Московский авиационный институт (национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторная работа №5 по курсу «Дискретный анализ»

Студент: А. Т. Бахарев
Преподаватель: А. А. Кухтичев
Группа: М8О-206Б

Дата:
Оценка:
Подпись:

Москва, 2018

Лабораторная работа №5

Задача: Необходимо реализовать алгоритм Укконена построения суффиксного дерева за линейное время. Построив такое дерево для некоторых из входных строк, необходимо воспользоваться полученным суффиксным деревом для решения своего варианта задания.

Алфавит строк: строчные буквы латинского алфавита (т.е., от a до z).

Вариант задания : 2 Поиск с использованием суффиксного массива. Найти в заранее известном тексте поступающие на вход образцы с использованием суффиксного массива.

Входные данные: `текст располагается на первой строке, затем, до конца файла, следуют строки с образцами.`

Выходные данные: `для каждого образца, найденного в тексте, нужно распечатать строчку, начинающуюся с последовательного номера этого образца и двоеточия, за которым, через запятую, нужно перечислить номера позиций, где встречается образец в порядке возрастания.`

1 Описание

Суффиксное дерево

Суффиксное дерево T для m-символьной строки S:

• Ориентированное дерево, имеющее ровно m листьев, пронумерованных от 1 до m.

• Каждая внутренняя вершина, отличная от корня, имеет не меньше двух детей.

• Каждая дуга помечена непустой подстрокой строки S (дуговая метка).

• Никакие две дуги, выходящие из одной вершины, не могут иметь меток, начинающихся с одинаковых символов.

• Для каждого листа i конкатенация меток от корня составляет $S[i..m]$

Наивный алгоритм строит суффиксное дерево за $O(m^2)$, однако его нельзя улучшить до линейного времени. Эско Укконен произвел несколько разумных и вполне простых трюков над алгоритмом, который работал за $O(m^3)$. Теперь мы можем строить суффиксное дерево за $O(m)$.

Общее описание алгоритма Укконена:

– Последовательно строим неявные деревья T_i для каждого префикса $S[1..i]$.

– Настоящее суффиксное дерево T можно получить из T_m построив следующее неявное дерево для строки с терминальным символом.

Первое ускорение: Суффиксные связи

Определение Пусть $x\alpha$ обозначает произвольную строку, где x – её первый символ,а α – оставшаяся подстрока. Если для внутренней вершины u с путевой меткой $x\alpha$ существует другая вершина $s(u)$ с путевой меткой α, то указатель из u в $s(u)$ называется **суффиксной связью**.

Второе ускорение: Сжатие суффиксных меток

В каждом ребре будет хранится не только один символ, а целая подстрока. Причем будем хранить ее не явно, а только координаты начала и конца. Так же, для всех листьев сделаем общий счетчик end. При добавлении новой буквы увеличиваем его на 1.

Поиск образца в тексте

• Строится суффиксное дерево для текста.

• Ищется путь, совпадающий с образцом. Если такого пути нет, то образец в текст не входит.

• Если путь есть, то все листья поддерева – вхождения.

Построение суффиксного массива

• Для текста T построить суффиксное дерево T.

• Обойти дерево T в глубину таким образом, что первыми проходятся дуги, чьи метки меньше остальных в лексикографическом смысле.

• Если дуги хранятся в порядке возрастаний первых символов меток, то такой обход будет натуральным

• Суффиксный массив – просто список посещений листьев при таком обходе.

• Тем самым, суффиксный массив строится за время $O(m)$.

Поиск в суффиксном массиве

• Составляем обычный массив суффиксов данной строки длины m

• Сортируем массив в лексикографическом порядке. Теперь мы готовы искать вхождения образцов.

• Ищем вхождение при помощи бинарного поиска. Если нашли, то как минимум одно вхождение есть. Нужно проверить, нет ли еще. Начинаем "расширять границы"найденного шаблона. Двигаемся влево и вправо, пока шаблон является префиксом для других суффиксов строки.

• Выводим индексы совпадений

Работа программы

▶ Производим инициализацию суффиксного дерева. Передаем данные об алфавите и терминальном символе.

▶ Считываем текст. Производим вставку каждой буквы в суффиксное дерево. После этого имеем неявное суффиксное дерево. Вставляем терминальный символ, тем самым получая явное суффиксное дерево.

▶ Выполняем преобразование нашего дерева в суффиксный массив. Теперь мы готовы искать подстроки.

▶ Считываем по очереди все паттерны. Ищем их в суффиксном массиве и получаем вектор вхождений.

2 Исходный код

suffix_tree.h

```
#define MAX_LENGTH std::numeric_limits<std::size_t>::max()
class TNode
{
public:
    using link = std::map<char, TNode *>;
    link edges;
    TNode *parent;
    TNode *suffixLink;
    std::size_t begin;
    std::size_t length;
    TNode(const std::string &, TNode * const &);
    TNode(const std::vector<char> &, const char &);
    TNode(const std::vector<char> &, const char &, TNode * const &);
    TNode(const std::string &, TNode * const &, const std::size_t &);
    virtual ~TNode();
};
class TSuffixTree;
class TSuffixArray;
class TSuffixArray
{
private:
    std::string text;
    std::vector<std::size_t> array;
public:
    explicit TSuffixArray(const TSuffixTree &);
    std::vector<std::size_t> Find(const std::string &pattern) const;
    virtual ~TSuffixArray();
};
class TSuffixTree
{
    std::string text;
    TNode *root;
    TNode *active_vertex;
    std::size_t activeLength;
    std::size_t activeCharIdx;
    void DFS(TNode * const &, std::vector<std::size_t> &, const std::size_t &) const;
    friend TSuffixArray::TSuffixArray(const TSuffixTree &);
public:
    explicit TSuffixTree(const std::vector<char> &, const char &);
    void PushBack(const char &ch);
    virtual ~TSuffixTree();
};
```

suffix_tree.cpp	
explicit TSuffixTree(const std::vector<char>& , const char&)	Конструктор суффиксного дерева. Для создания экземпляра класса необходимо передать в констуктор алфавит и терминирующий символ.
explicit TSuffixArray(const TSuffixTree &)	Конструктор класса суффиксного массива. Массив строится на основании построенного суффиксного дерева.
void TSuffixTree::PushBack(const char& new_ch)	Функция добавления символа в суффиксное дерево в режиме реального времени.
void TSuffixTree::DFS(TNode const &curr, std::vector<std::size_t> &result, const std::size_t &summary)	Преобразование суффиксного дерева в суффиксный массив при помощи алгоритма обхода в глубину.
std::vector<std::size_t> TSuffixArray::Find(const std::string &pattern)	Поиск подстроки в суффиксном массиве.

3 Консоль

```
alex$make
g++ suffix_tree.cpp main.cpp -std=c++11 -pedantic -Wall -Werror -Wno-sign-compare
-Wno-long-long -lm -o diskran_laba_5
alex$ cat 01.t
bbcabcbacdcabbdbabacbacaddaabcaaddacbccbdaacbdcaaddbabbddadabcbbbdadcaadacbaaada
abbacbdddccbcabccdadabaccdacaddbbccd
bbac
bbca
aabc
dcab
dadd
dbab
dbac
bbba
acbd
bbba
ccab
bdad
dadd
bbca
alex$ ./*5 <01.t
1: 17
3: 49
4: 95
5: 82
6: 1
7: 27
8: 10
10: 15,51
13: 43,84
16: 65
18: 1
alex$
```

4 Тест производительности

Поиск подстроки в строке при помощи суффиксного массива я сравнивал с наивным алгоритмом. Ниже приведен листинг бенчмарка и генератора тестов на python: benchmark.cpp

```cpp
#include <cstdlib>
#include <iostream>
#include <string>
#include <vector>
std::vector<int> Find(const std::string& pattern, const std::string& text)
{
    std::vector<int> res;
    for(int i = 0; i < text.size(); ++i)
    {
        int k = i;
        for(int j = 0; j < pattern.size(); ++j)
        {
            if(text[k] == pattern[j])
                ++k;
            else
                break;
        }
        if(i - k == pattern.size() - 1)
            res.push_back(i);
    }
    return res;
}
int main()
{
    std::string text;
    std::string pattern;
    std::cin>>text;
    size_t n = 1;
    std::string buf;
    while(std::cin>>pattern)
    {
        std::vector<int> res = Find(pattern, text);
        if(!res.empty())
        {
            std::cout<<n<<": ";
            for(auto i : res)
                std::cout<<res[i]<<", ";
        }
        ++n;
    }
    return 0;
}
```

Generator.py

```
import random
import sys
def GenText(l, alphabet):
    text = str()
    for i in range(l):
        text += random.choice(alphabet)
    return text

inputFile = open("01.t", "w")
outputFile = open("01.a", "w")
textLen = 1000000
alphabet = ('a', 'b', 'c', 'd')
text = str(GenText(textLen, alphabet))# Generate text
inputFile.write(text + '\n')
pCount = random.randint(1000, 2000)# Generate count of patterns
for i in range(1, pCount):
    pattern = str()
    for j in range(random.randint(20, 50)):
        pattern += random.choice(alphabet)
    inputFile.write(pattern + "\n")
    buf = str()
    res = text.find(pattern, 0, len(text))
    isMatch = False
    if res != -1:
        buf += str(i) + ": " #write num of pattern if we have at least one match
    while res != -1:
        if isMatch:
            buf += ", "
        isMatch = True
        buf += str(res)
        res = text.find(pattern, res + 1, len(text))
    if len(buf) != 0:
        outputFile.write(buf + "\n")
```

```
alex$python3 Gen*
alex$wc 01.t
1120    1120 1004472 01.t
alex
alex$make
g++ suffix_tree.cpp main.cpp -std=c++11 -pedantic -Wall -Werror -Wno-sign-compare
-Wno-long-long -lm -o diskran_laba_5
alex$g++ -std=c++11 bench*
alex$time ./a.out <01.t >tmp

real    0m30.215s
user    0m30.204s
sys     0m0.000s
alex$time ./*5 <01.t >tmp
real    0m2.946s
user    0m2.556s
sys     0m0.308s
```

Как видно из теста производительности, суффиксное дерево быстрее чем наивный алгоритм. Разница колоссальная. Текст состоит из большого количества символов, шаблоны много меньше по длине. Но если длина паттерна будет, к примеру, 4-5 символов, то разницы во времени работы наших алгоритмов не будет. Это связано с тем, что наивный алгоритм поиска хорошо показывает себя, когда нам надо найти в большом тексте маленький фрагмент. Время работы будет линейно зависеть от длины исходного текста(длину паттерна в этом случае можно принять за константу). Однако при работе с большими образцами, время работы этого алгоритма возрастет до квадрата от длины текста(в худшем случае), в то время как суффиксное дерево будет работать линейно от длины шаблона! Так что, по моему мнению, нет абсолютно универсального и удобного метода поиска подстроки в строке. Нужно смотреть по ситуации, что в данном случае проще, удобнее и самое главное быстрее.

5 Выводы

Выполнив пятую лабораторную работу по курсу «Дискретный анализ», я познакомился с новым алгоритмом поиска подстрок в тексте – суффиксное дерево. Он крайне эффективен, если нам заранее известен текст. Мы подготавливаем структуру дерева для эффективного поиска любого количества образцов. Время построения суффиксного дерева – линейное. Время работы при поиске подстрок – тоже линейное, зависящее от их длин. Существует несколько реализаций построений суффиксных деревьев. Однако самым простым и эффективным является алгоритм Укконена. Его большой плюс в том, что нам не нужно хранить строку с текстом целиком. Мы считываем символ за символом и строим наше дерево. Это так называемый алгоритм реального времени(online algorithm). Его можно использовать в большом спектре задач, где нужна интерактивность и минимальное время отклика.

Список литературы

[1] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*

[2] *Суффиксное дерево – Википедия.*
URL: `https://ru.wikipedia.org/wiki/Суффиксное_дерево` (дата обращения: 01.10.2018).

[3] *Суффиксное дерево. Алгоритм Укконена*
URL: `http://e-maxx.ru/algo/ukkonen` (дата обращения: 01.10.2018).