Московский авиационный институт (национальный исследовательский университет)

Факультет информационных технологий и прикладной математики
Кафедра вычислительной математики и программирования

Лабораторная работа №4 по курсу "Дискретный анализ"

Студент:Кондратьев Егор

Преподаватель: А. А. Кухтичев

Группа: М8О-206Б

Дата:

Оценка:

Подпись:

Москва, 2021

Лабораторная работа №4

Задача: Необходимо реализовать один из стандартных алгоритмов поиска образцов для указанного алфавита.

Вариант алгоритма: Поиск большого количества образцов при помощи алгоритма Ахо-Корасик.

Вариант алфавита: Слова не более 16 знаков латинского алфавита (регистронезависимые).

1 Описание

Требуется написать алгоритм Ахо-Корасик для поиска подстроки в строке. Существует много алгоритмов для поиска подстроки в строке, таких как: алгоритм Кнута-Морриса-Пратта, Бойера-Мура, Апостолико-Джанкарло. Но все они полезны для поиска одной подстроки в тексте, так как их сложность может возрасти в n раз, если необходимо будет искать n подстрок. Это связано с тем, что для поиска каждой подстроки необходимо будет заново пройтись по тексту.

На помощь приходит алгоритм Ахо-Корасика, согласно [1] его сложность равна O(m+t+a), где m - сумма длин всех подстрок, t - длина текста, a - количество ответов. Алгоритм Ахо-Корасика несложен: сначала из подстрок строится обычный бор, затем его преобразование, добавляя суффиксные ссылки и терминальные (их еще назы-вают сжатыми суффиксными) ссылки, и после этого просто просто идем по тексту, переходя по узлам бора.

2 Исходный код

Вставка строки в бор происходит следующим способом, сначала мы находимся в корне и начинаем идти по строке:

1. Если в детях текущего узла нет слова, на котором мы остановились, то вставляем его.

2. Если есть, то переходим в узел со значением этого слова.

3. Переходим к следующему слову.

Когда строка закончилась, мы сохраняем в узел, в котором мы остановились, информацию о том, что это терминал, то есть конец слова.

Затем в бор нам необходимо добавить суффиксные ссылки. Идти по бору мы будем поиском в ширину:

1. Сначала находим суффиксную ссылку для узла:

   (a) Переходим к родителю.

   (b) Идём по суффиксным ссылкам, пока в узле по суффиксной ссылке не будет ребёнка со значением рассматриваемого узла. Если ребёнок есть, то суффиксной ссылкой указываем на это узел.

   (c) Если мы пришли в корень и у корня нет нужного ребёнка, то суффиксная ссылка - корень.
2. Затем необходимо определить терминальную ссылку:

   - Если узел по суффиксной ссылке - "терминальный", то терминальной ссылкой указываем на него.

   - В другом случае терминальная ссылка равна терминальной ссылке узла по суффиксной ссылке.
3. Добавляем детей текущего узла в очередь и рассматриваем относительно них.

Поиск подстрок происходит следующим способом. Сначала рассматриваемый узел - корень, мы идём по слова в строке:

1. Если текущего слова нет в детях текущего узла, то переходим по суффиксным ссылкам, пока не найдем необходимого ребёнка или не придем в корень.

2. Если текущий узел - "терминальный", то значит подстрока совпала и сохраняем узел в результат.

Если у текущего узла есть терминальная ссылка, то сохраняем узлы по ним в результат.

3. Переходим к следующему слову в строке.

Таблица функций:

main.cpp	
void SplitLine(string& line, vector<string>& vec)	Функция, разбивающая строку на вектор слов.
corasik.cpp	
VNode* VNode::GetChild(string& str)	Функция, возвращающая указатель на ребёнка со значением str
void VTrie::AddPattern(vector<string>& pattern, TUll line)	Функция, добавляющая слова в бор.
VNode* VTrie::GetSuffix(VNode* node, string& val)	Функция, возвращающая суффиксную ссылку для узла.
void VTrie::MakeSuffixes()	Функция, обрабатывающая бор: определяющая суффиксные ссылки.
void VTrie::SearchPatterns(vector<VInput>& text, vector<VResults>& res)	Функция, выполняющая поиск подстрок в тексте.
void VTrie::DeleteTrie(VNode* node)	Функция, удаляющая бор.

Структуры и классы без реализаций их методов:

```
struct VInput
{
    std::string Str;
    TUll Line;
    TUll Number;
};

struct TResults
{
    TUll Line;
    TUll Word;
    TUll Sample;
    TResults(TUll line, TUll word, TUll sample) : Line(line), Word(word), Sample(sample) {}
};

namespace NAlgo
{

    struct TNode
    {
        std::string Value;
        TUll Line;
        TUll Length;
        TNode *Par;
```

```
        TNode *SuffLink;
        TNode *TerminalLink;
        std::unordered_map<std::string, TNode *> Childs;
        std::unordered_map<std::string, TNode *> Next;

        TNode() : Value(""), Line(0), Length(0), Par(nullptr), SuffLink(nullptr), TerminalLink(nullptr) {}

        TNode(std::string &val, TNode *p) : Value(val), Line(0), Length(0), Par(p), SuffLink(nullptr),
TerminalLink(nullptr) {}

        TNode *GetChild(std::string &str);
        bool IsTerminal()
        {
            return Length > 0;
        }
    };

    class TTrie
    {
        TNode *GetSuffix(TNode *node, std::string &val);
        void DeleteTrie(TNode *root);

        TNode *Root;

    public:
        void AddPattern(std::vector<std::string> &pattern, TUll line);
        void MakeSuffixes();
        void SearchPatterns(std::vector<VInput> &text, std::vector<TResults> &res);

        TTrie() : Root(new TNode())
        {
            Root->SuffLink = Root;
        }

        ~TTrie()
        {
            DeleteTrie(Root);
        }
    };

}
```

3 Консоль

root@Du$ make

g++ -std=c++14 -pedantic -Wall -c corasik.cpp -o corasik.o

g++ -std=c++14 -pedantic -Wall main.cpp corasik.o -o solution

root@Du$ cat tests/main

cat dog cat dog
CAT dog CaT
Dog doG dog dOg

Cat doG cat dog cat dog cat Parrot
doG dog DOG DOG dog

root@Du$ ./solution <tests/pool.txt

1,1,2

1,1,1

1,3,2

1,3,1

1,5,2

2,1,3

2,2,3

4 Тест производительности

Тест производительности представляет из себя следующее: поиск образцов с помо-щью алгоритма Ахо-Корасика сравнивается с поиском с помощью std::find. Время построения бора и суффиксных ссылок в нём не учитывается. С помощью std::find ищутся все вхождения образца в тексте, а не одно. Тест состоит из 100 образцов, длина которых может быть от 2 до 10 слов, и текста, состоящего из 10^5 слов. Каждое слово состоит из 1 или 2 букв.

root@Du$ make benchmark

g++ -std=c++14 -pedantic -Wall -c corasik.cpp -o corasik.o

g++ -std=c++14 -pedantic -Wall benchmark.cpp corasik.o -o benchmark

root@Du$ ./benchmark <tests/pool1.txt

aho time: 40ms

find time: 87ms

Как видно, алгоритм Ахо-Корасика выиграл у std::find почти в 2 раза. Скорее всего, это связано с тем, что алгоритм Ахо-Корасика за один проход по тексту находит все подстроки, в то время, как std::find необходимо для каждого образца заново идти по всему тексту.

5 Выводы

Выполнив четвертую лабораторную работу по курсу "Дискретный анализ", я лучше разобрался с алгоритмами поиска подстрок в подстроке, особенно в алгоритме Ахо-Корасика, так как реализовывал его. Это полезно, так как в будущем это может помочь мне с выбором правильного решения для поиска подстроки. К примеру, если мне необходимо будет искать сразу несколько подстрок в строке, то, конечно же, я буду использовать алгоритм Ахо-Корасика. А если будет необходимо искать только одну подстроку, то для простоты можно реализовать и алгоритм Кнута-Морриса-Пратта.

Так как алфавитом являлись слова длины до 16 символов, то я решил использовать map для их хранения, но сложность моего алгоритма возросла, как минимум, до O((n + t) lg(n) + a). Но после я вспомнил, что существует такая структура, как unordered_map, основанная на хеш-таблице. И так как сложность вставки и поиска по ней, в среднем O(1), то в моем случае это гораздо лучше.

Список литературы

[1] Алгоритм Ахо Корасик Википедия.

URL: https://ru.wikipedia.org/wiki/Алгоритм_Ахо__Корасик
(дата обращения: 15.11.2021).

[2] Алгоритм Ахо Корасик Хабр.

URL:https://habr.com/ru/post/198682/(дата обращения: 15.11.2021).

[3] Алгоритм Ахо Корасик Вики Конспекты.

URL: https://neerc.ifmo.ru/wiki/index.php?title=Алгоритм_Ахо-Корасик

(дата обращения: 15.11.2021).