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: Л. Я. Вельтман
Преподаватель: А. А. Кухтичев
Группа: М8О-207Б

Дата:
Оценка:
Подпись:

Москва, 2019

Лабораторная работа №5

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

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

Вариант 1: Поиск в известном тексте неизвестных заранее образцов
Наити в заранее известном тексте поступающие на вход образцы.
Входные данные: текст располагается на первои строке, затем, до конца фаила, следуют строки с образцами.
Выходные данные: для каждого образца, наиденного в тексте, нужно распечатать строчку, начинающуюся с последовательного номера этого образца и двоеточия, за которым, через запятую, нужно перечислить номера позиции, где встречается образец в порядке возрастания. **Вариант алфавита:** строчные буквы латинского алфавита (т.е., от a до z).

1 Описание

Для реализации поиска заранее неизвестного образца в тексте нужно применить алгоритм Укконена. Каждый узел содержит итераторы, которые указывают на начало и конец этой подстроки в тексте, суффиксную ссылку, которая указывает на вершину с таким же суффиксом только без первого символа, если такой суффикс отсутствует, то суффиксная ссылка указывает на корень. Есть словарь с ребрами, выходящими из данной вершины. В дереве будет храниться текст (в конце стоит терминальный символ \$), по которому происходит поиск образца, root - указатель на корень, remainder, который показывает количество суффиксов необходимых для вставки. Указатель sufflinkNeed указывает на вершину, из которой необходимо создать суффиксную ссылку, если в данной фазе уже была вставлена вершина по 2 правилу, и сейчас оно используется вновь. Указатель activeNode указывает на вершину, которая имеет ребро activeEdge, в котором мы сейчас находимся. activeLength показывает на каком расстоянии от этой вершины мы находимся (нужное количество символов для пропуска, чтобы попасть в нужный). Итеративно проходим по тексту при создании дерева. На каждой итерации начинается новая фаза и remainder инкрементируется. Пока все оставшиеся суффиксы не вставлены в дерево, выполняется цикл. Если в вершине, в которой произошла остановка, еще нет ребра, начинающееся с первой буквы обрабатываемого суффикса, то по 1 правилу создаем суффиксную ссылку (если в этой фазе уже была создана вершина по 2 правилу). Если в вершине, в которой остановились, уже есть такое ребро, то нужно пройти вниз по ребрам на activeLength и обновить activeNode. Если путь на этом ребре начинается со вставляемого символа, значит по 3 правилу не нужно ничего делать, завершаем фазу, оставшиеся суффиксы добавятся неявно. Инкрементируем activeLength, при необходимсоти строим суффиксную ссылку. Если нет пути, который начинается со вставляемого символа, то нужно разделить ребро в этом месте, вставив 2 новые вершины - одну листовую и одну разделяющую ребро. При необходимости добавляется суффиксная ссылка. Декрементируем remainder, если вставили суффикс в цикле. Если после всех вышеперечисленных действий activeNode указывает на корень и activeLength > 0, то декрементируем activeLength, activeEdge устанавливаем на первый символ нового суффикса, который нужно вставить. Если activeNode не является корнем, то переходим по суффиксной ссылке. После конструирования дерева проиндексируем каждое ребро дерева. Это понадобится для поиска вхождений образца в текст. В векторе ind находятся начальные позиции суффиксов, и все эти суффиксы упорядочены.

2 Исходный код

```
1  const char term = '$';
2
3  class TNode {
4  public:
5      std::map<char, std::shared_ptr<TNode>> to_edge;
6      std::string::iterator begin, end;
7      std::shared_ptr<TNode> suffixLink;
8      int index;
9      TNode(std::string::iterator begin, std::string::iterator end) : to_edge(), begin(
           begin), end(end), index(0) {};
10     ~TNode() {};
11 };
12
13 class TSuffixTree {
14 public:
15     TSuffixTree(const std::string& str);
16     void Search(const std::string&, size_t);
17     ~TSuffixTree() {};
18 private:
19     std::string text;
20     std::shared_ptr<TNode> root;
21     size_t remainder;
22     std::shared_ptr<TNode> sufflinkNeed;
23     std::shared_ptr<TNode> activeNode;
24     size_t activeLength;
25     std::string::iterator activeEdge;
26
27
28     size_t EdgeLength(std::shared_ptr<TNode>, std::string::iterator);
29     void CountIndex(std::shared_ptr<TNode>, size_t);
30     void GetEntries(std::shared_ptr<TNode> current_node, std::vector<size_t>& entries);
31     void ExtendTree(std::string::iterator);
32     void AddSuffLink(std::shared_ptr<TNode>);
33     bool WalkDown(std::string::iterator, std::shared_ptr<TNode>);
34 };
```

3 Консоль

```
MacBook-Pro-Lina:da5 linuxoid$ make
g++ -std=c++11 -Wall -Werror -Wno-sign-compare -Wno-unused-result -O3 -o da5
da5.cpp TSuffTree.cpp
MacBook-Pro-Lina:da5 linuxoid$ cat test1
bccabfedfc
baf
efc
ceeb
febeed
c
ea
bebdd
fdbfceddeb
MacBook-Pro-Lina:da5 linuxoid$ ./da5 <test1
5: 2,3,10
MacBook-Pro-Lina:da5 linuxoid$ cat test2
fbfdedcbaccebccadfa
dfa
fdbdbbd
ecadacfa
adbdff
MacBook-Pro-Lina:da5 linuxoid$ ./da5 <test2
1: 17
MacBook-Pro-Lina:da5 linuxoid$ cat test3
ddbcecbdbfadcdbdeeedaaeadfccef
aefbabfdfa
adeaaadfdd
fbcaaaedea
cdcad
cddbed
fbfdfabdd
decdcbd
eabace
eceaffbfd
ebdcebf
afbcfe
efeeeec
f
df
```

```
e
dcce
fefd
dab
MacBook-Pro-Lina:da5 linuxoid$ ./da5 <test3
13: 10,26,30
14: 25
15: 5,17,18,19,23,29
```

4 Тест производительности

Для теста производительности я использую метод find в std::string из STL для сравнения со своим поиском в суффискном дереве в известном тексте неизвестных заранее образцов. В алгоритм find передается две точки: старт и конец поиска. Между точками находится диапазон, в котором находятся значения. Если встретился элемент, который равен третьему аргументу, то выполняется какое-то условие. Выходит, что этот метод работает за линейное время, так как просматривает символ в тексте.

Тест производительности представляет из себя следующее: 1 тест - 100 000 образцов. 2 тест - 500 000 образцов. 3 тест - 1 000 000 образцов. В каждом тесте текст состоит из 1000 случайных символов a, b, c, d, e, f. Затем идут образцы длиной от 1 до 10 символов данного алфавита.

```
MacBook-Pro-Lina:da4 linuxoid$ ./da5 <test1
Suffix tree search time is: 0.156312 seconds
std::find search time is: 0.21501 seconds

MacBook-Pro-Lina:da4 linuxoid$ ./da5 <test2
Suffix tree search time is: 0.95423 seconds
std::find search time is: 1.23439 seconds

MacBook-Pro-Lina:da4 linuxoid$ ./da5 <test3
Suffix tree search time is: 1.50101 seconds
std::find search time is: 2.19342 seconds
```

Анализируя время выполнения данных тестов можно сделать вывод, что суффиксное дерево показало хороший результат по времени как на маленьких, так и на больших данных. Это произошло из-за того, что поиск в дереве линеен относительно длин образцов, а поиск с помощью find линеен относительно текста.

5 Выводы

Таким образом, выполнив данную лабораторную работу, я научилась решать задачу поиска образца в заранее известном тексте.

В итоге, временная оценка построения суффиксного дерева O(m*logs), где m - длина текста, s - размер алфавита. Все вхождения образца в текст находятся за O(n*logk), где n - длина образца, k - количество ребёр суффиксного дерева.

Несмотря на то, что данный алгоритм является одним из самых простых в понимании алгоритмов для построения суффиксных деревьев и использует online подход, у него есть серьёзные недостатки, из-за которых его нечасто используют на практике. Размер суффиксного дерева сильно превосходит входные данные, поэтому при очень больших входных данных алгоритм Укконена сталкивается с проблемой memory bottleneck problem(другое её название thrashing). Также алгоритм предполагает, что дерево полностью должно быть загружено в оперативную память. Если же требуется работать с большими размерами данных, то становится не так просто модифицировать алгоритм, чтобы он не хранил всё дерево в ней.

Суффиксные деревья имеют широкую область применения. С помощью него можно найти количество различных подстрок данной строки, наибольшую общую подстроку двух строк, суффиксный массив, статистику совпадений. Все вышеприведенные задачи решаются за линейное время.

Список литературы

[1] *Построение суффиксного дерева: алгоритм Укконена.*
URL: `https://habr.com/post/111675/`.
(дата обращения: 29.12.2018).

[2] *Ukkonen's suffix tree algorithm .*
URL: `https://stackoverflow.com/questions/9452701`.
(дата обращения: 29.12.2018).

[3] *Visualization of Ukkonen's Algorithm.*
URL: `http://brenden.github.io/ukkonen-animation/`.
(дата обращения: 29.12.2018).