Московский авиационный институт (национальный исследовательский университет)

Факультет прикладной математики и физики

Кафедра вычислительной математики и программирования

Лабораторная работа №5 по курсу «Дискретный анализ»

Студентка: А. Довженко
Преподаватель: Д. Е. Ильвохин
Группа: 08-207

Дата:
Оценка:
Подпись:

Москва, 2017

Лабораторная работа №5

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 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 строку, начинающуюся с последовательного номера этого образца и двоеточия, за которым, через запятую, нужно перечислить номера позиций, где встречается образец в порядке возрастания.

1 Описание

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 needSL 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. activeLen 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 activeLen 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 activeLen 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, строим суффиксный массив. В нем расположен вектор, в котором находятся начальные позиции суффиксов, и все эти суффиксы лексикографически упорядочены. Массив строим из дерева, выполняя обход в глубину. Т.к. словарь, который находится в каждой вершине, это упорядоченный контейнер, то номера позиции после обхода в глубину будут также лексикографически упорядочены.

Поиск вхождений в массиве осуществляется с помощью бинарного поиска. В этом мне помогла библиотечная функция equal range. В зависимости от того, лексикографически меньше или больше буква в паттерне и буква в тексте, границы поиска в массиве сужаются наполовину. В конце equal range возвращает диапазон начальных позиций, в которых найдены вхождения.

2 Исходный код

```
class TNode
{
public:
    std::map<char, TNode *> to;
    std::string::iterator begin, end;
    TNode *suffixLink;
    TNode(std::string::iterator begin, std::string::iterator end);
    ~TNode() {};
};

class TSuffixTree
{
public:
    TSuffixTree(std::string str);
    void TreePrint();
    ~TSuffixTree() {};
    friend TSuffixArray;

private:
    std::string text;
    TNode *root;
    int remainder;
    TNode *needSL, *activeNode;
    int activeLen;
    std::string::iterator activeEdge;

    void NodePrint(TNode *node, int dpth);
    int EdgeLen(TNode *node, std::string::iterator pos);
    void TreeDestroy(TNode *node);
    bool WalkDown(std::string::iterator cur_pos, TNode *node);
    void SLAdd(TNode *node);
    void TreeExtend(std::string::iterator add);
    void DFS(TNode *node, std::vector<int> &result, int deep);
};

class TSuffixArray
{
public:
    TSuffixArray(TSuffixTree tree);
    std::vector<int> Find(std::string pattern);
    ~TSuffixArray() {};
private:
    std::string text;
    std::vector<int> array;
};
```

3 Тест производительности

Я случайным образом генерирую три файла, которые представляют из себя следующее:

1 тест – 100 000 образцов.
2 тест – 375 000 образцов.
3 тест – 1 000 000 образцов.

В каждом тесте текст состоит из 1000 случайных символов a, b, c, d, e, f. Далее идут образцы длиной от 1 до 10 символов выбранного алфавита.

Сравнение я буду проводить с поиском образца методом find в std::string, который использует линейный поиск, т.е. проверяет каждый символ в тексте, а также с поиском в суффиксном дереве.

```
karma@karma:~/mai_study/DA/lab5$ cat input100k | ./da5
Suffix array search time is: 0.09593 sec.
Suffix tree search time is: 0.155269 sec.
std::find search time is: 0.21282 sec.
karma@karma:~/mai_study/DA/lab5$ cat input375k | ./da5
Suffix array search time is: 0.341001 sec.
Suffix tree search time is: 0.555375 sec.
std::find search time is: 0.865759 sec.
karma@karma:~/mai_study/DA/lab5$ cat input1kk | ./da5
Suffix array search time is: 0.907325 sec.
Suffix tree search time is: 1.49605 sec.
std::find search time is: 2.18262 sec.
```

В результате видим выигрыш суффиксного массива на всех тестах. Суффиксное дерево и массив показали себя лучше, потому что поиск в массиве и дереве линеен относительно длин образцов, а поиск с помощью find линеен относительно текста. Массив выиграл у дерева, потому что при поиске в дереве мы сначала находим вхождение образца, а потом используем поиск в глубину, чтобы узнать все позиции вхождений, а в массиве поиск в глубину происходит на этапе препроцессинга, а не в самом поиске для каждого образца. Бинарный поиск в массиве выполняется за $O(\log m)$, где m – длина текста, а поиск в глубину имеет сложность $O(V + E)$, где E – количество ребер, V – количество вершин.

4 Отладка

1 посылка на чекер: Compilation error.
Инициализация в конструкторе класса была не в том порядке, в котором поля объявлены в классе. Исправлено.
2 посылка на чекер: Compilation error.
Вместо класса TNode в одном месте программы было написано Node. Исправлено.
3 посылка на чекер: Wrong answer.
Забыла закомментировать отладочную информацию. Закомментировала.
4 посылка на чекер: Wrong answer.
Была проблема с поиском в дереве. Номера вхождений были не в том порядке. Исправлено сортировкой полученных номеров вхождений.
5 посылка на чекер: OK.
6 посылка на чекер: OK.
В пятой посылке поиск осуществлялся не с помощью суффиксного массива, а с помощью дерева. Это было сделано намеренно, с целью проверить, хватает ли мне места и работает ли программа корректно. Как видим, проверка показала, что мое суффиксное дерево успешно проходит чекер, поэтому в 6 посылке был добавлен только суффиксный массив и поиск в нем для реализации задания варианта.
7 посылка: OK.
Были устранены утечки памяти.

5 Выводы

Суффиксное дерево в некоторых ситуациях может занимать слишком много памяти, чтобы оказаться практичным в некоторых приложениях. В таких случаях лучше использовать суффиксный массив, который очень рационально использует память. Конвертация из суффиксного дерева в суффиксный массив происходит за линейное время. Также, как мы увидели в бенчмарке, поиск образцов в моей реализации происходит быстрее в суффиксном массиве.

На малых алфавитах, например как у меня в бенчмарке, 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 решила, что он достатоно большой, и поэтому использовала map. Хэшированный словарь (unordered map) я не стала использовать, потому что мне было важно, чтобы контейнер, в котором хранятся дуги, был отсортирован. Это позволило мне при конструировании суффиксного массива просто совершить обход в глубину и сразу получить лексикографически упорядоченные суффиксы, ничего специально не сортируя.

В итоге, временная оценка построения суффиксного дерева – $O(m * \log k)$, где m – длина текста, k – размер алфавита. Все вхождения образца в текст находятся с помощью суффиксного массива за $O(n*\log m)$, где n – длина образца, 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. Хорошо, что мы изучили суффиксное дерево.