Suffix structure lecture
Moscow International Workshop ACM ICPC 2015

19 November, 2015

1 Introduction

Consider problem of pattern matching. One of possible statement is as follows: you are given text T and n patterns S_i. For each of patterns you are to say whether it has occurrence in text. There are two ways in solving this problem. The first one (considered easier) is Aho-Corasick algorithm which constructs automaton that recognizes strings that contain some of patterns as substring. The second one (considered harder) is usage of suffix structures which are mainly suffix array, suffix automaton or suffix tree. This lecture will be dedicated to the last two, their relations and applications.

2 Naive solution

Consider arbitrary substring $s[pos, pos + len - 1]$. It is the prefix of length len of suffix of string which starts in position pos. Given this, we can use following naive solution for the problem: let's add all the suffixes of the string in trie. Then for each prefix of every string in trie will correspond exactly one vertex hence we can check whether S_i is substring of T in $O(S_i)$.

Disadvantages are obvious - such solution needs $O(|T|^2)$ time and memory. There are two ways of solving this problem which lead to suffix tree and suffix automaton.

3 Suffix tree

We can see that every top-down way in trie is substring of T. Hence we can remove from trie all nodes which are, neither root nor vertex corresponding to some suffix and its degree equals 2 (i.e., nodes which are not crossroads - they have exactly one ingoing and exactly one outgoing edge). Instead of path formed by such vertices from one crossroad to another one we can simply write on edge indices of substring corresponding to such path. Such compressed structure is called suffix tree. You can find its example below.

Let's show that suffix tree needs $O(|T|)$ memory. Let's consequentially add new suffixes keeping trie compressed. Then on each step we will add either one or two new vertices. Indeed after we added first new vertex we have splitted some edge and simply hanged new leaf to it.

There are some algorithms of fast suffix tree construction. Maybe the most known among them is Ukkonen algorithm but unfortunately it will not be considered in this lecture.

4 Suffix automaton

Talking formally Deterministic Finite Automaton (DFA) is defined as a fiver $A = (Q, \Sigma, \delta, q_0, F)$. Where Q set of states, Σ alphabet, δ set of transitions, q_0 initial state, F set of final states.

But we will consider automaton as directed graph in which every edge has some letter written on it (edges in automaton are called transitions and vertices are called states). Also there can't be transition from the same state by the same character written on edge (that's what deterministic states for).

We will say that q accepts string s if there is such path from q_0 to q, such that if we will consequentially write all characters from this path we will have string s. Because of determinism of automaton it is bijection - every path corresponds to some string and vice versa every string corresponds to some path. Automaton accepts string s if it is accepted by one of its final states.

So, suffix automaton of string s is such *minimal* automaton that accepts all suffixes of s and only them. Minimal states for minimum amount of states. We will note that it will be not only directed but also acyclic because set of accepted words is finite and if we had a cycle we would have an opportunity to make accepted word as huge as we want.

Here is example of suffix automaton for string *abbcbc* and example of suffix tree for string *cbcbba* (In the first picture the double circle marked the final state):

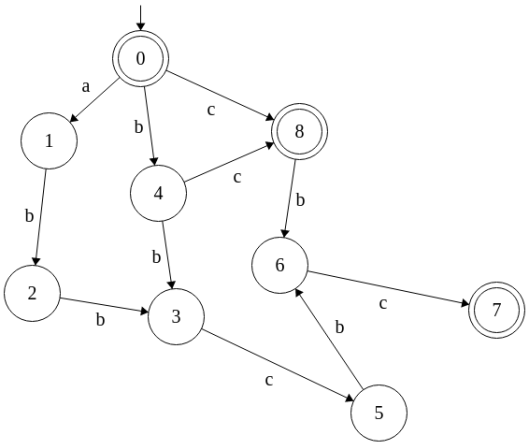

Figure 1: Automaton for *abbcbc*

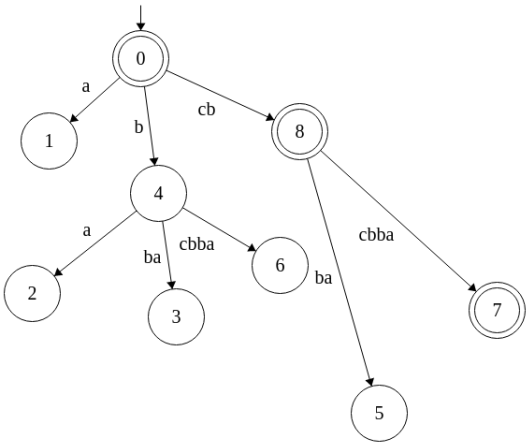

Figure 2: Suffix tree for *cbcbba*

As you might guess, we placed these two structures together not accidentally. After looking at the automaton to the left, you can easily understand that a set of strings, which accepted by state k coincides with the set of reversed strings which end on the edge that leads to the vertex k in suffix tree. This fact is not a coincidence, and will be discussed below.

To have an opportunity to construct and use this structure we have to know how it works.

First trivial fact that we can find out is that for every pair of strings a, b which are accepted by the same state q of arbitrary automaton and for any string x strings ax and bx are accepted or not accepted by the automaton at the same time. Indeed no matter by what path did we come in state q if we will draw path corresponding to string x from it, we will know in which state we will come (so also know whether it is final or not).

Hence any state q has set of strings $X(q)$, which lead from it into one of the final states. This set is called the right context of state. It is defined not only for the state but also for the strings that it accepts (their right context coincides with the right context of state). It can be concluded that there are *not less* states in the automaton than different right context of the strings that it accepts (because for any string that can be extended to become accepted by automaton corresponds some path in it, and therefore, some state).

Let's assume that the automaton has two states q_1, q_2 such that $X(q_1) = X(q_2)$. We can remove the state q_2 and redirect transitions leading to it in a state q_1. Obviously, the set of accepted by automaton words will not change, therefore, we can continue this process until the number of states is not equal

to the number of different right contexts. Thus, *DFA is minimal if and only if the right contexts of all of its states are distinct.* □

Finally, knowing that facts let's return to suffix automaton.

Pretty easy to understand that in case of suffix automaton right conext $X(a)$ of string a has mutual correspondence with the set of right positions of occurrences of the string a in the string s. Indeed, if ax is accepted by automaton, that is, it is a suffix of s, then $s = yax$, and the string x can be matched with position $|s| - |x| - 1$. Thus, each state of the automaton accepts strings with the same set of *right* positions of occurrences in s and vice versa all strings with such set of positions is accepted by this state.

5 Relation between suffix automaton and suffix tree

Consider an edge in the suffix tree of string s, or more precise all substrings of s which corresponds to the "internal" vertices of edge (ie, one of the "compressed" in compression), or to vertex that is bottom end of the edge. Let's show that *for any string x and any pair of strings a, b from this set xa and xb are or are not prefixes of s simultaneously* (in other words, their sets of *left* positions of occurrences are same).

Let $|a| < |b|$. Then, a is the ancestor of b in trie, that is, its prefix, then its set of occurrences for sure contains all occurrences of the string b. Suppose there is a position $|x|$, in which there is an occurrence of the string a, but not occurrence of string b. Consider the string a', which is the maximum prefix of b, which can be found in that position. If a' is not limited by the end of string, it can be extended by at least two different characters still being a substring of s. Hence, the corresponding vertex in the tree has degree more than two and must split an edge into two which conflicts with the assumption that a and b are taken from the same. If the a' can not be continued because s ends there, it still must split an edge, as it is a suffix of the whole string and its vertex is not removed during compression. □

Similarly it can be shown that for any string a all strings b with the same set of left positions of occurrences are on the same edge with a. Thus, we found that *for any state q of suffix automaton of string s there exists such vertex of q' in suffix tree of string s^T such that the set of strings accepted by state q coincides with the set of strings, such that their corresponding vertex in the suffix tree lies on the edge leading into the vertex q' (including the string corresponding to q').* □

From this fact we can also find out following properties based on the structure of reversed substrings of s in suffix tree of string s^T:

1. If two strings a and b are accepted by state q and $|a| < |b|$, then a is the suffix of b.

2. If strings a and b are accepted by state q and $|a| < |b|$, then state q also accepts all strings c such that $|a| < |c| < |b|$ and c is the suffix of b.

3. Consider strings a and b, $|a| \leq |b|$. If a is the suffix of b, then $X(b) \subseteq X(a)$ or $X(a) \cap X(B) = \varnothing$ otherwise.

Thus, we have completely described states of automaton.

6 Construction of suffix automaton

Finally, consider the algorithm for suffix automaton construction. To do this, let's define the term called suffix link. Let the length of the shortest line, which is accepted by the state q is equal to . Then suffix link $link(q)$ leads from this state to a state that takes the same string without its first character. Referring again to the suffix tree we can understand that in this structure suffix link will lead to the ancestor of q. Thus, the suffix links form a tree, which corresponds to the suffix tree of the reversed string. Also let's denote the length of the longest line, which is accepted by the state q as $len(q)$. Obviously, the length of the shortest line of q in this case will be equal to $len(link(q)) + 1$. Just by the definition of suffix link.

Now let's add characters to the end of s one by one keeping correct suffix automaton and its suffix links. Suppose we have automaton of string s and s is accepted by state $last$. We want to update it so it will be automaton for string sc. So we need for each suffix of new string to have the final state which accepts it.

Let's add a state that accepts the whole string sc and call it new. Right context of string sc, obviously is just an empty string, it means that new will include only those suffixes that have exactly one occurrence in the string. All of these strings can be obtained by adding the character c to suffixes of s that are accepted by state, from which there are no transition by character c. Thus, to make that new suffixes accepted, we will need to "jump" by the suffix links and add transitions by the character c leading to the state new, until we come the root or find the state which already have some transition by character c.

In case we came to the root, *every* non-empty suffix of string sc is accepted by state new hence we can make $link(new) = q_0$ and finish our work on this step.

Otherwise we found such state q', which already has transition by character c. It means that all suffixes of length $\leq len(q') + 1$ are already accepted by some state in automaton hence we don't need to add transitions to state new anymore. But we also have to calculate suffix link for state new. The largest string accepted by this state will be suffix of sc of length $len(q') + 1$. It is accepted by state t at the moment, in which there is transition by character c from state q'. But state t can also accept strings of bigger length. So, if $len(t) = len(q') + 1$, then t is the suffix link we are looking for. We make $link(new) = t$ and finish algorithm.

Otherwise t is a state in automaton which accepts its suffixes as well as some other strings. Because of this we can't determine whether it is final or not. To solve the case we have to split off from t some state t', which will accept

every string which is accepted by t and has length $\leq len(q') + 1$, hence thouse suffixes of sc which maked trouble. To do this let's copy in t' all transitions and suffix link of t but set $len(t')$ to $len(q') + 1$. After this we can say that $link(new) = link(t) = t'$.Finally to redirect pathes corresponding to such strings from t to t' we will jump through the suffix links of q' until transition from that states by character c leads to t and redirect those transitions to t'. After considering all this cases suffix automaton will be finally obtained.

Code in C++ which performs this algorithm:

```
const int maxn = 2e5 + 42; // Maximum amount of states
map<char, int> to[maxn]; // Transitions
int link[maxn]; // Suffix links
int len[maxn]; // Lengthes of largest strings in states
int last = 0; // State corresponding to the whole string
int sz = 1; // Current amount of states

void add_letter(char c) // Adding character to the end
{
    int p = last; // State of string s
    last = sz++; // Create state for string sc
    len[last] = len[p] + 1;
    for(; to[p][c] == 0; p = link[p]) // (1)
      to[p][c] = last; // Jumps which add new suffixes
    if(to[p][c] == last)
    { // This is the first occurrence of c if we are here
        link[last] = 0;
      return;
    }
    int q = to[p][c];
    if(len[q] == len[p] + 1)
    {
      link[last] = q;
      return;
    }
    // We split off cl from q here
    int cl = sz++;
    to[cl] = to[q]; // (2)
    link[cl] = link[q];
    len[cl] = len[p] + 1;
    link[last] = link[q] = cl;
    for(; to[p][c] == q; p = link[p]) // (3)
      to[p][c] = cl; // Redirect transitions where needed
}
```

7 Complexity of algorith

Let's show that algorithm consumes $O(n)$ time and memory. On each step there are three things which works not in proper $O(1)$:

1. Jumps by the links of *last* for creating transitions to state *new*.
2. Copying transitions of t to t'.
3. Jumps by the links of q' for redirecting transitions leading to t from state t to state t'.

In first two cases we create new transition in automaton. Let's show that there is only $O(n)$ transitions. Let's split all transitions $\delta(v, c) = u$ leading from v to u by character c into two classes - "continuous", for which $len(v) + 1 = len(u)$ and all remaining.

There are no more than $O(n)$ continuous states because every state except root has exactly one continuous transition leading to it. And as we already know there are only $O(n)$ states in automaton.

Consider now the non-continuous transitions. Each such transition can be associated with the string acb, where a is the largest string accepted by v, and b is the largest string that corresponds to some path from u. This acb string is a suffix of s (otherwise we will have an opportunity to extend the b to the right). Additionally $|a| = Len(v)$, hence it can be concluded that $|a|$ corresponds to some path which is made only of contiguous transitions. So, for arbitrary suffix we can determine non-continuous transition considering it is the first one we would met if we "feed" suffix to the automaton. Hence, this is mutual relation, so the amount of non-contiguous transitions in automaton is not greater then amount of different suffixes of string which. Thus we can conclude that there is only $O(n)$ transitions in automaton.

Finally, let's prove that third case also takes $O(n)$ time. Let's for convenience call the value of $link(link(q))$ as the second suffix link to the state q. Also, we will use the following notation: $last$ - state which accepts s, new - state which accepts sc, p - state, which "jumps" by suffix links in a cycle (you can see it in the code). Since sc could not occur in the suffix automaton of s, there are no transitions from state $last$ by character c, so in cycle (1) we will do at least one step. Hence $len(link(p)) \leq len(link(link(last)))$ (indeed, initially $p = last$, after the first iteration, $p = link(last) \to len(link(p)) = len(link(link(last)))$, in all successive iterations $len(link(p))$ strictly decreases, hence, inequality from above is true.

When we came out of the loop (1), we have $\delta(p, c) = q$. Obviously, if there is a transition from the state p in the state q, then if we append the symbol c to the shortest string which is accepted by p (its length is equal to $len(link(p)) + 1$), we will get a string that will not be shorter than the shortest line, accepted by the state q, (its length is equal to $len(link(q)) + 1$. That is, $len(link(p)) + 2 \geq len(link(q)) + 1 \to len(link(p)) + 1 \geq len(link(q))$.

After exit from cycle (1) wherever we put the suffix link of new, the second suffix link will be exactly $link(q)$. Hence $link(q) = link(link(new)) \to len(link(q)) = len(link(link(new)))$.

Now let's take look at the cycle (3). It will run till $\delta(p, c) = q$, that is, as mentioned above $len(link(p)) + 1 \geq len(link(q))$. Also as we know initially $len(link(link(last))) \geq len(link(p))$. Since as mentioned above $len(link(q)) = len(link(link(new)))$, and at each step $len(link(p))$ decreases, we see that the whole cycle run for no longer than $len(link(link(last))) - len(link(link(new)))$, that is, no more iterations then difference between lengthes of largest strings accepted by second suffix links of $last$ and new.

Finally, gathering all received together, we obtain the following inequality: $len(link(link(last)))+1 \geq len(link(p))+1 \geq len(link(q)) = len(link(link(new)))$,

hence $len(link(link(last))) + 1 \geq len(link(link(new)))$, which means that after each step length of second suffix link either decreased or increased by exactly 1 (which means that total decrease will not exceed $O(n)$). The linear time consume of the algorithm is proved! $\square$

8 Applications

1. **Amount of distinct substrings.** Given string s, you need to find the amount of its distinct substrings. Each states accepts strings of lengthes from $len(link(q)) + 1$ to $len(q)$. In total it accepts $len(q) - len(link(q))$ strings. We will have the answer if we sum up this number over all states.

   Exercise: solve this problem in $O(n)$, considering that characters are added one by one to s and after each new you have to determine the amount of distinct substrings of s.

   Exercise:* Consider previous task but now we also have queries of erasing letters from the beginning of the string. You have to answer such queries. Complexity of this algorithm still have to be linear. *Hint:* sometimes Ukkonen algorithm is helpful too.

2. **Search of substrings in text.** By feeding a string through to automaton, we can tell whether it is occur in the text. Suppose we want to know some information about this occurrences. For example, we want to know any particular occurrence. As we already know, every occurrence of the string corresponds to x such that ax is a suffix of s. Or in other words to some way from q to some final state. With dynamic programming on automaton as on the acyclic directed graph, we can find the length of some of such ways (for example, a minimum or maximum length). Note that similar dynamic programming can be used to count many other useful values, such as the number of strings in the right context of state (or, equivalently, the number of occurrences of a string from state in string s).

   An alternative solution would be to apply the suffix link tree, which, as we know, is a suffix tree for s^T. As we mentioned at the beginning, any substring of a string s is a prefix of one of the initial string's suffix. Thus, if we write in each "suffix" vertex index of corresponding to it suffix, then all positions of occurrence of the string t in s can be found in the subtree of vertex that corresponds to the string t. That means, in particular, this dp can be used to find the first or the last occurrence in s.

   Moreover, considering that we are working with tree now, we can traverse it in such a way that at every step we will keep the set of occurrence positions of strings from current state. To do this we apply the idea of fast set merging int which we always adding elements from a smaller set to a larger one, but not vice versa. Then this algorithm will require $O(n \log n)$ overall insert operations in the set, because every time we move

some particular number k from one set to another one, the size of its new set will be at least twice as size of old one, in which it was stored.

Exercise:* given string s. Find the longest string t such that it has at least 3 *disjoint* occurrences in string s.

3. **Largest common substring.** Given k strings $s_1, s_2, \ldots, s_k$. we have to find largest string t such that t occurs in every string s_i. One of possible solutions is to construct automaton for string $s_1 t_1 s_2 t_2 \ldots s_n t_n$, where t_i is the unique for each of k strings metacharacter. Now we can maintain dynamic programming $dp[q][i]$, which will equal to 1 if it is possible to reach from state q some state which has transition by character t_i without going through any other metacharacter. It will mean that q occurs in s_i. Like before we can calculate this dynamic programming with depth-first search over directed acyclic graph. Total complexity will be $O(k \cdot \sum |s_i|)$.

   Exercise:* solve this problem in $O(\sum |s_i|)$.