Курсовая работа по курсу дискретного анализа: Архиватор LZ-77

Выполнил студент группы М80-308Б-20 Морозов Артем Борисович.

Условие

Кратко описывается задача:
Ваша программа должна читать входные данные из стандартного потока ввода и выводить ответ на стандартный поток вывода.
Вам будут даны входные файлы двух типов.
Первый тип:
compress
<text>

Текст состоит только из малых латинских букв. В ответ на него вам нужно вывести тройки, которыми будет закодирован данный текст.
Второй тип:
decompress
<triplets>

Вам даны тройки (<offset, len, char>) в которые был сжат текст из малых латинских букв,
вам нужно его разжать.

Метод решения

Алгоритм LZ-77 принадлежит семейству алгоритмов Лемпеля-Зива словарных методов сжатия текстов. Алгоритмы этого семейства оперируют некоторым словарем: на каждой итерации алгоритма мы берем слово и пытаемся найти это слово в нашем словаре, если есть – заменяем его на его код. Чтобы реализовать алгоритм LZ-77, нужно было реализовать операции сжатия и декодирования текста.

Сжатие текста заключается в поиске **максимальной** подстроки, которая начинается с символа alpha незакодированной части и совпадает с какой-то последовательностью из закодированной части. Сжатие происходит тройками – на каждой итерации у нас заполняется экземпляр структуры **Node:** <offset, size, next>, где **offset –** смещение на некоторое количество символов, чтобы найти совпадающую строку с обрабатываемой на данный момент незакодированной частью; **size –** размер строки, которая совпадает с подстрокой незакодированной части; **next –** следующий символ в необработанной части.

Декодирование текста заключается попросту в декодировании последовательности троек: мы просто выполняем процедуру, обратную сжатию – смещаемся на величину нашего смещения **offset,** берем **size** букв, пишем их, и дописываем **next** символ.

Описание программы

```
#include <iostream>
#include <vector>
const char END_OF_FILE = ' ';

//<offset, size, next> - code
struct Node {
    int offset;
    int size;
    char next;
    Node (int offset, int size, char next) {
        this->offset = offset;
        this->size = size;
        this->next = next;
    }
};

void Encode (std:: string& text, std:: vector<Node*>& encoded) {
    int global_position = 0;
    while (global_position < text.size()) {
        Node* result = new Node(0, 0, text[global_position]);
        for (int i = 0; i < global_position; ++i) {
            int local_position = global_position;
            int local_size = 0;
            for (int j = i; text[local_position] == text[j] && local_position <
text.size(); ++j) { //смещаемся по тексту и ищем самый большой size
                ++local_size;
                ++local_position;
            }
            if (local_size >= result->size && local_size != 0) {
                result->size = local_size;
                result->offset = global_position - i;
                if (local_position >= text.size()) { //если превысили или на конце текста
                    result->next = END_OF_FILE;
                }
                else {
                    result->next = text[local_position]; //кладем следующий символ, если
все ок
                }
            }
        }
```

```
        encoded.emplace_back(result);
        global_position += result->size + 1; // смещаем наш итератор на следующую ячейку
после последнего обработанного символа
    }
}

void Decode (std:: string& text, std:: vector<Node*>& code) {
    int global_position = 0;
    for (int i = 0; i < code.size(); ++i) {
        if (code[i]->size > 0) { //если есть что брать
            global_position = text.size() - code[i]->offset; //смещаемся на длину строки
- смещение
            for (int j = 0; j < code[i]->size; ++j) {
                text.push_back(text[global_position + j]); //прибавляем в текст пока j <
размера того, что берем
            }
        }
        if (code[i]->next == END_OF_FILE) { //если пустой, выходим
            break;
        }
        else {
            text.push_back(code[i]->next); //добавляем следующий символ в текст, если не
пустой символ
        }
    }
}

int main() {
    std:: ios:: sync_with_stdio(false);
    std:: cin.tie(0);
    std:: cout.tie(0);
    std:: string operation;
    std:: string text;
    std:: vector<Node*> encoded;
    std:: vector<Node*> code;
    std:: cin >> operation;
    if (operation == "compress") {
        std:: cin >> text;
        Encode(text, encoded);
        for (int i = 0; i < encoded.size(); ++i) {
            std:: cout << encoded[i]->offset << " " << encoded[i]->size << " " <<
encoded[i]->next << "\n";
        }
```

```
    }
    else if (operation == "decompress") {
        int offset, size;
        char next = END_OF_FILE;
        while (std:: cin >> offset >> size) {
            std:: cin >> next;
            Node* inputed = new Node(offset, size, next);
            code.emplace_back(inputed);
        }
        if (code[code.size() - 1]->next == code[code.size() - 2]->next) {
            code[code.size() - 1]->next = END_OF_FILE;
        }
        Decode(text, code);
        std:: cout << text << "\n";
    }
    for (int i = 0; i < code.size(); ++i) {
        delete code[i];
    }
    for (int i = 0; i < encoded.size(); ++i) {
        delete encoded[i];
    }
    return 0;
}
```

Дневник отладки

Функции сжатия и декодирования в самом начале работали не совсем корректно и выдавали вместо ожидаемых результатов ошибочные, однако это было быстро устранено.

Тест производительности

Для начала протестируем функцию **сжатия**: сгенерируем 5 строк разной длины: 5000, 10000, 20000, 30000, 500000.

1) N = 5000:

```
21 ms
```

2) N = 10000:

```
84 ms
```

3) N = 20000:

```
220 ms
```

4) N = 50000:

```
1162 ms
```

5) N = 100000:

```
4398 ms
```

Как мы видим, время очень сильно увеличивается, когда N меняется от 50000 до 100000.

Теперь протестируем функцию **декодирования**:
1) Декодирование слова в 5000 букв:

```
5 ms
```

2) Декодирование слова в 10000 букв:

```
14 ms
```

3) Декодирование слова в 20000 букв:

```
37 ms
```

4) Закодированное слово в 50000 слов:

```
104 ms
```

5) Закодированное слово в 100000 слов:

```
1094 ms
```

Как мы видим, время так же очень сильно растет между 4 и 5 пунктами. Оно и логично – ведь мною была реализована наивная версия этого алгоритма. Так как мы в цикле по длине строки еще и ищем подстроку за квадратичную сложность, то итоговая сложность такой реализации асимптотически похожа на кубическую, то есть на O(N^3).

Недочёты

Недочетов в программе обнаружено не было, однако стоит упомянуть, что программа работает только при условии корректного ввода, так как была разработана исключительно в учебных целях. Любой неправильный ввод может убить работоспособность моей программы.

Выводы

Сделав упрощенный курсовой проект по дискретному анализу, я познакомился с таким интересном алгоритмом, как архиватор LZ-77 и научился его программировать. Повторюсь, что была разработана лишь наивная реализация алгоритма. При использовании суффиксного дерева можно было бы добиться квадратичной сложности, однако и запрограммировать это было бы гораздо сложнее.