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

Разработал: ст. гр. 3331506/80401 Серов Д. К.

Преподаватель Кузнецова Е. М.

Санкт-Петербург
2021 г

Содержание

Введение

Алгоритм Укконена – это один из наиболее простых и популярных алгоритм построения суффиксного дерева для строки длиной n за время $O(n)$.

Суффиксные деревья, в свою очередь, могут быть использованы для поиска различных подстрок в строке или поиска множества заранее неизвестных слов в тексте за время $O(w)$, где $w -$ длина слова. Помимо этого, суффиксные деревья могут быть преобразованы в другую структуру данных для хранения информации о подстроках строки – суффиксный автомат.

Описание алгоритма

Алгоритм Укконена строит суффиксное дерево, добавляя в него по одной букве. Алгоритм работает по шагам, слева направо, один шаг на каждый символ строки. При этом каждый шаг может включать более одной операции, но количество операций все равно получается $O(n)$.

Подстроки, на которые разбивается входная строка, хранятся во всех узлах кроме корня как индексы начала и конца элемента во входной строке (на последующих рисунках для более понятной визуализации алгоритма они будут представлены в виде самих подстрок в качестве ребер между узлами).

На каждом шаге в алгоритме вставляются все суффиксы до текущей позиции и в простых случаях, когда нет повторяющихся элементов в текущем узле, это должно делаться за счет вставки нового узла к текущему и продления листовых узлов.

Однако, существует правило, согласно которому, конец подстроки в листовых узлах хранится как индекс последнего элемента входной строки (конца строки). Это ключевое правило «листом был – листом и останешься» позволяет не добавлять в листовые узлы новые элементы при каждом шаге, серьезно уменьшая время работы алгоритма (данное правило игнорируется в визуализации для упрощения восприятия работы алгоритма, но является существенным при его реализации).

Пример последовательного построения суффиксного дерева в простейшем случае для строки “*123”* представлен на рисунках 1–3.

Для визуализации алгоритма и приведения примеров был использован следующий сайт: http://brenden.github.io/ukkonen-animation/

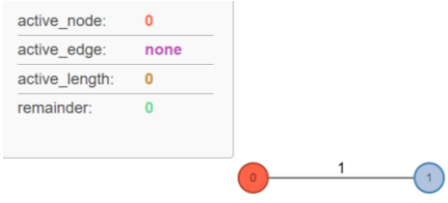

Рисунок 1 — Добавление символа “1”

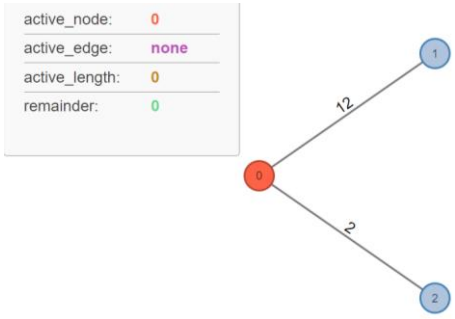

Рисунок 2 — Добавление символа “2”

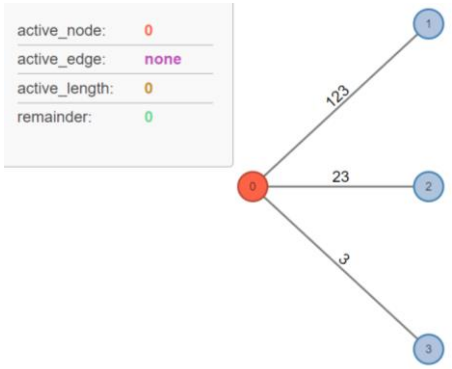

Рисунок 3 — Добавление символа “3”

Для добавления уже повторяющихся элементов алгоритм становится сложнее и предполагает использование следующих понятий:

1) Активная длина – место в подстроке, на котором мы находимся при попытке добавить новый символ.
2) Текущий узел – узел, в подстроке ребенка которого, осуществляется попытка добавить символ
3) Активное ребро – узел, по подстроке которого осуществляется проход в данный момент.
4) Остаток – сколько значений в подстроке было пройдено, ничего не добавив

Таким образом, если осуществляется попытка добавить уже повторяющийся элемент в дерево, активная длина и остаток увеличиваются на один, активным ребром становится узел, подстрока которого начинается на тот же элемент, что пытаемся добавить, а текущим узлом – родитель этого активного ребра.

Далее осуществляется проход по активному ребру до тех пор, пока не будет найдено несовпадение символа в подстроке активного ребра и символа, который осуществляется попытка добавить или пока ребро не кончится. Рассмотрим эти 2 случая:

1) Если было найдено несовпадение символа, то активное ребро расщепляется, конец его подстроки становится тем элементом, после которого было найдено несовпадение символа, а его детьми становятся два узла, в один из которых идут все оставшиеся символы только что расщепленной подстроки, а в другой – символ, попытка добавить который осуществлялась.

При этом к расщепленному узлу идет суффиксная ссылка от предыдущего расщепленного узла (до следующего увеличения остатка), остаток уменьшается на один. Далее опять возможны два случая:

а) Если суффиксная ссылка из расщепленного узла уже была ранее сделана, то текущим узлом становится узел, на который указывает данная ссылка, активная длина сохраняется, среди детей текущего узла находится активное ребро и происходит дальнейшее добавление того же символа, при попытке добавить который, расщепился предыдущий узел.

б) Если суффиксной ссылки у расщепленного узла нет, то активная длина становится равной остатку, текущим узлом становится корень и из него происходит дальнейшее добавление того же символа.

Частичное отражение данного случая может быть получено при построении суффиксного дерева для строки “123124”. Построение дерева для первых трех символов идентично продемонстрированным ранее на рисунках 1-3, дальнейшее построение представлено на рисунках 4-8.

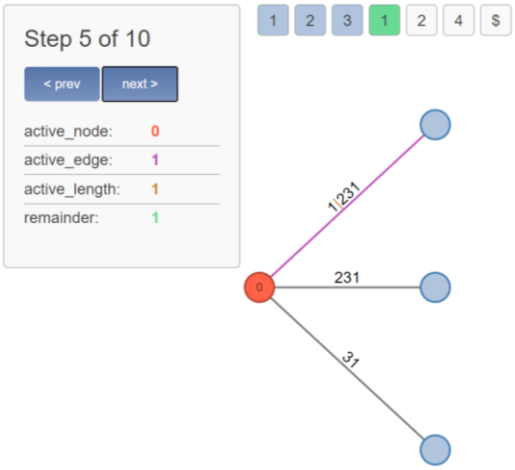

Рисунок 4 — Попытка добавление символа “1”

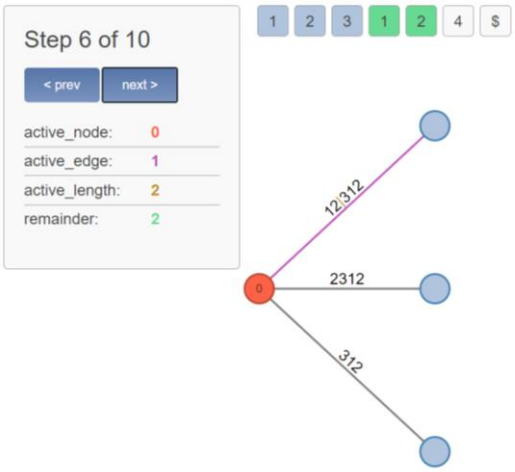

Рисунок 5 — Попытка добавление символа “2”

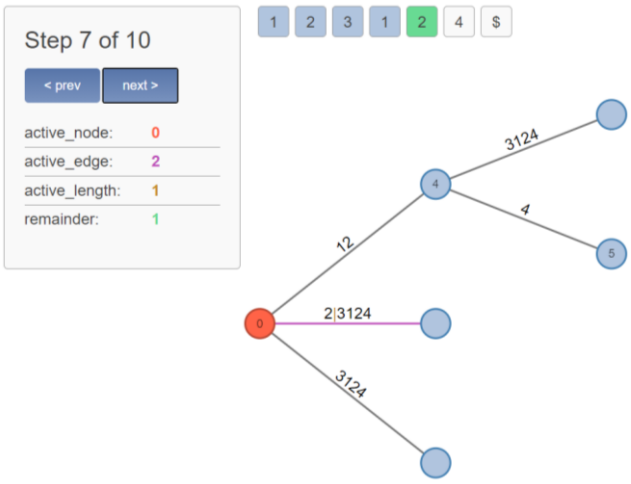

Рисунок 6 — Расщепление узла 4, смена активного и текущего узлов

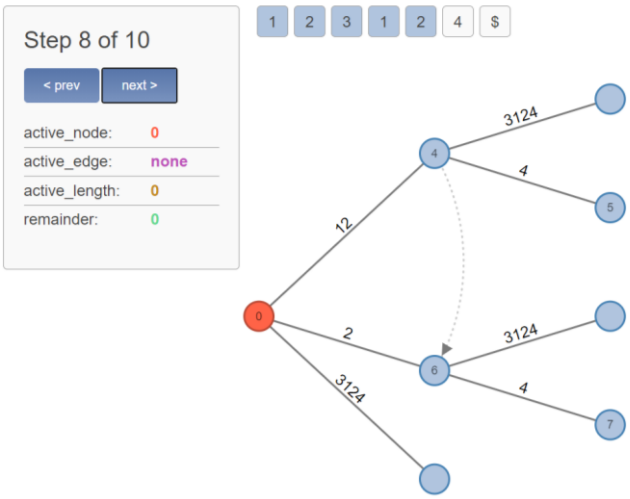

Рисунок 7 — Расщепление узла 6, смена активного и текущего узлов

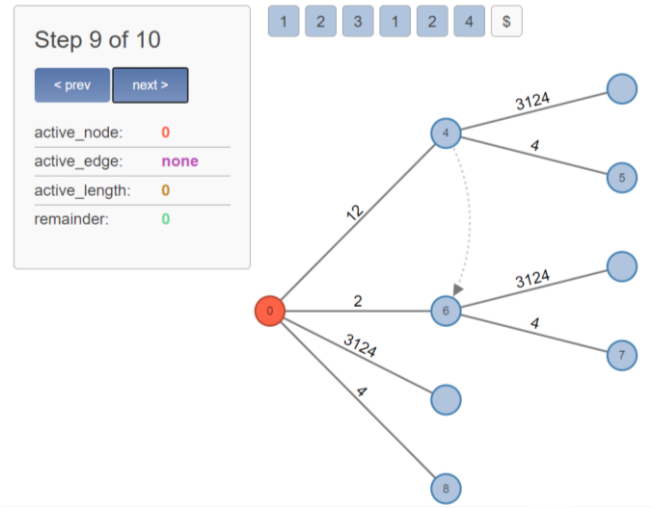

Рисунок 8 — Добавление символа “4”

2) Если активное ребро кончилось, тогда активным ребром становится его потомок, начинающийся на следующий символ, который осуществляется попытка добавить, (если таковой имеется, иначе просто добавляется новое ребро как описано выше), текущим узлом становится бывшее активное ребро, а активная длина становится равно единице.

Пример подобной ситуации может быть получен при попытке построения дерева для строки “1231241245” на одном из шагов, как показано на рисунках 9-10.

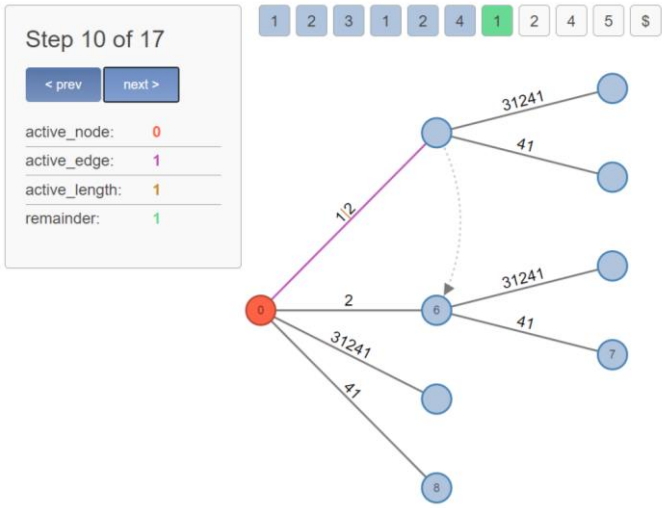

Рисунок 9 — Попытка добавить “2”

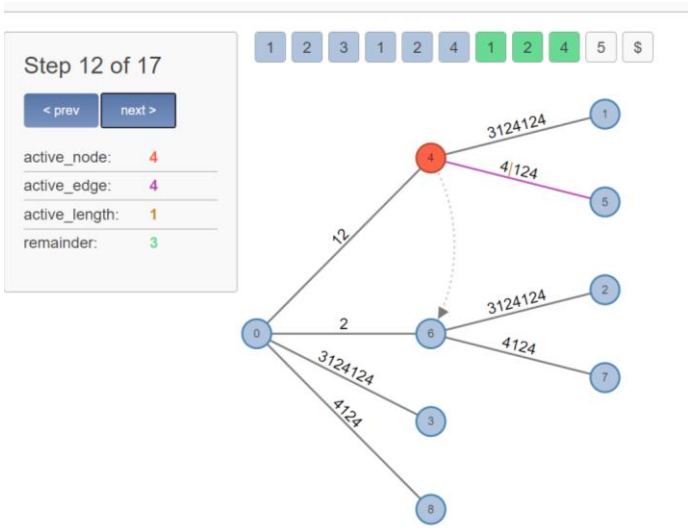

Рисунок 10 — Попытка добавить “4”

Также стоит отметить, что в некоторых реализациях алгоритма для упрощения организации окончания алгоритма в конце строки добавляют некий дополнительный символ, не встречавшийся в алфавите входной строки.

Сложность алгоритма

Без применения суффиксных ссылок и правила «листом был – листом и останешься» асимптотика алгоритма составляла бы $O(n^3)$.

Однако суффиксные ссылки и активная длина позволяют нам не проходить все дерево заново при каждом шаге, уменьшая асимптотику до $O(n^2)$.

Правило «листом был – листом и останешься» также позволяют при прохождении нового символа в исходной строке, не добавлять этот символ в листовые узлы, а производить изменения в дереве только в одном месте на каждый символ, тем самым уменьшая асимптотику до $O(n)$.

Исследование алгоритма

Для тестирования времени работы алгоритма использовались функции из библиотеки <chrono>. Было проведено несколько измерений для каждого числа входных символов и вычислено среднее арифметическое полученных результатов. В качестве входных данных был использован отрывок из статьи автора алгоритма Ukkonen, E. (1995). "On-line construction of suffix trees". Полученные результаты представлены в таблицах 1 и 2.

Таблица 1- Время выполнения алгоритма при небольшой длине входных данных.

Число элементов, шт.	10	25	50	100	200
Время выполнения, нс	27680	71770	127800	257700	524500

Таблица 2- Время выполнения алгоритма при увеличенной длине входных данных.

Число элементов, шт.	400	25	50	100
Время выполнения, нс	1103900	1730200	2588900	3142400

По результатам измерений, приведенных в таблицах 1 и 2, был построен график, как показано на рисунке 11.

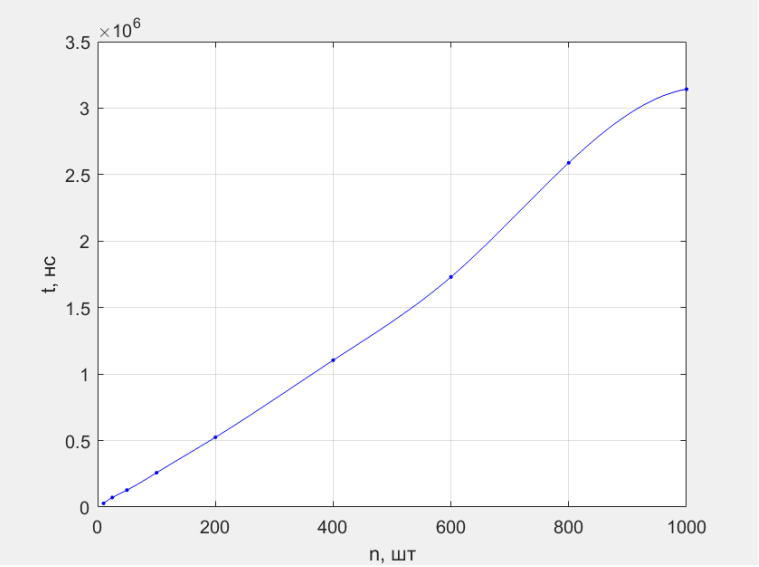

Рисунок 11 – График времени выполнения алгоритма

По рисунку 11 видно, что время выполнения алгоритма действительно линейное, как и утверждалось ранее. Таким образом, сложность алгоритма действительно $O(n)$.

Список литературы

1. Habrahabr – Алгоритм Укконена: от простого к сложному https://habr.com/ru/post/533774/

2. Ukkonen, E. "On-line construction of suffix trees", Algorithmica, 14:3 (1995), 249-260.

3. Дэн Гасфилд — Строки, деревья и последовательности в алгоритмах: Информатика и вычислительная биология — СПб.: Невский Диалект; БХВ-Петербург, 2003. — 654.

4. A. Apostolico, M. Crochemore, M. Farach-Colton, Z. Galil, S. Muthukrishnan, “40 years of suffix trees”, Communications of the ACM, 59:4 (2016), 6

Приложение 1. Код программы

```
#include <iostream>
#include <string>
#include <cstring>
#include <chrono>



// Ukkonen's algorithm
static volatile int arrLength = 0;

char input[10000] = {"print your text here"};

class Node {
public:
    int start;
    int end;
    Node *children[255] = {nullptr};
    Node *suffixLink;
public:
    Node();
    Node(int start, int end);
    ~Node();
};
class Tree {
private:
    Node *root;
    Node *current;
    Node *lastSplit;
    int counter;
    int remainder;
    int activeLength;
    bool isCntInc;
private:
    void split(Node* temp);
    void add(Node *temp);
public:
    void makeTree ();
public:
    Tree();
    ~Tree();
};

Node::Node() {
    start = 0;
    end = arrLength+1;
    suffixLink = nullptr;
}

Node::Node(int start, int end) {
    this->start = start;
    this->end = end;
    suffixLink = nullptr;
}


Node::~Node() {}
```

```
59  Tree::Tree() {
60      root = nullptr;
61      current = nullptr;
62      lastSplit = nullptr;
63      remainder = 0;
64      activeLength = 0;
65      counter = 0;
66      isCntInc = false;
67  }
68
69  Tree::~Tree() {
70      delete root;
71      delete current;
72  }
73
74  void Tree:: split(Node* temp) {
75      int start = temp->start;
76      temp->start = start + activeLength;
77      Node *parent = new Node(start, end: start+activeLength-1);
78      current->children[input[start]] = parent;
79      parent->children[input[temp->start]] = temp;
80      parent->children[input[counter]] = new Node (counter,  end: arrLength+1);
81      remainder --;
82
83      if (lastSplit != nullptr)
84          lastSplit->suffixLink = parent;
85      lastSplit = parent;
86
87      if (current == root){
88          activeLength = remainder;
89          return;
90      }
91
92      if (current->suffixLink == nullptr) {
93          current = root;
94          activeLength = remainder;
95          return;
96      }
97
98      current = current->suffixLink;
99      return;
100 }
101
102 void Tree::add(Node* temp) {
103     if ((counter == arrLength+2) && (remainder == 0))
104         return;
105     if (isCntInc == false) {                                    ⚠ 4
106         if (activeLength == 0){
107             if (current->children[input[counter]] == nullptr) {
108                 current->children[input[counter]] = new Node(counter,  end: arrLength+1);
109                 if (remainder == 0) {
110                     counter++;
111                     return add(current);
112                 }
113                 remainder--;
114                 if (temp->suffixLink == nullptr){
115                     current = root;
116                     activeLength = remainder;
117                 }
118                 else
119                     current = temp->suffixLink;
120                 return add(current);
121             }
```

```
122             else {
123                 isCntInc = true;
124                 remainder ++;
125                 activeLength ++;
126                 counter ++;
127                 lastSplit = nullptr;
128                 return add( temp: current->children[input[counter-1]]);
129             }
130         }
131 
132         if (activeLength != 0) {
133             Node *child = current->children[input[counter-activeLength]];
134             int length = child->end - child->start;
135             if ((length+1) <= activeLength) {
136                 current = child;
137                 activeLength = activeLength - (length+1);
138                 return add(current);
139             }
140             if (input[child->start+activeLength]!=input[counter]){
141                 split(child);
142                 return add(current);
143             }
144             else {
145                 isCntInc = true;
146                 remainder ++;
147                 activeLength ++;
148                 counter ++;
149                 lastSplit = nullptr;
150                 return add(child);
151             }
152         }
153     }
154 
155     if (isCntInc == true) {
156         int index = temp->start+activeLength;
157         int length = temp->end - temp->start;
158         if ((length+1)==activeLength){
159             current = temp;
160             activeLength = 0;
161             if (current->children[input[counter]] == nullptr){
162                 isCntInc = false;
163                 return add(current);
164             }
```

```
165             lastSplit = nullptr;
166             counter ++;
167             remainder ++;
168             activeLength++;
169             return add( temp: current->children[input[counter-1]]);
170         }
171
172         if (input[index] == input[counter]){
173             lastSplit = nullptr;
174             counter ++;
175             remainder ++;
176             activeLength ++;
177             return add(temp);
178         }
179
180         isCntInc = false;
181         split(temp);
182         return add(current);
183     }
184 }
```

```
185                                                                    ⚠ 4
186
187
188 void Tree::makeTree() {
189     arrLength = strlen(input)-1;
190     Node *head = new Node();
191     root = head;
192     current = head;
193     input[arrLength+1] = '$';
194     add(current);
195 }
196
197
198
199 int main() {
200     auto begin = std::chrono::steady_clock::now();
201     Tree testTree;
202     testTree.makeTree();
203     auto end = std::chrono::steady_clock::now();
204     auto elapsed_ms = std::chrono::duration_cast<std::chrono::nanoseconds>(end - begin);
205     std::cout << "The time: " << elapsed_ms.count() << " ns\n";
206 }
207
```