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: К. С. Саженов
Преподаватель: И. Н. Симахин
Группа: М8О-308Б-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 4: Линеаризовать циклическую строку, то есть найти минимальный в лексикографическом смысле разрез циклической строки.

Входные данные: Некий разрез циклической строки.

1 Описание

Делаем строку на вход алгоритму Укконена удвоенной копией исходной строки.

Алгоритм Укконена реализован следующим образом: Каждый узел содержит итераторы, указывающие на начало и конец этой подстроки в тексте, суффиксную ссылку, которая либо указывает на вершину с таким же суффиксом как и в этой, только без первого символа, либо при отсутствии такой вершины – на корень.

Также есть словарь с ребрами, выходящими из данной вершины.

В дереве храним текст (в конце которого терминальный символ), по которому ищем, указатель на корень, переменную remainder, которая показывает, сколько суффиксов еще надо вставить. Указатель lastNode указывает на вершину, из которой необходимо создать суффиксную ссылку, если в данной фазе уже была вставлена вершина по 2 правилу продолжений, и сейчас оно используется вновь. Указатель activeNode указывает на вершину, которое имеет ребро activeEdge, в котором мы сейчас находимся. activeEdgeLength показывает на каком расстоянии от этой вершины мы находимся (сколько символов пропустить, чтобы попасть в нужный). При создании дерева итеративно проходим по тексту.

На каждой итерации начинается новая фаза и remainder увеличивается на 1. Далее, пока все невставленные суффиксы не вставлены в дерево, выполняем цикл.

Если в той вершине, в которой мы остановились еще нет ребра, начинающегося с первой буквы обрабатываемого суффикса, то по 1 правилу продолжений создаем новую вершину, которая будет листом.

Если это необходимо, создаем суффиксную ссылку (если до этого в этой фазе была создана вершина по 2 правилу продолжений).

Если в той вершине, в которой мы остановились, уже есть такое ребро, то нужно пройти вниз по ребрам на activeEdgeLength и обновить activeNode.

Если некоторый путь на этом ребре начинается со вставляемого символа, значит по 3 правилу продолжений нам ничего делать не надо, заканчиваем фазу, оставшиеся суффиксы будут добавлены неявно.

Увеличиваем activeEdgeLength на 1 (т.к. учитываем, что этот символ уже есть на данном пути), по необходимости строим суффисную ссылку.

Если никакой путь не начинается со вставляемого символа, то нужно разделить ребро в этом месте, вставив 2 новых вершины – одну листовую и одну разделяющую ребро.

Далее по необходимости добавляем суффиксную ссылку. Уменьшаем remainder на 1, если вставили суффикс в цикле.

Если после всех этих действий activeNode указывает на корень и activeLen больше 0, то уменьшаем activeLen на 1, а activeEdge устанавливаем на первый символ нового суффикса, который нужно вставить. Если activeNode не корень, то переходим по

суффиксной ссылке.

После конструирования дерева, проходим по дереву рекурсивно для поиска минимальной строки длины исходной строки – это и будет минимальная линеаризация циклической строки.

2 Исходный код

Файл `main.cpp`:

```
#include "suffixtree.h"

#include <iostream>
#include <string>

int main() {
    std::string s;
    std::cin >> s;

    const TSuffixTree suffixTree(s+s);

    std::cout << suffixTree.FindMinimumString(s.size()+1).substr(0, s.size()) << std::
        endl;
// std::cerr << s+s << std::endl;
// std::cerr << suffixTree;

    return EXIT_SUCCESS;
}
```

Файл `suffixtree.h`:

```
//
// Created by sakost on 19.06.2021.
//

#ifndef LAB5_SUFFIXTREE_H
#define LAB5_SUFFIXTREE_H

#include <string>
#include <map>
#include <iostream>


const char TERMINATE_LEAF_SYMBOL = '$';

class TSuffixTree;
class TNode;

class TNode{
public:
    friend class TSuffixTree;
    friend std::ostream& operator<<(std::ostream&, const TSuffixTree&);

    TNode(TNode* link, std::size_t start, std::size_t* end) : suffix_link(link),
                                                              start(start),
                                                              end(end)
```

```
26      {}
27
28      std::size_t size() const {
29          if(end == nullptr){
30              return 0;
31          }
32          return (*end) - start + 1;
33      }
34
35      ~TNode(){
36          if(end != nullptr && *end != -1){
37              delete end;
38          }
39          suffix_link = nullptr;
40          for(auto &child: children){
41              delete child.second;
42          }
43      }
44
45  private:
46      void RecursiveDisplay(std::ostream& out, std::size_t depth, const std::string& str)
            const;
47      std::map<char, TNode*> children;
48      TNode* suffix_link = nullptr;
49      std::size_t start;
50      std::size_t* end = nullptr;
51  };
52
53  class TSuffixTree {
54  public:
55      friend std::ostream& operator<<(std::ostream&, const TSuffixTree&);
56
57      explicit TSuffixTree(const std::string& str);
58      explicit TSuffixTree(std::string&& str);
59      void Build();
60      std::string FindMinimumString(std::size_t n) const;
61
62      ~TSuffixTree();
63
64  private:
65      std::string FindMinimumStringRecursive(const TNode* &node, std::size_t n) const;
66
67      void AppendSuffixesOfPrefix(std::size_t pos);
68
69      inline void UpdateLastNodeLink(TNode*& node);
70
71
72
73      TNode* root = nullptr;
```

```
74      TNode* lastNode = nullptr;
75
76      std::string m_str;
77
78      TNode* activeNode = nullptr;
79
80      std::size_t* globalEnd = nullptr;
81
82      std::size_t activeEdge = 0;
83      std::size_t activeEdgeLength = 0;
84      std::size_t remainder = 0;
85
86  };
87
88
89
90  #endif //LAB5_SUFFIXTREE_H
```

3 Консоль

```
~/university/2 course/diskran/lab5/cmake-build-debug on master
$ cmake ../
--The C compiler identification is AppleClang 13.0.0.13000029
--The CXX compiler identification is AppleClang 13.0.0.13000029
--Detecting C compiler ABI info
--Detecting C compiler ABI info -done
--Check for working C compiler: /Applications/Xcode.app/Contents/Developer/Toolchains/
-skipped
--Detecting C compile features
--Detecting C compile features -done
--Detecting CXX compiler ABI info
--Detecting CXX compiler ABI info -done
--Check for working CXX compiler: /Applications/Xcode.app/Contents/Developer/Toolchain
-skipped
--Detecting CXX compile features
--Detecting CXX compile features -done
--Configuring done
--Generating done
--Build files have been written to: /Users/k.sazhenov/university/2 course/diskran/lab5

~/university/2 course/diskran/lab5/cmake-build-debug on master
$ cmake --build .
[ 12%] Building CXX object CMakeFiles/suffixtree.dir/suffixtree.cpp.o
[ 25%] Linking CXX static library libsuffixtree.a
[ 25%] Built target suffixtree
[ 37%] Building CXX object CMakeFiles/naive.dir/naive.cpp.o
[ 50%] Linking CXX static library libnaive.a
[ 50%] Built target naive
[ 62%] Building CXX object CMakeFiles/lab5_bench.dir/benchmark.cpp.o
[ 75%] Linking CXX executable lab5_bench
[ 75%] Built target lab5_bench
[ 87%] Building CXX object CMakeFiles/lab5.dir/main.cpp.o
[100%] Linking CXX executable lab5
[100%] Built target lab5

~/university/2 course/diskran/lab5/cmake-build-debug on master
$ ./lab5
xabcd
abcdx
```

```
~/university/2 course/diskran/lab5/cmake-build-debug on master
$
```

4 Тест производительности

Тест производительности будет производиться на рандомно сгенерированной строке, состоящей из 90 000 символов латинского алфавита и на строке, состоящей из 1 000 000 символов латинского алфавита.

```
~/university/2 course/diskran/lab5/cmake-build-debug on master
$ ./lab5_bench < ../input.txt
Naive algorithm: 0.717206
Suffix tree: 0.231178

~/university/2 course/diskran/lab5/cmake-build-debug on master
$ ./lab5_bench < ../input_big.txt
Naive algorithm: 81.1535
Suffix tree: 3.73507

~/university/2 course/diskran/lab5/cmake-build-debug on master
$
```

Как видно из тестов – с ростом количества символов растет и отрыв в производительности алгоритма с суффиксным деревом от наивного алгоритма поиска.

5 Выводы

Суффиксное дерево в некоторых ситуациях может занимать слишком много памяти, чтобы оказаться практичным в некоторых приложениях. На малых алфавитах логичнее было бы использовать вектор для хранения ребер. Это позволило бы иметь константный доступ к ребрам и не очень бы увеличило используемую память. Использование словаря в моей реализации – это компромисс между местом и скоростью. Добавление дуги и поиск требуют ($O(log(k))$) времени и $O(k)$ памяти, где k – число дочерних вершин из данной вершины. Вообще говоря, использование словаря осмысленно только при действительно больших k. Но т.к. в задании было сказано, что алфавит – это строчные буквы английского алфавита, я решил, что он достатоно большой, и поэтому использовала map. Хэш-таблицу я не стал использовать, потому что мне было важно, чтобы контейнер, в котором хранятся дуги, был отсортирован. Это позволило мне при нахождении минимальной линеаризации циклической строки быстро находить ребра с минимальным первым значением, ничего специально не сортируя.

В итоге, временная оценка построения суффиксного дерева – $O(m * log(k))$, где m – длина текста, k – размер алфавита. Поиск линеаризации циклической строки работает за $O(2 * m * log(k)) = O(m * log(k))$, где m – длина текста.

Суффиксное дерево широко применимо в разных областях. С его помощью можно найти, например: количество различных подстрок данной строки, наибольшую общую подстроку двух строк, суффиксный массив и массив lcp исходной строки, статистику совпадений. Все эти задачи решаются за линейное время благодаря алгоритму Укконена и, собственно, суффиксному дереву.

Список литературы

[1] Т. Кормен, Ч Лейзерсон, Р Ривест, К Штайн. *Алгоритмы: построение и анализ, 3-е издание.* — Издательский дом «Вильямс», 2013. Перевод с английского: ООО «И.Д. Вильямс» — 1328 с. (ISBN 978-5-8459-1794-2 (рус.))

[2] *Алгоритм Укконена – Викиконспекты*
URL: `https://neerc.ifmo.ru/wiki/index.php?title=Алгоритм_Укконена` (дата обращения: 17.10.2021).