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: Е. А. Суханов
Преподаватель: А. А. Кухтичев
Группа: М8О-306Б
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: Найти самую длинную общую подстроку двух строк.

Формат входных данных Две строки.

Формат результата На первой строке нужно распечатать длину максимальной общей подстроки, затем перечислить все возможные варианты общих подстрок этой длины в порядке лексикографического возрастания без повторов.

1 Описание

Решение задачи состоит из двух частей.

1. Нужно построить суффиксное дерево для конткантинации строк. Для этого я буду использовать алгоритм Укконена;

2. Проходимся по суффиксному дереву с помощую обхода в глубину.

Алгоритм Укконена:
Алгоритм делится на итерации. На каждой итерации нужно к существующему дереву добавить следующий символ c.

1. Сначала увеличиваем количество суффиксов, которые нужно будет вставить, если это количество равно 1, то значит, мы в корне и следующее ребро должно начинаться с c;

2. Затем мы ищем, есть ли какое-либо ребро, начинающееся со буквы, на которой мы остановились;

3. Если такое ребро есть, но мы пришли к его концу, то переходим в следующее ребро;

4. Затем проверяем, совпала ли наша буква с буквой, которая находится следующей в ребре. Если совпала, то переходим к следующей букве текста;

5. Если же не совпала то нам необходимо разбить узел на 2 узла, а затем рассмотреть узел, доступный по суффиксной ссылке.

После построения суффиксного дерева, нам нужно пройтись обходом в глубину.

1. Находим самые длинные пути из корня в вершину, которая удовлетворяет следующему условию:

2. Из этой вершины можно добраться до вершины, которая является "листом"для первого текста, а также для вершины, которая является "листом"для второго текста, но не включает первый текст.

3. Такие пути сохраняем в векторе, а затем выводим их.

2 Исходный код

Заголовочный файл tree.hpp:

```
1  namespace NSuffixTree {
2
3  struct TNode {
4      std::map<char, TNode*> Next;
5      TNode* Suff;
6
7      int Begin;
8      int End;
9
10     TNode(int begin, int end);
11     ~TNode();
12 };
13
14 class TSuffixTree {
15 public:
16     std::string Text;
17     TNode* Root;
18     TNode* CurrNode;
19     int End;
20     int Rem;
21     int CurrLen;
22     int CurrEdge;
23
24     void Solve();
25     TSuffixTree(const std::string& textA, const std::string& textB);
26     ~TSuffixTree();
27 private:
28     int IdxFirstSep;
29     int dfs(std::vector<std::pair<int,int>>& ans, int& maxLen, TNode* node, int len,
           int begin);
30     void Iterate(int i);
31 };
32
33 } // namespace NSuffixTree
```

Реализация алгоритма Укконена:

```
TSuffixTree::TSuffixTree(const std::string& textA, const std::string& textB) {
    Root = new TNode(0, 0);
    Root->Suff = Root;

    CurrNode = Root;
    CurrLen = 0;
    CurrEdge = 0;
    Rem = 0;
    IdxFirstSep = textA.size();

    Text = textA + FIRST_SEP + textB + SECOND_SEP;
    End = 0;
    for(int i = 0; i < (int)Text.length(); i++){
        Iterate(i);
        End++;
    }
}


void TSuffixTree::Iterate(int i){
    TNode* prevInsertedNode = nullptr;
    Rem++;

    if (Rem == 1) {
        CurrEdge = i;
    }

    while (Rem > 0) {
        auto nextIt = CurrNode->Next.find(Text[CurrEdge]);
        TNode* nextNode = (nextIt != CurrNode->Next.end() ? nextIt->second : nullptr);

        if (nextNode == nullptr) {
            CurrNode->Next[Text[CurrEdge]] = new TNode(i, -1);
            if (prevInsertedNode != nullptr) {
                prevInsertedNode->Suff = CurrNode;
            }
            prevInsertedNode = CurrNode;
        } else {
            int edgeLen = nextNode->End == -1 ? End - nextNode->Begin + 1 : nextNode->
                End - nextNode->Begin;

            if (CurrLen >= edgeLen) {
                CurrNode = nextNode;
                CurrLen -= edgeLen;
                CurrEdge += edgeLen;
                continue;
            }

```

```
48              if (Text[nextNode->Begin + CurrLen] == Text[i]) {
49                  CurrLen++;
50                  if (prevInsertedNode != nullptr) {
51                      prevInsertedNode->Suff = CurrNode;
52                  }
53                  break;
54              }
55
56              TNode* midNode = new TNode(nextNode->Begin, nextNode->Begin + CurrLen);
57              CurrNode->Next[Text[CurrEdge]] = midNode;
58              midNode->Next[Text[i]] = new TNode(i, -1);
59              nextNode->Begin += CurrLen;
60              midNode->Next[Text[nextNode->Begin]] = nextNode;
61
62              if (prevInsertedNode != nullptr) {
63                  prevInsertedNode->Suff = midNode;
64              }
65              prevInsertedNode = midNode;
66          }
67
68
69          if (CurrNode == Root && CurrLen > 0) {
70              CurrEdge++;
71              CurrLen--;
72          }else if (CurrNode != Root) {
73              CurrNode = CurrNode->Suff;
74          }
75
76          Rem--;
77      }
78  }
```

3 Консоль

```
$ make
g++ -c -Wall -pedantic -std=c++14 -O2 main.cpp -o main.o -O2
g++ -c -Wall -pedantic -std=c++14 -O2 tree.cpp -o tree.o -O2
g++ -O2 main.o tree.o -o solution
$ ./solution
xabay
xabcbay
3
bay
xab
```

4 Тест производительности

Мое решение я буду сравнивать с наивным. На случайных данных.

Два текста, длиной 100:

```
$ python3 ../generated_tests/generator.py && ./benchmark <randomtest.txt
My Solution : 67us
Default Solution : 144us
```

Два текста, длиной 200:

```
$ python3 ../generated_tests/generator.py && ./benchmark <randomtest.txt
My Solution : 141us
Default Solution : 897us
```

Из тестов видно, что наивное решение (перебор вариантов), работает куда хуже, чем алгоритм Укконена. Наивное решение имеет сложность $O(n^3)$, тогда как решение с алгоритмом Укконена $O(n)$

5 Выводы

Выполнив пятую лабораторную работу по курсу "Дискретный анализ я изуил и написал алгоритм Укконена, Вспомнил структуру данных trie. А так же работу с графами в целом. Суффиксные деревья имеют много приложений.

Список литературы

[1] *Алгоритм Укконена: от простого к сложному*
URL: `https://habr.com/ru/post/533774/` (дата обращения: 24.11.2021)