонента X: Размер в байтах
Компонента У: Количество операций

{'haff': [(1048576, 66773449),
(2097152, 133552864),
(3145728, 200325119)],
'shan': [(1048576, 66805609),
(2097152, 133589636),
(3145728, 200372760)]}

Данные для графика: Разархивирование набор символов 2 небольшие файлы
Компонента X: Размер в байтах
Компонента У: Количество операций

{'haff': [(20480, 1328363),
(40960, 2630087),
(61440, 3933353),
(81920, 5236886),
(102400, 6538764)],
'shan': [(20480, 1330448),
(40960, 2634001),
(61440, 3938425),
(81920, 5243255),
(102400, 6546938)]}

Данные для графика: Разархивирование размер файла 1048576 байт
Компонента X: Набор символов
Компонента У: Количество операций

{'haff': [(0, 54006411),
(1, 63964208),
(2, 66773449)],
'shan': [(0, 54017036),
(1, 63975098),
(2, 66805609)]}

Данные для графика: Разархивирование размер файла 2097152 байт
Компонента X: Набор символов
Компонента У: Количество операций

{'haff': [(0, 108013293),
(1, 127916468),
(2, 133552864)],
'shan': [(0, 108029456),
(1, 127929265),
(2, 133589636)]}

Данные для графика: Разархивирование размер файла 3145728 байт
Компонента X: Набор символов
Компонента У: Количество операций

{'haff': [(0, 162020022),
(1, 191865792),
(2, 200325119)],
'shan': [(0, 162040602),
(1, 191882207),
(2, 200372760)]}

Анализ методов архивирования

Графики подтверждают что сложность обоих алгоритмов линейная. Это видно по тому что с увеличением размера при архивации и разархивации файлов график зависимости операций от размера входного файла сохраняет форму прямой линии.

По поводу количества операций для каждого алгоритма то по сравнению с алгоритмом Шеннона, алгоритм Хаффмана требует столько же или меньших затрат на вычисление. Основываясь на исходных кодах данной реализации можно сказать что во-первых для построения кодового дерева Хаффмана требуется меньше элементарных операций чем для построения дерева Шеннона, во-вторых кодирование Шеннона может приводить к постройке не самого оптимального кодового дерева, из за которого увеличиться количество операций побитового чтения / записи в файл.

Построение не самого оптимального кодового дерева алгоритмом Шеннона связано с тем что разбиение множества элементов производиться не строго (индекс разбиения находиться через минимизацию функции разницы в частоте между правой и левой веткой кодового дерева). Оно может бы произведено несколькими способами. Выбор разбиения на уровне n может ухудшить варианты разбиения на следующем уровне и привести к неоптимальности кода в целом. Другими словами, оптимальное поведение на каждом шаге пути ещё не гарантирует оптимальности всей совокупности действий.

При архивации в проведённых экспериментах разница в количестве операций между алгоритмами не особо заметна. Она равна примерно десяти тысячам операций для файлов размером меньше 1 Мб. Алгоритм Хаффмана довольно существенно выигрывает по количеству операций при постройки кодового дерева, особенно для файлов со набором символов номер 2 (включающем все символы).

При разархивировании на количество операций влияет только степень оптимальности построенного кодового дерева. Поэтому в данном эксперименте для разархивирования разница в количестве операций между двумя алгоритмами еще меньше чем при архивировании в силу того, что алгоритм Шеннона на данных входных файлах строит кодовое дерево очень близкое к оптимальному.

С моей точки зрения нахождение алгоритмом Шеннона кодового дерева близкого к оптимальному связано с тем что при создании файлов для тестирования работы программы, символы из наборов брались в случайном порядке. Из-за этого у каждого символа примерно одинаковая частота встречаемости, и конечно чем больше файл тем более одинаковая у все символов частота. Такие условия позволяют алгоритму Шеннона находить оптимальное, либо очень близкое к оптимальному кодирование. Это подтверждается приведёнными выше графиками и сравнением размеров архивных файлов. В данном эксперименте длина последовательности, полученной сжатием по методу Шеннона, очень близка к длине сжатой последовательности с использованием кодирования Хаффмана.

Так например для файла в 3 Мб с набором символов номер 2 (все возможные символы) разница в размере архивных файлов составляет лишь 711 байт. На маленьких файлах размера 20Кб наблюдается такое же явление. А именно разница в архивных файлах для входных данных со всеми символами составляет 31 байт.

Для того чтобы сделать эксперимент более показательным мне кажется стоило протестировать работу программы и обоих алгоритмов на файлах в которых частота встречающихся символов значительно отличается.

Также стоит отметить что для обоих алгоритмов, при любых входных символах и размерах архивный файл всегда был меньше несжатого исходного.

Заключение

Разница в количестве операций на архивацию / разархивацию между двумя алгоритмами невелика. Она составляет примерно десять тысяч операций для файлов размером меньше 1 Мб.

Разница в размере созданных архивных файлов также невелика. Для больших файлов она составляет примерно половину 1 Кб. На маленьких файла она меньше 100 байт.

На данных входных данных алгоритм Шеннона строит кодовое дерево очень близкое к оптимальному, то есть к дереву построенному алгоритмом Хаффмана, поэтому разница в работе алгоритмов минимальна.

Инструменты использованные для разработки / отладки

Разработка велась на дистрибутиве линукс ArchLinux (little-endian архитектура). Для компиляции использовался компиятор g++ (GCC) 6.2.1 20160830 и опцией –std=c++11. Для автоматизации сборки использовалась программа make. Код написан в редакторе vim 8.0. Для отладки использовался набор программ, в частности: gdb, xxd, bless. Для подготовки набора данных под архивацию, многократного запуска программы и автоматизации тестов использовались скрипты для интерпретатора python3.

Использованные источники

[1] Новиков Ф. А. Дискретная математика для программистов: Учебник для вузов. 3-е изд. — СПб.: Питер, 2009. ISBN 978-5-91180-759-7

[2] 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