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: В. В. Косогоров
Преподаватель: А. А. Кухтичев
Группа: М8О-206Б

Дата:
Оценка:
Подпись:

Москва, 2020

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.
Вариант алгоритма: Найти в заранее известном тексте поступающие на вход образцы с использование суффиксного массива.
Вариант алфавита: Строчные буквы латинского алфавита.

1 Описание

Требуется написать реализацию алгоритма Укконена построения суффсиксного дерева за линейной время и, создав на его основе суффиксный массив, произвести поиск образца в массиве.

Время $O(m)$ на препроцессинг и $O(n)$ на поиск − это удивительный и крайне полезный результат. Такая оценка не достигается методами Кнута-Морриса-Пратта и Бойера-Мура; они предварительно обрабатывают каждую строку на вводе, а затем завтрачиваю в худшем случае время $O(m)$ на поиск подстроки в тексте. Так как m может быть огромно по сравнению с n, эти алгоритмы будут непрактичны для любых текстов нетривиального размера[1].

Алгоритм Укконена строит неявное суффиксное дерево T_i для каждого префикса $S[l..i]$ строки S, начиная с T_1 и увеличивая i на единицу, пока не будет построено T_m. Настоящее суффиксное дерево для S получается из T_m, и вся работа требует времени $O(m)$. Алгоритм Укконена обычно объясняют, представив сначала метод, с помощью которого все деревья строятся кубическое время, а затем оптимизируют реализацию этого метода так, что будет достигнута заявленная линейная скорость.

Пусть $S[j..i] = b$ — суффикс $S[1..i]$. В продолжении j, когда алгоритм находит конец b в текущем дереве, он продолжает b, чтобы обеспечить присутствие суффикса $bS(i+1)$ в дереве. Алгоритм действует по одному из следующих трех правил.

1. В текущем дереве путь b кончается в листе. Это значит, что путь от корня с меткой b доходит до конца некоторой «листовой» дуги (дуги, входящей в лист). При изменении дерева нужно добавить к концу метки этой листовой дуги символ $S(i+1)$.

2. Ни один путь из конца строки b не начинается символом $S(i+1)$, но по крайней мере один начинающийся оттуда путь имеется. В этом случае должна быть создана новая листовая дуга, начинающаяся в конце b, помеченная символом $S(i+1)$. При этом, если b кончается внутри дуги, должна быть создана новая вершина. Листу в конце новой листовой дуги сопоставляется номер j. Таким образом, в правиле 2 возможно два случая.

3. Некоторый путь из конца строки b начинается символом $S(i+1)$. В этом случае строка $bS(i+1)$ уже имеется в текущем дереве, так что ничего не надо делать (в неявном суффиксном дереве конец суффикса не нужно помечать явно).[2].

2 Исходный код

Для решения задачи я создал классы TSuffixTree, TNode и TSuffixArray. Класс узла отвечает за работу с суффиксными ссылками и путевыми метками, а класс дерева – за корректное построение дерева путём использования active point.

suftree.cpp	
bool WalkDown(TNode *node)	Метод для изменения активного узла в случае переполнения путевой метки
void DFS(TNode *node, std::vector<int> &vector, int d)	Обход дерева в глубину для построения суффиксного массива
void TreeExtend(std::string::iterator newSymbol)	Расширение суффиксного дерева с помощью алгоритма Укконена
void AddSufLink(TNode *node)	Добавление суффиксной ссылки от последнего добавленного узла к новому
long int EdgeLength(TNode* node)	Метод, возвращающий длину путевой метки узла от его родителя
void Destroy(TNode *node)	Удаление дерева
sufarray.cpp	
int Compare(int index, const std::string &pattern)	Сравнение образца с содержимым массива
std::vector<int> Find(std::string &pattern)	Поиск образца в суффиксном массиве

Класс суффиксного дерева:

```
class TSuffixTree {
public:
    TSuffixTree(std::string input);      ~TSuffixTree();
    friend TSuffixArray;
private:
    std::string Text;
    long int Remainder, ActiveLength;
    TNode* LastNewNode, *ActiveNode;
    std::string::iterator ActiveEdge;
    long int EdgeLength(TNode* node);
    void AddSufLink(TNode* node);
    void TreeExtend(std::string::iterator add);
   void DFS(TNode *node, std::vector<int> &vector, int d)
};
```

Класс суффиксного массива:

```
class TSuffixArray {
public:
    std :: vector<int> Find(std::string& pattern);      TSuffixArray(TSuffixTree& Tree);
    ~TSuffixArray() = default;
private:
    std :: vector<int> Array;
    std :: string  Text;
    int Compare(int index, const std::string& pattern);
};
```

3 Консоль

```
 .../da_labs/lab_05  cat test1.test
abcdabc
abcd
bcd
bc

 .../da_labs/lab_05  cat test2.test
bfgctsg
gg
bfg
g
 .../da_labs/lab_05  ./a.out <test1.test
1: 1
2: 2
3: 2, 6

 .../da_labs/lab_05  ./a.out <test2.test
2: 1
3: 3, 7
```

4 Тест производительности

Для тестирования будем использовать генератор тестов, написанный на python. На вход программе будет подаваться образец длиной 1000 и 10 000 символов и тексты длиной 1 000 000 и 10 000 000 символов. Время работы программы будем замерять с помощью стандартной библиотеки *chrono*.

Тест 1: Длина образца 1000, длина текста - 1 000 000.
time: 3.00234 seconds

Тест 2: Длина образца 1000, длина текста - 10 000 000.
time: 30.37529 seconds

Тест 3: Длина образца 10 000, длина текста - 10 000 000.
time: 38.40231 seconds

Как видно из проведённых тестов, стадия препроцессинга в алгоритме Укконена почти не влияет на скорость поиска образца, чего нельзя сказать о длине текста, ведь при увеличении её в 10 раз, время поиска увеличилось примерно в 10 раз, что доказывает линейную сложность алгоритма.

5 Выводы

Выполнив четвёртую лабораторную работу по курсу «Дискретный анализ», я написал реализацию алгоритма Укконена построения суффиксного дерева за линейное время. Также в ходе выполнения работы я познакомился с алгоритмом поиска образца в суффиксном массиве.
Линейная сложность алгоритма подтвердилась при анализе производительности, что делает этот алгоритм очень эффективным из-за возможности построения дерева online.

Список литературы

[1] Дэн Гасфилд. *Строки, деревья и последовательности в алгоритмах* — «Невский Диалект», 2003. Перевод с английского: И. В. Романовский. — 654 с. (ISBN 5-94157-321-9 (рус.))

[2] *Suffix Tree*
URL: `https://en.wikipedia.org/wiki/Suffix_tree` (дата обращения: 10.04.2020)