Санкт-Петербургский Политехнический университет Петра Великого
Институт машиностроения, материалов и транспорта
Кафедра «Мехатроника и роботостроение» (при ЦНИИ РТК)

КУРСОВАЯ РАБОТА
Дисциплина: Объектно-ориентированное программирование
Тема: Алгоритм Укконена

Выполнила:
Студент группы 3331506/70401 М.Н.Тестерева

Преподаватель М.С.Ананьевский

«____»_____ 202__г.

Санкт-Петербург
2020 г.

Содержание

Введение

Курсовая работа представляет собой описание и реализацию алгоритма Укконена в соответствии с заданием преподавателя.

Алгоритм Укконена (англ. Ukkonen's algorithm) — алгоритм построения суффиксного дерева для заданной строки *s* за линейное время.

В данной работе реализован алгоритм только построения суффиксного дерева, рассмотренная теория для описания алгоритма не включает математические доказательства и описания принципов работы суффиксных деревьев, - того не требовалось в задании.

Формулировка задачи, которую решает алгоритм

Сам алгоритм решает задачу построения суффиксного дерева за линейное время.

Задачи, которые можно решить с помощью применения данного алгоритма можно разделить как:

1. Поиск подстрок:

Допустим, у нас есть шаблон, который мы хотим найти в тексте.

Построим суффиксное дерево для текста и будем читать шаблон вдоль дерева от корня. Если в какой-то момент не сможем прочитать следующую букву шаблона, значит шаблон ни разу не встречался в тексте.

Допустим, что он встречался, тогда, прочитав его, приходим в вершину *v* или останавливаемся на ребре.

В случае остановки на ребре пройдем дальше от корня до первой вершины *v*. Далее прочитаем числа, записанные в листьях-потомках вершины *v*. Эти числа — номера суффиксов, начинающихся с шаблона, т.е. индексы вхождений шаблона в текст.

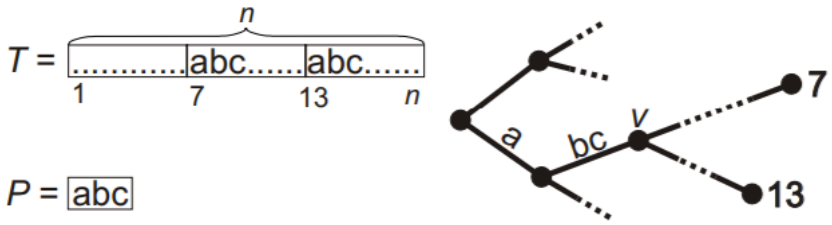

Рисунок 1 – Пример.

2. Поиск наибольшей общей подстроки:

Построим суффиксное дерево для объединенных текстов. Суффиксы из одного текста назовём «короткие», из другого – «длинные», и для каждой внутренней вершины выясним, есть ли у неё потомки, соответствующие и тем, и другим. Это можно сделать обходом в глубину.

Каждому узлу сохраним тип при этом тип суффиксов. Если для вершины хранятся оба типа – значит, это общий суффикс. Найдём самую удалённую от корня такую вершину, и её глубина и будет решением задачи.

Словесное описание алгоритма

Как уже говорилось, *алгоритм Укконена* – это алгоритм построения суффиксного дерева строки за линейное время, использующий on-line подход, т.е. после *k*-ого шага алгоритма получим суффиксное дерево для префикса строки длины *k*.

1. О суффиксных деревьях.

Суффиксное дерево (СД) — это способ представления текста, точнее говоря бор, состоящий из всех суффиксов данной строки, а бор в свою очередь – корневое дерево, все ребра которого помечены разными символами. В нашем случае – ребра дерева будут помечены всевозможными суффиксами строки.

Неформально говоря, чтобы построить СД для текста, нужно приписать специальный символ $ в конец текста, взять все получившиеся суффиксы, подвесить их за начала и склеить все ветки, идущие по одинаковым буквам.

Очевидно, что листьев всегда будет (n + 1) для строки $t_1...t_n$, то есть столько же, сколько суффиксов, а общее число вершин в суффиксном дереве квадратично.

Чтобы хранить суффиксное дерево, используя линейную память. оставим в СД только вершины разветвления, то есть имеющие не менее двух детей. Вместо строки для ребра будем хранить ссылку на сегмент текста. В таком виде суффиксное дерево называется *сжатым*.

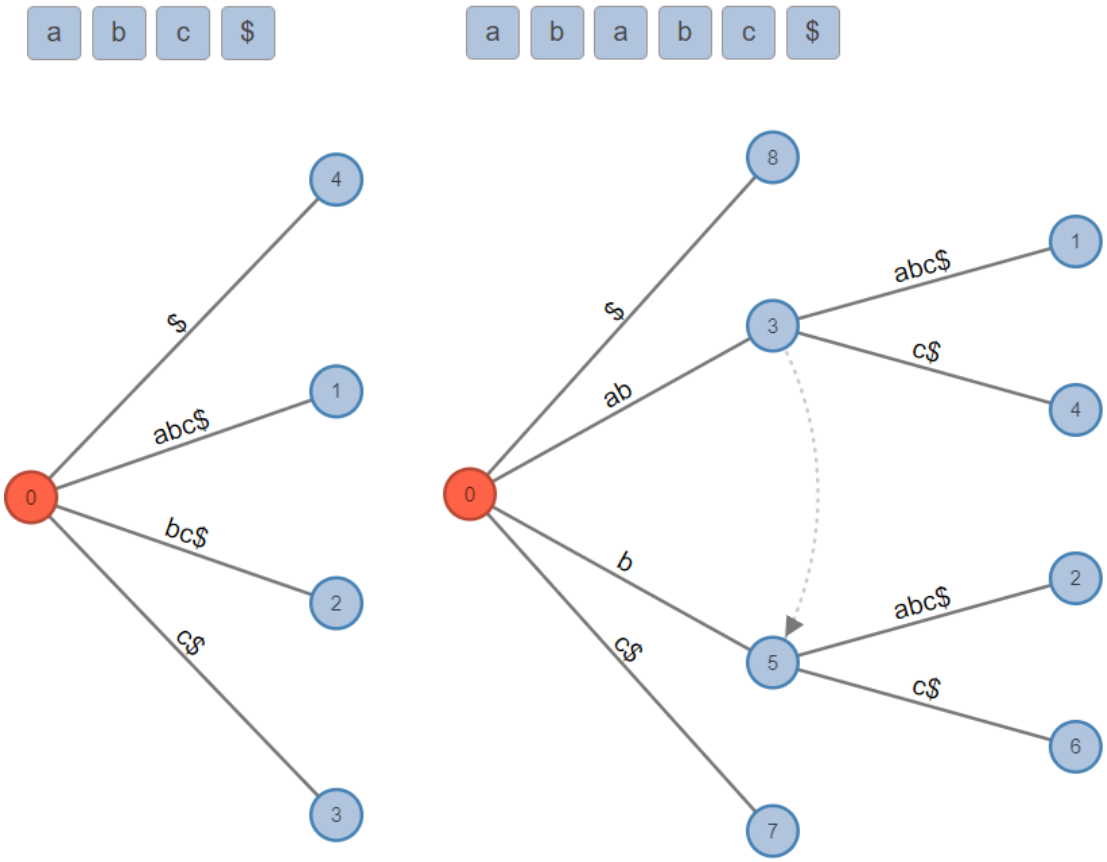

Рисунок 2 – Примеры суффиксных деревьев.

2. Наивный алгоритм *O(n^3)*.

Рассмотрим сначала наивный метод, который строит дерево за время *O(n^3)*, где *n* — длина исходной строки *s*. В дальнейшем данный алгоритм будет оптимизирован таким образом, что будет достигнута линейная скорость работы.

Алгоритм последовательно строит неявные суффиксные деревья (в таких СД нет знака «$») для всех префиксов исходного текста.

На *i*-ой фазе неявное суффиксное дерево τ_{i-1} для префикса s[1…i−1] достраивается до τ_i для префикса s[1…i]. Достраивание происходит следующим образом: для каждого суффикса подстроки s[1…i−1] необходимо спуститься от корня дерева до конца этого суффикса и дописать символ s_i.

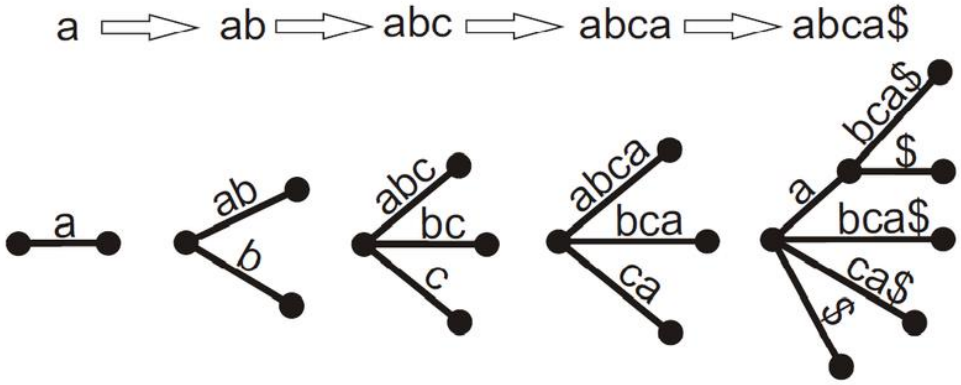

Рисунок 3 - Построение суффиксного дерева.

Алгоритм состоит из *n* фаз. На каждой фазе происходит продление всех суффиксов текущего префикса строки, что требует $O(n^2)$ времени. Следовательно, общая асимптотика алгоритма составляет $O(n^3)$.

3. Продление суффиксов (правила).

Можно увидеть, что существует несколько решений при продлении суффиксов символом *s*, конкретно говоря их три:

1) *Продление листа:* суффикс закончился в листе → добавляем символ s в конец подстроки, которой помечено ребро, ведущее в этот лист.

2) *Разветвление:*

а) суффикс закончился на вершине, не являющейся листом, из которой нет пути по символу → создадим новый лист, в который из текущей вершины ведёт дуга с пометкой *s*;

б) суффикс закончился на ребре → разобьем текущее ребро новой вершиной и подвесим к ней еще одного ребенка с дугой, помеченной *s*.

3) *Ничего не делать:* суффикс закончился в вершине, из которой есть путь *s*.

Назовём это *правилами*.

Также есть наблюдения:

1) Как только применилось правило 3, оно применяется до конца фазы.

2) После того, как мы применили правило ответвления и создали новый лист, в следующих фазах к этому листу всегда будет применяться правило удлинения.

(«Стал листом – листом и останешься»).

4. Линейный алгоритм.

Так как алгоритм Укконена предполагает работу за линейное время, внесём в уже описанный выше наивный алгоритм некоторые изменения:

1) Добавим суффиксные ссылки.

Переход по суффиксной ссылке будет вести в вершину, соответствующую той же строке, но без первого символа. Она определена для внутренних вершин дерева. Для корня суффиксная ссылка не определена.

2) *Ускоренное прохождение рёбер*.

3) *Пропуск части продолжений, где не нужно ничего вычислять*.

4) Более того, уже было рассмотрено *использование линейной памяти* с помощью использования сжатых суффиксных деревьев.

Так как теперь каждая из внутренних вершин является вершиной разветвления, то она добавляет к своему поддереву как минимум один лист. Листьев же в СД всего (n + 1) для строки $t_1...t_n$, поэтому внутренних вершин может быть в диапазоне 1...n.

По отдельности оптимизации выглядят как полезная эвристика для ускорения, но вместе дают «прорыв» во времени выполнения алгоритма.

Временная оценка рассматривается в пунктах далее.

Реализация алгоритма Укконена

Программная реализация на C++ в приложении 1 к курсовой работе. Также алгоритм приложен как main.cpp на сайте github.

В данной программе реализовано построение суффиксного дерева методом Укконена.

В программе существует структура, описывающая узел (под узлом имеется в виду и развилка, и лист) а также входящее в него ребро.

Кроме того, есть основные функции, главным образом реализация представлена в функции *void suffix_tree_extend(char symbol)*, которая добавляет символ в суффиксное дерево.

По поводу всех функций и неочевидных переменных представлены комментарии в коде.

Анализ полученного программного решения алгоритма

1. Анализ сложности алгоритма.

Длина входной строки – *n*.

В течение работы алгоритма создается не более *O(n)* вершин по лемме о размере суффиксного дерева для строки.

Все суффиксы, которые заканчиваются в листах на каждой итерации мы увеличиваем на текущий символ по умолчанию за *O(1)*.

Текущая фаза алгоритма будет продолжаться, пока не будет использовано правило продления 3. Сначала неявно продлятся все листовые суффиксы, а потом по правилам 2.а) и 2.б) будет создано несколько новых внутренних вершин. Так как вершин не может быть создано больше, чем их есть, то в среднем на каждой фазе будет создано *O(1)* вершин.

Так как мы на каждой фазе начинаем добавление суффикса не с корня, а с индекса j∗, на котором в прошлой фазе было применено правило 3, то нетрудно показать, что суммарное число переходов по рёбрам за все *n* фаз равно *O(n)*.

Таким образом, при использовании всех приведённых эвристик алгоритм Укконена работает за ***O(n)***.

2. Время работы при разных входных данных.

Для наглядного сравнения сопоставим алгоритм Укконена за $O(n)$ с наивным алгоритмом с суффиксными ссылками за $O(n^2)$.

Составим два графика – для малой обычной случайной входной строки и для больших строк в худшем случае, и посмотрим, за какое время каждый алгоритм построит суффиксное дерево.

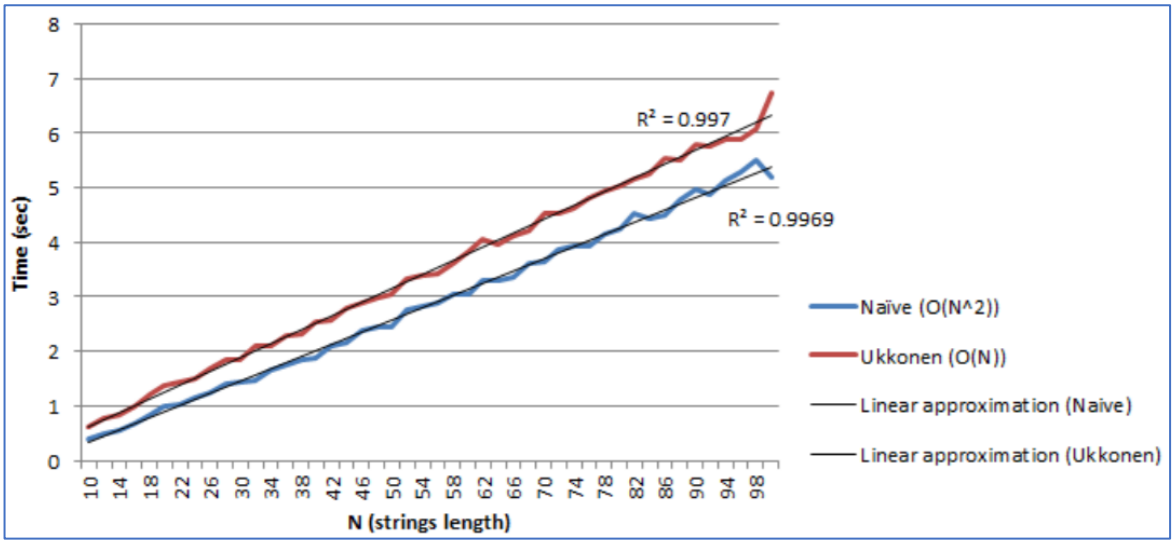

Рисунок 4 -График зависимости при рандомных строках до стони символов.

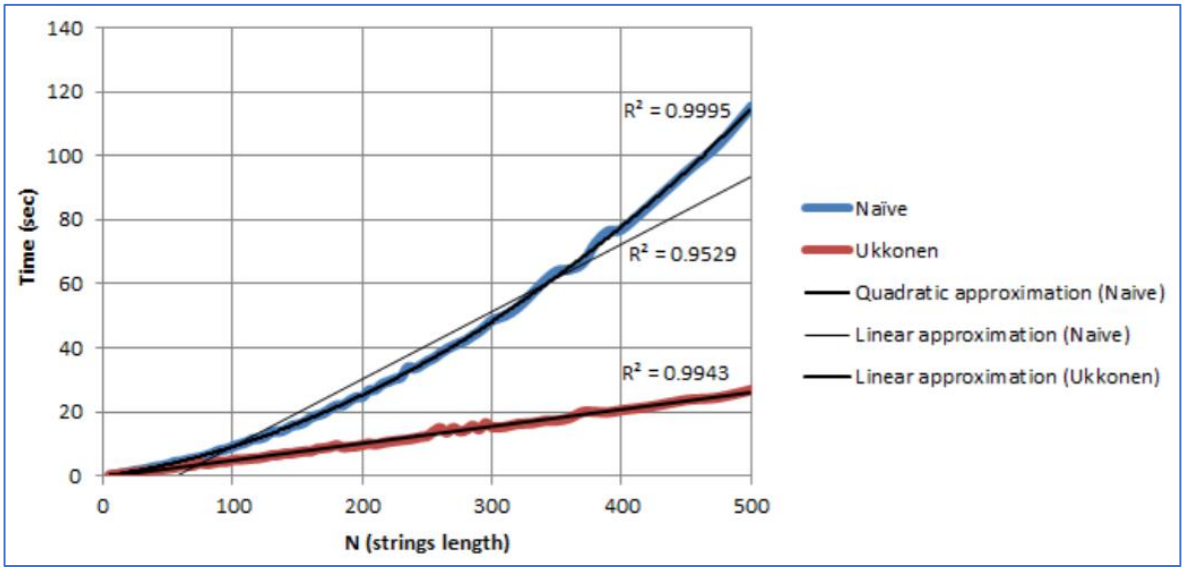

Рисунок 5 – График зависимости при худших входных строках длинной до 500.

Как видно, для малых случайных входных данных разница пренебрежимо мала, но при худшем варианте входных данных алгоритм Укконена сохраняет линейный характер работы и заметно выигрывает у наивной реализации алгоритма.

Область применения

Практично использовать алгоритм для нахождения различных подстрок в строках и описанных задач из п.2, но на самом деле неизвестно, кто же его всё-таки применяет на практике. Дело в том, что область применения алгоритма ограничена его минусами:

1) Размер суффиксного дерева сильно превосходит входные данные, поэтому при очень больших входных данных алгоритм Укконена сталкивается с проблемой memory bottleneck problem.

2) Для несложных задач, таких как поиск подстроки, проще и эффективней использовать другие алгоритмы (например поиск подстроки с помощью префикс-функции).

3) При внимательном просмотре видно, что на самом деле алгоритм работает за время $O(n \cdot |\Sigma|)$, используя столько же памяти, так как для ответа на запрос о существовании перехода по текущему символу за $O(1)$ необходимо хранить линейное количество информации от размера алфавита в каждой вершине. Поэтому, если алфавит очень большой требуется чрезмерный объём памяти. Можно сэкономить на памяти, храня в каждой вершине только те символы, по которым из неё есть переходы, но тогда поиск перехода будет занимать $O(\log|\Sigma|)$ времени.

4) Константное время на одну итерацию — это амортизированная оценка, в худшем случае одна фаза может выполняться за O(n) времени.

5) На сегодняшний день существуют кэш-эффективные алгоритмы, превосходящие алгоритм Укконена на современных процессорах.

6) Также алгоритм предполагает, что дерево полностью должно быть загружено в оперативную память. Если же требуется работать с большими размерами данных, то становится не так тривиально модифицировать алгоритм, чтобы он не хранил всё дерево в ней.

В общем, особой популярностью не пользуется.

Заключение

Выполнена курсовая работа по теме «Алгоритм Укконена», где:

- представлено словесное описание алгоритма и его теоретические основы;
- перечислены задачи, которые можно решить с использованием данного алгоритма, и причины, почему на практике его применение ограничено.
- представлена реализация алгоритма на C++;
- проведен анализ алгоритма, в том числе его анализ временной сложности и оценка времени работы в зависимости от различных входных данных;

Список литературы

1. Вики-конспекты от ИТМО
https://neerc.ifmo.ru/wiki/index.php?title=%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC_%D0%A3%D0%BA%D0%BA%D0%BE%D0%BD%D0%B5%D0%BD%D0%B0

2. Юрий Лифшиц — Построение суффиксного дерева за линейное время.
http://yury.name/internet/01ianote.pdf

3. Habrahabr — Построение суффиксного дерева: алгоритм Укконена
https://habr.com/ru/post/111675/

Приложение 1. Код программы

```cpp
#include <iostream>
#include <string>

const int INF = 1e9;
const int ALP_SIZE = 256;
const int MAXN = 5000;

using namespace std;

int root;
int count_last_added_node;
int position;
int node_suffix_link;
int remainderr;                                     /// число явно добавляемых суффиксов.
/// Составляющие активной точки - узел, ребро, длина. По этим координатам добавляется новый суффикс.
int active_node, active_edge, active_length;

/// Структура, описывающая узел суффиксного дерева
struct Node {

    int start;                                      /// метка ребра - позиция самого правого элемента
    int end;                                        /// метка ребра - позиция самого левого элемента (для листа: inf)
    int suffix_link;                                /// суффиксная ссылка
    int next_nodes[ALP_SIZE];                        /// массив прямых ссылок на следующие узлы/узел:
                                                    /// для листа - заполнен нулями,
                                                    /// для разветвления - код след символа -> номер следующего узла.
    int edge_length() {
        return (min(end, position + 1) - start);    /// расчёт длины ребра
    }
};
```

```cpp
/// Инициализируем массив узлов - дерево
Node tree[2*MAXN];
/// Массив символов, добавленных в дерево - текст/строка
char text[MAXN];

/// Функция создания нового узла
int new_node(int start, int end = INF) {
    Node node{};
    node.start = start;
    node.end = end;
    node.suffix_link = 0;
    for (int & i : node.next_nodes)
    {i = 0;}
    /// Записываем только что созданный узел в дерево
    tree[++count_last_added_node] = node;
    return count_last_added_node;
}

/// Первый символ ребра от активной точки
char symbol_active_edge() {
    return text[active_edge];
}
```

```
/// Добавление суффиксной ссылки в узел
void add_suffix_link(int node) {
    if (node_suffix_link > 0) tree[node_suffix_link].suffix_link = node;
    node_suffix_link = node;
}
```

```
/// Нужно ли спуститься на следующий узел дальше?
bool walk_down(int node) {
    /// В случае, если активная длина больше длины ребра активного узла, переход будет совершен
    if (active_length >= tree[node].edge_length()) {
        active_edge += tree[node].edge_length();
        active_length -= tree[node].edge_length();
        active_node = node;
        return true;
    }
    return false;
}

/// Функция для инициализации каждого нового суффиксного дерева
void suffix_tree_initialization() {
    node_suffix_link = 0; count_last_added_node = -1; position = -1;
    remainderr = 0; active_node = 0; active_edge = 0; active_length = 0;
    root = active_node = new_node( start: -1,  end: -1);
}
```

```
/// Функция для пользователя
void create_suffix_tree() {
    string my_string;
    cout << "Введите произвольную строку:  ";
    cin >> my_string;
    suffix_tree_initialization();
    for(char i : my_string)
        suffix_tree_extend(i);
    suffix_tree_extend( symbol: '$');
}

int main() {
    create_suffix_tree();
    create_suffix_tree();
    return 0;
}
```

```
/// Функция, добавляющая символ (букву) в суффиксное дерево
void suffix_tree_extend(char symbol) {
    text[++position] = symbol;
    node_suffix_link = 0;
    remainderr++;

    while(remainderr > 0) {
        /// Если активная точка в узле, то активное ребро определяется добавляемым символом по его позиции
        if (active_length == 0) active_edge = position;
        /// Если от активной точки двигаться дальше по нужному символу невозможно...
        if (tree[active_node].next_nodes[symbol_active_edge()] == 0) {
            /// ...создаём новый лист
            int leaf = new_node(position);
            /// Присваиваем прямую ссылку узлу на лист
            tree[active_node].next_nodes[symbol_active_edge()] = leaf;
            /// Добавляем к активному узлу, который стал развилкой, суффиксную ссылку
            add_suffix_link(active_node);

        ///remainder == 0
        } else {
            int next_node = tree[active_node].next_nodes[symbol_active_edge()];
            if (walk_down(next_node)) continue;
            /// Если суффикс, который мы хотим добавить, уже существует в дереве
            if (text[tree[next_node].start + active_length] == symbol) {
                active_length++;                /// изменяем параметры активной точки
                add_suffix_link(active_node);   /// добавляем суффиксную ссылку в активный узел, если это необходимо
                break;
            }
            /// Создание разветвления
            int split = new_node(tree[next_node].start, end: tree[next_node].start + active_length);
            tree[active_node].next_nodes[symbol_active_edge()] = split;
            int leaf = new_node(position);
            tree[split].next_nodes[symbol] = leaf;
            tree[next_node].start += active_length;
            tree[split].next_nodes[text[tree[next_node].start]] = next_node;
            add_suffix_link(split);
        }
        remainderr--;

        /// Если активный узел - корень, но активная точка не в узле - изменяются параметры акт точки
        if (active_node == root && active_length > 0) {
            active_length--;
            active_edge = position - remainderr + 1;
        /// Если из акт узла (не корня при этом) произошла вставка
        /// (возможно лишь когда активная длина = 0, т.е. акт точка в узле)
        /// - сменить активный узел.
        } else
            active_node = tree[active_node].suffix_link > 0 ? tree[active_node].suffix_link : root;
    }
}
```