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 Игорвеич

Москва 2016

Содержание

1 Постановка задачи

Необходимо было реализовать с использованием языка $C++$ программы для архивирования и текстовых файлов. При этом использовать два известных алгоритма кодирования информации:

1. Хаффмана (не адаптивный, 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.

Оба алгоритма работают в два прохода. Сначала строится таблица частот встречаемости символов в конкретном архивируемом файле (кодируем только те символы из набора допустимых, которые реально встречаются в файле). Затем строится кодовое дерево (не обязательное). По нему (или по таблице кодов) и архивируется файл. Для разархивирования алгоритмам потребуется знать таблицу, которая использовалась при архивировании. Соответствующая таблица должна сохраняться в архивном файле в самом его начале и использоваться при разархивировании. В начале пишется количество различных символов n, имеющихся в кодируемом файле, а затем n пар (код символа UTF-8, битовый код в архиве). Порядок — по убыванию частоты встречаемости символа в кодируемом файле.

Провести вычислительный эксперимент с целью оценки реализованных алгоритмов архивации / разархивации. Оценить количество элементарных операций каждого алгоритма. Для этого

1. Подготовить тестовый набор из нескольких текстовых файлов разного объема (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:
   (a) первый набор: символы латинского алфавита и пробел
   (b) второй набор: символы из первого набора + символы русского алфавита
   (c) третий набор: символы из второго набора + следующие знаки и спецсимволы: знаки арифметики „+ - * / =“, знаки препинания „. , ; : ? !“, „% @ # $ ‘’, скобки разных типов „( ) [ ] < >“ , кавычки „„“”),
2. Измерить (экспериментально) количество операций (в рамках модели RAM (взять из лекционного материала)), выполняемых за время работы (архивирования, разархивирования) каждого алгоритма на нескольких различных (не менее трех) файлах для каждого размера входного файла и набора символов (итого получается $8 \times 3 \times 3 = 72$ эксперимента по архивированию и 72 по разархивированию для каждого алгоритма,

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.

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 следующем 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 качетве аргумента указатель на этого ребенка. Аналогично с правым ребенком, только добавляем в вектор code не 0, а 1. Если же и левуй 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 проиходит 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. Записваю 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 еденице, 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 сида «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, пишум buf в файл и обнуляем его. В конце, если у нас кол-во бито оказалось не кратным 8, пишум 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 еденица, то идем дальше. Считываем длину таблицы частот и такое количество раз сяитываем 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 бир 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, считывю 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 *probTable** - 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 рекрсивная функция. База: return, если в группе всего лишь 1 символ. Если 2 символа, первому добавляем к коду 0, а ко второму - 1. Для остальныйх вариантов: проходим по (отсортиированнной!) 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 текщего 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 алгориитму 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 сообщени, 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.

- **bool isInTable(vector<bool>)** - вспомогательная функция. Возвращает true, если в таблице кодов есть данный вектор, иначе - false.

- **char getchar(vector<bool>)** - вспомогательная функция. Возвращает символ, которому в таблице кодов соответствует данный вектор.