Московский авиационный институт (национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Курсовой проект по курсу «Дискретный анализ»

Студент: М. А. Бронников
Преподаватель: А. А. Журавлёв
Группа: М8О-207Б

Дата:
Оценка:
Подпись:

Москва, 2019

Архиватор (Huffman + BWT + MTF + RLE)

Задача:
Необходимо реализовать четыре известных метода пребрзования данных для сжатия одного файла.

Формат запуска должен быть аналогичен формату запуска программы gzip, должны быть поддержаны следующие ключи: -c, -d, -k, -l, -r, -t, -1, -9. Должно поддерживаться указание символа дефиса в качестве стандартного ввода.

Ключи:

- **-c** - произести вывод результата в стандартный вывод
- **-k** - не удалять входной файл после сжатия
- **-l** - вывести информацию о сжатом файле
- **-d** - разархивирование файла
- **-r** - рекурсиный проход по директориям с архивироанием лежащих файлов
- **-t** - проерк целостности файла
- **-1** - быстрое сжатие
- **-9** - глубокое сжатие

Помимо ключей при запуске следует указать пути к архивируемым файлам. Если пути не указаны, программа должна считать данные из стандартного поток ввода.

1 Описание

Принцип работы программы:

Необходимо разработать программу, которая для сжатия заданного файла последовательно применяет следующие алгоОснритмы сжатия:

1. **Burrows-Wheeler transform**
2. **Move-To-Front**
3. **Run-Length Encoding**
4. **Код Хаффмана**

Такая последоательность обусловленная тем, что *преобразоавние Барроуза-Уилера* преобразует данные к виду, в котором часто встречаются последовательности поторяющихся символов. С такими последоательностями хорошо работают алгоритмы *MTF* и *RLE* для последующего преобразования алгоритмом Хаффмана, который после оптимизирует кодирование повторяющихся последовательностей, полученных после *BWT* что улучшает степень сжатия. [4]

Принцип работы программы:

Основной принцип работы заключается в следующей последовательности шагов:

1. Открытие файлового потока, который необходимо сжать
2. Создание временного файла для размещения в нем инормации из потока
3. Последовательное применение алгоритмов сжатия к временному файлу. После применения каждого алгоритма преобразоавния на выходе получается новый временный файл с результатом работы алгоритма, после чего закрывается и удаляется старый временный файл.
4. После применения всех алгритмов преобразоавния на основе полученного результпата сжатия ормируется результирующий файл, который будет содерать необходимую информацию и результат сжатия.

Преобразование Барроуза-Уилера:

«Преобразование Барроуза — Уилера (англ. Burrows-Wheeler transform) — алгоритм, используемый для предварительной обработки данных перед сжатием, разработанный для улучшения эффективности последующего кодирования. Преобразование Барроуза — Уилера меняет порядок символов во входной строке таким образом, что повторяющиеся подстроки образуют на выходе идущие подряд последовательности

одинаковых символов.» [3]

Преобразование выполняется в три этапа:

1. Составляется таблица всех циклических сдвигов входной строки.
2. Производится лексикографическая (в алфавитном порядке) сортировка строк таблицы.
3. В качестве выходной строки выбирается последний столбец таблицы преобразования и номер строки, совпадающей с исходной.

Наивная реализация первых двух этапов описанного алгоритма будет иметь как пространстенную, так и временную сложность $O(n^2)$. Что непозволительно долго.
Поэтому я воспользовался идеей применения суффиксного массива, который по определению является перестанокой всех суффиксов строки в лексикографическом порядке. Сущестуют довольно сложные алгоритмы, например описанный в [6] *алгоритм Каркkайнена-Сандерса*, производящие построение суффиксного массива за время $O(n)$, однако при этом встает нетривиальная задача преобразования суффиксного массива к перестаноке циклических сдвигов.
Поэтому я решил пожертвоать асимптотикой и остановился на описанном в [1] алгоритме построения суффиксного массива за $O(n \log n)$, который обладает полезным свойством: он строит массив на основе сортировки циклических сдвигов путем добавления к алгоритму терминального символа, что делает этот метод идеально адаптируемым к нашей задаче, так как мы получим необходимый результат лишь пропустив шаг добавления терминала.
К приятным свойствам алгоритма можно отнести и то, что он требует $O(n)$ памяти.

Опишем алгоритм построения массива. Как сказано в [1]:
«На нулевой фазе мы должны отсортировать циклические подстроки длины 1, т.е. отдельные символы строки, и разделить их на классы эквивалентности (просто одинаковые символы должны быть отнесены к одному классу эквивалентности). Это можно сделать тривиально, например, сортировкой подсчётом. Для каждого символа посчитаем, сколько раз он встретился. Потом по этой информации восстановим отсортированный массив. После этого, проходом по массиву и сравнением символов, строится массив классов эквивалентности.
Далее, пусть мы выполнили $k-1$-ю фазу, теперь научимся за $O(n)$ выполнять следующую, k-ю, фазу. Поскольку фаз всего $O(\log n)$, это даст нам требуемый алгоритм с временем $O(n \log n)$. Для этого заметим, что циклическая подстрока длины 2^k состоит из двух подстрок длины 2^{k-1}, которые мы можем сравнивать между собой за $O(1)$, используя информацию с предыдущей фазы — номера классов эквивалентности. Таким образом, для подстроки длины 2^k, начинающейся в позиции i, вся необхо-

димая информация содержится в паре чисел классов эквивалентности для позиции i и $i + 2^{k-1}$.
Это даёт нам весьма простое решение: отсортировать подстроки длины 2^k просто по этим парам чисел, это и даст нам требуемый порядок.
Воспользуемся здесь приёмом, на котором основана так называемая цифровая сортировка: чтобы отсортировать пары, отсортируем их сначала по вторым элементам, а затем — по первым элементам (но уже обязательно стабильной сортировкой, т.е. не нарушающей относительного порядка элементов при равенстве).»

После построения массива сдигов выполнить 3-ий шаг преобразоания BWT - трииальная задача.

При декодировании воспользуемся оптимизацией наивного алгоритма. Наиный алгоритм из [3]: «Выпишем в столбик нашу преобразованную последовательность символов. Запишем её как последний столбик предыдущей матрицы (при прямом преобразовании Барроуза — Уилера), при этом все предыдущие столбцы оставляем пустыми. Далее построчно отсортируем матрицу, затем в предыдущий столбец запишем преобразоанную последовательность. Опять построчно отсортируем матрицу. Продолжая таким образом, можно восстановить полный список всех циклических сдвигов строки, которую нам надо найти. Выстроив полный отсортированный список сдвигов, выберем строку с номером, который нам был изначально дан. В итоге мы получим искомую строку.»

Оптимизация наивного алгоритма из [3]: «Заметим, что при каждом проявлении неизвестного столбца выполнялись одни и те же действия. К предыдущему приписывался новый столбец и имеющиеся данные сортировались. На каждом шаге к строке, которая находилась на i-ом месте, приписывался в начало i -ый элемент столбца входных данных. Пусть изначально известно, каким по порядку является приписанный в начало символ (то есть каким по порядку в столбце). Из предыдущего шага известно, какое место занимала строка без этого первого символа (i -ое). Тогда несложно заметить, что при выполнении такой операции строка с номером i всегда будет перемещаться на позицию с номером j.
Поскольку в нашем алгоритме новый столбец приписывается в начало, то мы из состояния i (левый столбец) переходим в состояние j (правый). Для того, чтобы восстановить строку, нам необходимо от последней такой цифры по пути из j в i восстановить строку. »

Описанный в [3] алгоритм декодирования дает сложность $O(n + m)$, где m - размер алфавита.

Преобразоавание MTF:

«Преобразование MTF (англ. move-to-front, движение к началу) — алгоритм кодирования, используемый для предварительной обработки данных (обычно потока байтов) перед сжатием, разработанный для улучшения эффективности последующего кодирования.»[4]

Мы будем использовать наивный способ реализации алгоритма, описанного в [4]: «Изначально каждое возможное значение байта записывается в список (алфавит), в ячейку с номером, равным значению байта, т.е. $(0, 1, 2, 3, ..., 255)$. В процессе обработки данных этот список изменяется. По мере поступления очередного символа на выход подается номер элемента, содержащего его значение. После чего этот символ перемещается в начало списка, смещая остальные элементы вправо.»

Изменяющийся алфавит реализуем при помощи структуры - список.

Декодирование будет производится последовательно аналогично шагам кодирования.

Кодирование длинн серий:

«Кодирование длин серий (англ. run-length encoding, RLE) или кодирование повторов — алгоритм сжатия данных, заменяющий повторяющиеся символы (серии) на один символ и число его повторов. Серией называется последовательность, состоящая из нескольких одинаковых символов. При кодировании (упаковке, сжатии) строка одинаковых символов, составляющих серию, заменяется строкой, содержащей сам повторяющийся символ и количество его повторов.»[5]

При реализации будем последовательно считать либо количество подряд идущих неповторяющихся символов, либо неповторяющихся символов, записываем полученное число перед каждой серией: отрицательное если серия неповторяющихся символов и положительное если серия повторяющихся, после чего помещаем серию после отрицательного числа или один повторяющийся после положительного.

Интересным моментом является то, что мы кодируем подсчитываемое число одним байтом, что дает нам закодировать серии до 127 повторяющихся символов и до 128 неповторяющихся, после чего будет необходимо обнулить счетчик и рассматривать дальнейшую последовательность символов как новую серию.

Декодирование происходит нивным методом: считываем размер серии, и если он полоительный выписываем нужное количество раз повторяющийся символ, иначе выписывем указанное количество симолов после числа из входной последовательности.

Алгоритм Хаффмана:

«Идея, положенная в основу кодировании Хаффмана, основана на частоте появления символа в последовательности. Символ, который встречается в последовательности чаще всего, получает новый очень маленький код, а символ, который встречается реже всего, получает, наоборот, очень длинный код.»[2]

Важное свойсво кодирования: каждый код символа не является префиксом для кода другого символа.

Алгоритм будет использовать очередь с приоритетом для построения дерева. Приоритетом будет выступать частота встречи символа в последовательности.
Описание алгоритма на основе материала из [2]:

1. Для начала посчитаем частоты всех символов
2. После вычисления частот мы создадим узлы бинарного дерева для каждого знака и добавим их в очередь, используя частоту в качестве приоритета.
3. Теперь мы достаём два первых элемента из очереди и связываем их, создавая новый узел дерева, в котором они оба будут потомками, а приоритет нового узла будет равен сумме их приоритетов. После этого мы добавим получившийся новый узел обратно в очередь.
4. После того, как мы свяжем два последних элемента, получится итоговое дерево
5. Теперь, чтобы получить код для каждого символа, надо просто пройтись по дереву, и для каждого перехода добавлять 0, если мы идём влево, и 1 — если направо

После построения дерева построим таблицу кодов при помощи обхода дерева для быстрого кодирования символа.

Дерево мы сохраним в файле для возмоности декодирования последовательности.

При декодировании необходимо считать дерево из входной последоавтельности, после чего в заисимости от встреченного бита во входной последовательности будем двигаться влево или вправо пока не придем в лист. Если попадаем в лист выписываем полученный символ в выходную последовательность и возвращаемся в корень.

2 Исходный код

Структура программы построена таким обрзом, что каждый класс, инкапсулирующий в себе реализацию алгоритма преобразования наследуется от общего класса **Compressor**, переопределяя его метод *buffer_ encode* и *buffer_ decode*.

Определение класса **Compressor** в *Compressor.hpp*:

```
class Compressor{
    public:
        Compressor();
        Compressor(istream& inpt, ostream& otpt);
        Compressor(istream* inpt, ostream* otpt);
        void set_input(istream& inpt);
        void set_input(istream* inpt);
        void set_output(ostream* otpt);
        void set_output(ostream& otpt);
        uint64_t get_until_size();
        uint64_t get_after_size();
        void encode();
        void decode();
        virtual ~Compressor(){};

    protected:
        virtual void buffer_encode() = 0;
        virtual void buffer_decode() = 0;
        uint64_t until_size = 0;
        uint64_t after_size = 0;
        istream* is;
        ostream* os;
        string in_buffer;
        string out_buffer;
        const uint32_t max_buffer_size = 4194304;
};
```

Класс **RLE** объялен в *RLE.hpp* следующим образом:

```
class RLE : public Compressor{
    public:
        RLE();
        RLE(istream& inpt, ostream& otpt);
        RLE(istream* inpt, ostream* otpt);
    protected:
        void buffer_encode() override;
        void buffer_decode() override;

```

```
};
```

Класс **MTF** из *MTF.hpp* содержит в себе список *alfabet*, реализующий в себе алгоритм, а так же 2 функции позоляющие получить код для входного символа:

```
class MTF : public Compressor{
    public:
        MTF();
        MTF(istream& inpt, ostream& otpt);
        MTF(istream* inpt, ostream* otpt);

    protected:
        void buffer_encode() override;
        void buffer_decode() override;
    private:
        list<uint8_t> alfabet;
        uint8_t move_to_front_encode(uint8_t c);
        uint8_t move_to_front_decode(uint8_t c);
};
```

Класс **Huffman** из *Huffman.hpp* имеет функции для подсчета входений симола в файл, создания дерева, его чтения и записи, а таке несколько вспомогательных функций и структур для работы с деревом.

```
class Huffman : public Compressor{
    public:
        Huffman();
        Huffman(istream& inpt, ostream& otpt);
        Huffman(istream* inpt, ostream* otpt);
    protected:
        void buffer_encode() override;
        void buffer_decode() override;
    private:
        typedef struct Node{
            int32_t left;
            int32_t right;
            uint32_t count;
            char value; // has value only if list, else it rubish
        }node;

        // comparator for priority queue
        typedef struct functor{
            bool operator()(const pair<int32_t, uint32_t> lhs, const pair<int32_t,
                uint32_t> rhs){
                return lhs.second > rhs.second;
            }
        }comparator;
```

```
23
24          // TREE (too lazy create new structure):
25          int32_t tree = -1; // root
26          vector<node> vertex; // vector of vertexes of tree
27          // AND OF TREE
28
29          void tree_create(); // create tree of encoding
30          void tree_from_input(uint32_t& i); // create tree from input for decoding
31          void tree_write(int32_t obj); // write tree in output
32          void count_table(map<char, uint32_t>& table_of_counts); // count simbols
33          void create_encode_table(map<char, vector<uint8_t>>& encode_table);
34          void depth_walker(int32_t obj ,vector<uint8_t>& vec, map<char, vector<uint8_t
                >>& mp);
35  };
```

Класс **BWT** из *BWT.hpp* содержит в себе функцию построения суффиксного массива и позицию исходной строки в таблице сдигов.

```
1   class BWT : public Compressor{
2       public:
3           BWT();
4           BWT(istream& inpt, ostream& otpt);
5           BWT(istream* inpt, ostream* otpt);
6       protected:
7           void buffer_encode() override;
8           void buffer_decode() override;
9       private:
10          uint32_t position; // position in sorted cyclic table
11          uint32_t count_suff_array(vector<uint32_t>& array); // returns position
12  };
```

Полный код на моем *GitHub: https://github.com/Bronnikoff/DA/tree/master/KP*

3 Консоль

```
(base) max@max-X550CC:~/DA/KP$ ls
Arhivator  main.cpp  Makefile test.txt
(base) max@max-X550CC:~/DA/KP$ make
g++ -std=c++17 -Wall -pedantic -c Arhivator/Compressors/Compressor.cpp
g++ -std=c++17 -Wall -pedantic -c Arhivator/Arhivator.cpp
g++ -std=c++17 -Wall -pedantic -c Arhivator/Compressors/RLE/RLE.cpp
g++ -std=c++17 -Wall -pedantic -c Arhivator/Compressors/BWT/BWT.cpp
g++ -std=c++17 -Wall -pedantic -c Arhivator/Compressors/Huffman/Huffman.cpp
g++ -std=c++17 -Wall -pedantic -c Arhivator/Compressors/MTF/MTF.cpp
g++ -std=c++17 -Wall -pedantic -c main.cpp
g++ Compressor.o Arhivator.o RLE.o BWT.o Huffman.o MTF.o main.o -o KP
rm -rf *.o
(base) max@max-X550CC:~/DA/KP$ ./KP test.txt
(base) max@max-X550CC:~/DA/KP$ ls
Arhivator  main.cpp  Makefile KP test.txt.gz
(base) max@max-X550CC:~/DA/KP$ ./KP -d test.txt.gz
(base) max@max-X550CC:~/DA/KP$ ls
Arhivator  main.cpp  Makefile KP test.txt
(base) max@max-X550CC:~/DA/KP$ exit
```

4 Выводы

Выполнив Курсовой проект по курсу «Дискретный анализ», я познакомился с такой тяжелой, но очень интересной темой, как сжатие данных. Данное задание заставило меня вспомнить почти все, что я прошел за 2 семестра изучения дисциплины: деревья, очереди, суффиксные массивы, динамическое программирование, алгоритмы сжатия, сортировки за линейное время - все это мне пришлось использовать в этой работе. Также не обошлось без знаний ООП для реализации наследуемых классов и виртуальных функций.

Из минусов своей реализации я вынужден отметить то, что я применяю последоательно алгоритмы сжатия для временных файлов побуфферно, что тратит отгромное количество памяти на жестком диске и занимает время на перезаписывание. Мне стоило считывать кодируемы файл побуфферно, после чего последоательно применять алгоритмы сатия непосредстенно к буфферу.

Однако несмотря на всю несоершенность моей реализации я получил огромный опыт при работе над курсовым проектом и, возможно, буду испольэоть его в дальнейшем в качестве портфолио и демонстрции своих навыков.

Список литературы

[1] *Суффиксный массив — MAXimal*
URL: `https://e-maxx.ru/algo/suffix_array` (дата обращения: 05.12.2019).

[2] *Алгоритм Хаффмана на пальцах — Хабр*
URL: `https://habr.com/ru/post/144200` (дата обращения: 02.12.2019).

[3] *Преобразование Барроуза-Уилера — ИТМО*
URL: `https://neerc.ifmo.ru/wiki/index.php?title=Преобразование_Барроуза-Уилера` (дата обращения: 04.12.2019).

[4] *Преобразование MTF — ИТМО*
URL: `https://neerc.ifmo.ru/wiki/index.php?title=Преобразование_MTF` (дата обращения: 04.12.2019).

[5] *Кодирование длин серий — Википедия*
URL: `https://ru.wikipedia.org/wiki/Кодирование_длин_серий` (дата обращения: 01.12.2019).

[6] *Алгоритм Каркkайнена-Сандерса — ИТМО*
URL: `https://neerc.ifmo.ru/wiki/index.php?title=Алгоритм_Каркkайнена-Сандерса` (дата обращения: 04.12.2019).