Московский авиационный институт
(национальный исследовательский университет)

Институт информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторная работа №5 по курсу «Дискретный анализ»

Студент: И. С. Глушатов
Преподаватель: А. А. Кухтичев
Группа: М8О-207Б-19
Дата:
Оценка:
Подпись:

Москва, 2021

Лабораторная работа №5

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

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

Вариант №1:

Найти в заранее известном тексте поступающие на вход образцы.

1 Описание

Суффиксное дерево - это особая структура данных, содержащая все суффиксы исходной строки, по которой это дерево было построено. Такое дерево позволяет искать подстроку в строке за линейное время, пропорциональное длине заданной подстроки.

Свойства суффиксного дерева:

- Количество листьев равняется количеству букв в исходной строке
- У каждой внутренней вершины есть хотя бы два ребенка.
- Каждое ребро помечено подстрокой из исходной строки
- Никакие два ребра, выходящие из одной вершины, не имеют пометки, начинающиеся с одинакого символа
- Дерево содержит все суффиксы исходной строки, причем все они заканчиваются в листе и больше нигде
- При построении дерева созданный на каком-либо шаге лист всегда останется листом вплоть до окончания построения дерева.

2 Исходный код

Сначала, заметив, что в суффиксном дереве у вершины неизвестное число потомков, я решил, что данное дерево будет удобнее представлять в виде графа, поэтому вершины храняться в целостном массиве, где индекс в массиве является ее номером. Мы знаем, что число вершин в дереве не превосходит $2(n-1)$, поэтому мы сразу можем выделить память под максимальное количество вершин. Такое представление помогло меньше думать об утечках памяти и представлять все суффиксные и обычные ссылки как номера типа long.

В дереве дополнительно описан класс итератора, с помощью которого оно будет строится. Он имеет пять полей: вершина, в которой сейчас находится; индекс символа и сам символ, который вставляется; счетчек дополнительных вставок; и длинна подстроки (суффикса), которую нужно вставить.

Сама структура хранит указатель на исходную строку, массив вершин, глобальную переменную end для построения дерева и количество вершин.

```
1  namespace Tree {
2
3      class SufTree {
4
5      public:
6
7          class Node {
8              public:
9                  long l, r, index;
10                 long suf_link = 0;
11                 long str_index;
12                 std::unordered_map<char, long> next_vertexes;
13
14                 Node(const long _l = -1, const long _r = -1, const long _index = -1,
15                 const long _str_index = -1): l(_l), r(_r), index(_index),
16                 str_index(_str_index) {}
17
18                 long Length(const long end) const {
19                     return std::min(r, end) - l;
20                 }
21
22         };
23
24         class Iterator {
25             public:
26                 Node* active_node;
27                 long active_edge_index;
28                 long active_length;
29                 unsigned long remainder;
30                 char active_edge_char;
31
32                 Iterator(Node* _active_node, long _active_edge_index,
33                 char _active_edge_char, long _active_length, long _remainder):
34                 active_node(_active_node), active_edge_index(_active_edge_index),
35                 active_length(_active_length), remainder(_remainder),
36                 active_edge_char(_active_edge_char) {}
37
38                 void CreateEdge(const unsigned long node_count) {
39                     active_node->next_vertexes[active_edge_char] = node_count;
40                 }
41
42                 void FirstRule(const std::string* text) {
```

```
                active_edge_index++;
                if (active_edge_index < text->size())
                    active_edge_char = (*text)[active_edge_index];
                active_length--;
            }

            void ThirdRule(const Node* vertexes) {
                active_node = (Node*) vertexes + active_node->suf_link;
            }

    };

    long end = 0;
    const std::string* text;
    Node* vertexes;
    unsigned long node_count = 1;
};
```

В main'е сначала считывается текст, на его основе строится суффиксное дерево. Далее в цикле while считываются паттерны, для которых надо найти вхождения. В метод суффиксного дерева Find я подаю указатель на вектор, чтобы его заполнили номерами позиций, с которых паттерн начинается. После этого массив сортируется, выводится ответ и очищаетя для следующего запроса.

```
int main() {

    std::ios::sync_with_stdio(false);
    std::cin.tie(0);
    std::cout.tie(0);

    std::string text;
    std::cin >> text;
    text += '$';
    std::vector<long> occurrences;

    Tree::SufTree st(&text);

    /*-------------------------------------------*/

    std::string pattern;
    unsigned long pattern_index = 1;

    while (std::cin >> pattern) {
        st.Find(&pattern, &occurrences);

        if (occurrences.size() == 0) {
            pattern_index++;
            continue;
        }

        std::stable_sort(occurrences.begin(), occurrences.end());

        printf("%ld: ", pattern_index);
        for (size_t j = 0; j < occurrences.size() - 1; ++j) {
           printf("%ld, ", occurrences[j]);
        }

        printf("%ld\n", occurrences.back());

        pattern_index++;
        occurrences.clear();
```

```
38      }
39
40      return 0;
41  }
```

main.cpp	
Tree::SufTree::Node::Node (...)	Конструктор класса Node
long Tree::SufTree::Node::Length (const long end) const	Метод узла, возращающий длину ребра
Tree::SufTree::Iterator::Iterator (...)	Конструктор класса итератор
void Tree::SufTree::Iterator::CreateEdge (const unsigned long node_count)	Создает переход по таблице в вершину с номером node_count
void Tree::SufTree::Iterator::FirstRule (const std::string* text)	Первое правило для случая, когда активная вершина корневая
void Tree::SufTree::Iterator::ThirdRule (const Node* vertexes)	Третье правило для случая, когда активная вершина не корневая
Tree::SufTree::SufTree(const std::string* t)	Конструктор суффиксного дерева
void Tree::SufTree::FindIndexes(const Node* cur, std::vector<long>* r) const	Заполняет вектор r всеми индексами листьев поддерева с корнем cur
bool Tree::SufTree::Find(const std::string* pattern, std::vector<long>* r)	Ищет все вхождения паттерна и заносит их в вектор r
Tree::SufTree::~SufTree()	Деструктор

3 Консоль

```
igor@igor-Aspire-A315-53G:~/Рабочий стол/c++/DA/lab5$ ./main2.out
abcdabc
abcd
1: 1
bcd
2: 2
bc
3: 2,6
igor@igor-Aspire-A315-53G:~/Рабочий стол/c++/DA/lab5$
```

4 Тест производительности

Для тестов я использвал утилиту gnuplot для построения графиков зависимости времени построения дерева от количества букв в тексте. Так же для сравнения использовал библиотеку chrono для замера времени.

Я приведу графики зависимости времени построения дерева от количества букв в тексте. Изначально в конструкторе Node я создавал таблицу переходов для всех букв латинского алфавита. Однако на ejudge программа не проходила из-за тайм лимита. Я попробовал добавлять в таблицу пары по мере построения дерева, и результат оказался лучше некуда, так как скорость построения дерева увеличилось примерно в 10 раз, хотя и ассимптотика никак не изменилась.

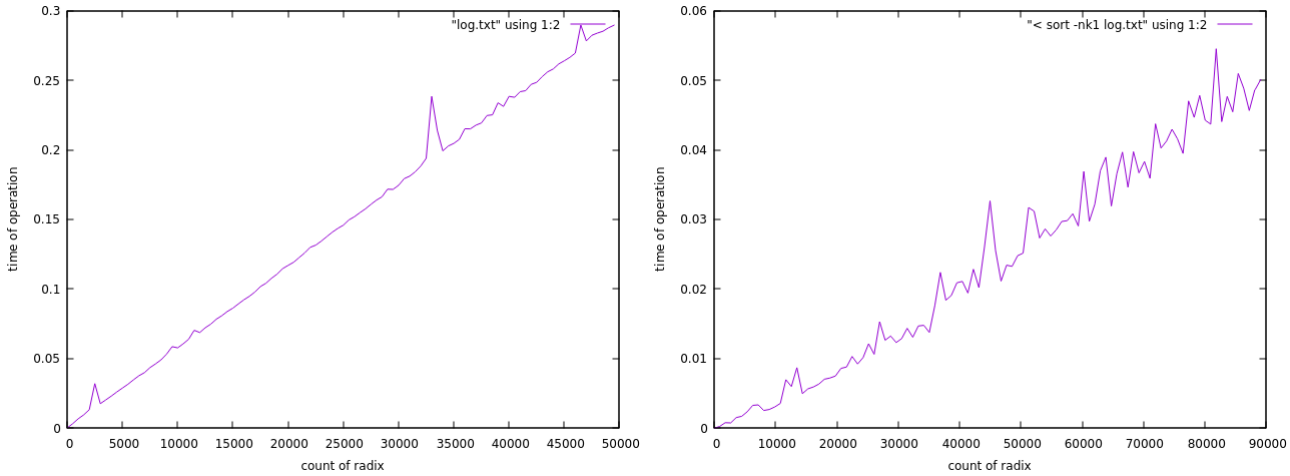

Рис. 1: Зависимость времени построения дерева от длины текста до и после исправления

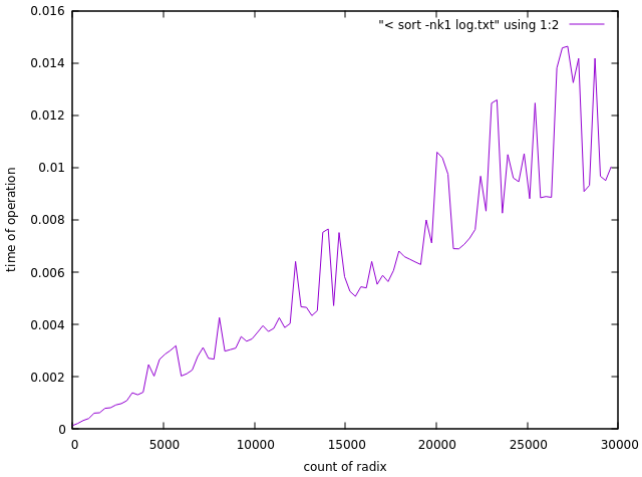

Рис. 2: Зависимость времени поиска от длины паттерна

5 Выводы

В ходе пятой лабораторной работы я узнал еще один из многочисленных способов поиска подстроки в строке с препроцессингом. Реализовал алгоритм Укконена, позволяющий построить суффиксное дерево за линейную сложность. Сначала было непонятно, с чего вообще начать, но обратив внимание на тот факт, что дерево можно представить графом, рассмотрев англоязычные статьи с псевдокодами и визуализатор алгоритма, я понял, что именно нужно делать. Обрадовало небольшое количество операций с ссылками, из-за чего программе потребовалось очень мало времени на дебаг утечек памяти.

Список литературы

[1] *Поисковик - Google.*
URL: https://www.google.com/

[2] *Сайт с подробной документацией библиотек C++*
URL: https://en.cppreference.com/

[3] *Про алгоритм Укконена*
URL: https://habr.com/ru/post/533774/

[4] *Построение дерева*
URL: http://brenden.github.io/ukkonen-animation/

[5] *Про алгоритм Укконена*
URL: https://users.math-cs.spbu.ru/ okhotin/teaching/tcs2_2019/ okhotin_tcs2alg_2019_l3.pdf

[6] *Про алгоритм Укконена*
URL: https://www.cs.helsinki.fi/u/ukkonen/SuffixT1withFigs.pdf