



quasisphere's blog

A short guide to suffix automata

 By [quasisphere](#), 13 months ago,  

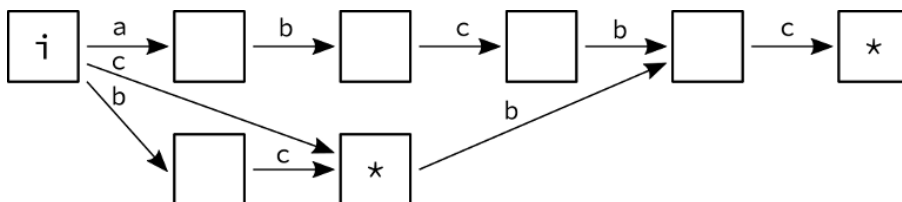
A short guide to suffix automata

This is supposed to be a short intro to suffix automata — what they are, how they are constructed, and what can they do? It seems that people have had trouble finding an explanation in English (the most popular source mentioned is [e-maxx.ru](#)). I don't speak Russian, so this will be mainly based on the paper [Automata for Matching Patterns](#) by Crochemore and Hancart.

Disclaimer: I am not an expert in the subject, and I am yet to use the automaton in an actual competition. Comments and corrections are welcome!

What is a suffix automaton?

A **suffix automaton** A for a string s is a minimal finite automaton that recognizes the suffixes of s . This means that A is a directed acyclic graph with an initial node i , a set of *terminal* nodes, and the arrows between nodes are labeled by letters of s . To test whether w is a suffix of s it is enough to start from the initial node of i and follow the arrows corresponding to the letters of w . If we are able to do this for every letter of w and end up in a terminal node, w is a suffix of s .



The above picture shows an automaton for the string "abcbc". The initial node is marked with 'i', and the terminal nodes are marked with asterisks.

To better understand suffix automata, let us consider the following game: A string s is given. Your friend has chosen a substring w of s and starts reciting the letters of w one by one from the beginning. You have a serious case of amnesia and can only remember the last letter that your friend has said. When your friend has reached the end, can you tell whether the word w is a suffix of the string s ?

To solve the problem let's do the following: At each stage we remember all the locations in s where we could be based on the previous letters. When the next letter comes in, we just check which of the locations have the correct next letter and form our next set of possible locations. The final set of locations are the endpoints of all the occurrences of w in s . Of course if the end of the string s is among these, w is a suffix of s .

For example if our string is "abcbc" and our friend has picked the string "bcbc", the possible lists of locations develop as follows (marked with upper case letters): aBcBc, abCbC, abCbC, abCbC.

A bit of thinking will convince you that remembering these locations is not only enough but also necessary. Thus the nodes of the suffix automaton should correspond exactly to the possible sets of locations that can occur during the game (when played for all possible substrings w). This also gives a (slow) way of constructing the suffix automaton: Go through the game for every suffix of s and at every step add nodes and edges to the automaton if they

→ Pay attention

Before contest
[Codeforces Round #379 \(Div. 2\)](#)
 8 days

Like 34 people like this.

→ gbuenoandrade

Rating: **1900**
 Contribution: 0

- [Settings](#)
- [Blog](#)
- [Teams](#)
- [Submissions](#)
- [Groups](#)
- [Talks](#)
- [Contests](#)



gbuenoandrade

→ Top rated

#	User	Rating
1	tourist	3580
2	Petr	3481
3	TooDifficult	3211
4	matthew99	3100
5	dotorya	3094
6	eatmore	3030
7	Endagorion	3020
7	W4yneb0t	3020
9	PavelKunyavskiy	3007
10	Um_nik	2999

[Countries](#) | [Cities](#) | [Organizations](#)
[View all →](#)

→ Top contributors

#	User	Contrib.
1	gKseni	182
2	Errichto	173
3	Petr	164
4	rng_58	159
5	Zlobober	155
6	Swistakk	154
7	Edvard	150
8	csacademy	148
9	zscoder	145
10	Xellos	144

[View all →](#)

→ Find user

Handle:

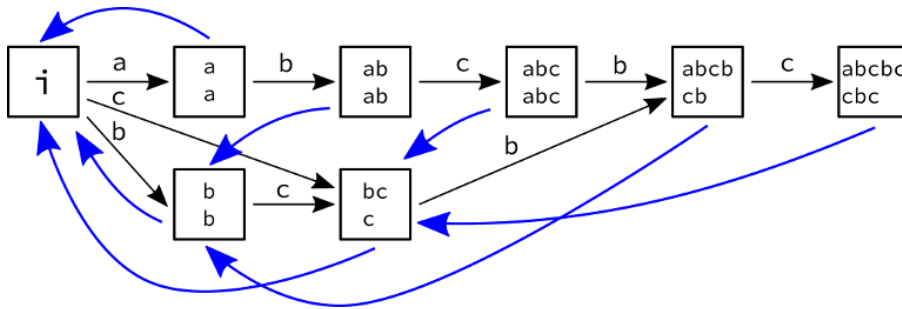
Find

don't yet exist.

We will say that two substrings u and v of s are equivalent if running u and v through the suffix automaton brings them to the same node, which by the above analysis is the same as saying that the endpoints of the occurrences of u and v in s agree. Thus one could also say that the nodes of the suffix automaton correspond to the equivalence classes of substrings of s under this relation.

The suffix tree and suffix links

Assume that u and v are the shortest and longest words in their common equivalence class. Then deleting the first letter of u will result in a larger set of possible endpoints, and adding a letter to the front of v will result in a smaller set. The words in the equivalence class are exactly those that are between u and v . The choice of the letter one can add in front of v induces a tree structure on the nodes of the suffix automaton. The removal of the first letter of u then means moving to the parent in this tree. These links to the parent are called **suffix links**, and they are useful both in constructing the suffix automaton and in applications.



The above picture shows in each node the longest and shortest strings in the equivalence class, as well as the suffix links (in blue).

Notice that one can find the terminal nodes of a suffix automaton by starting from the node corresponding to the whole string s and following the suffix links up to the initial node.

One may note that the tree that appears is actually the **Suffix tree** of the reversed string s . Risking potential confusion, we will refer to our tree also as the 'suffix tree'.

Constructing the suffix automaton

The suffix automaton can be constructed in linear time via an online algorithm that adds characters of s one by one and updates the automaton.

To understand the algorithm, it is necessary to consider how the equivalence classes change when adding a letter x to the end of the string. Consider moving in the suffix tree from the root along the path obtained by adding letters from the back of the new string one by one. Each such move either

1. stays in the current equivalence class,
2. moves down the tree to the next equivalence class, or
3. is impossible

When the case 3 happens (it will happen because the whole new string is not in the old tree), there are two possibilities:

1. we are in the longest string of some equivalence class q but there is no link forward to the next node
2. the next longest string in the class q has the wrong first letter

In the first case we may simply add a new equivalence class r containing the remaining new suffixes that could not be found in the tree. It will have a suffix link to q . In the second case we will have to add two new equivalence classes:

1. r containing the remaining new suffixes
2. q' containing the strings in q that are suffixes in the new string (i.e. were found before the search ended)

The strings in q' are of course removed from q , splitting the original class in two pieces.

→ Recent actions

- [gKseni](#) → [Codeforces announces contest for the best motto for laptop stickers](#)
- [manishbisht](#) → [Google APAC 2017 — Round E](#)
- [shashank21j](#) → [Sears Dots and Arrows Coding Contest\[HackerRank\]](#)
- [temp_test](#) → [Programming community](#)
- [AnasAbbas](#) → [Why not always use long long](#)
- [MikeMirzayanov](#) → [All-Siberian Olympiad 2016 \(XVII Open Cup, GP of Siberia\)](#)
- [Petr](#) → [A Fourier combination week](#)
- [minimario](#) → [Problems that are Dijkstra with a Twist](#)
- [anand_20](#) → [HackerEarth CodeMonk \[Searching\] Contest](#)
- [abdelkarim](#) → [Egyptian Collegiate Programming Contest 2016](#)
- [KaiZeR](#) → [Editorial Codeforces Round #247 \(Div. 2\)](#)
- [theLifter](#) → [C++: What is the best way to read a line containing of string followed by numbers?](#)
- [barish2003](#) → [Problem Chords \(geometry\)](#)
- [zemen](#) → [Week of Code 25 on HackerRank](#)
- [atatomir](#) → [Fast Fourier Transform and variations of it](#)
- [MikeMirzayanov](#) → [2016-2017 CT S03E08: Codeforces Trainings Season 3 Episode 8](#)
- [sidhant](#) → [Tutorial on FFT — The tough made simple](#)
- [now_you_see_me](#) → [would you please help me to find out the error in my code for lightoj 1220](#)
- [dalex](#) → [Gym — Samara University ACM ICPC 2016-2017 Quarterfinal Qualification Contest](#)
- [Apptica](#) → [CodeRing — II An ICPC Style Practice Contest](#)
- [piloop](#) → [Codeforces Round #119 — Editorial](#)
- [orz_CWY](#) → [How to get large test datas](#)
- [r3gz3n](#) → [HackerEarth Collegiate Cup Final Round Results](#)
- [abacadaea](#) → [Editorial for Codeforces #174](#)
- [adamant](#) → [Manacher's algorithm and code readability](#)

[Detailed →](#)

For instance consider adding 'c' to the end of "abcb". In the original tree when we start following the letters from the back of the new string "abcbc", we will come to the equivalence class q with longest string "abc". The search will terminate at "bc" since the next letter we would try is 'c', which is different from 'a'. Thus we add the equivalence classes q' and r with longest strings "bc" and "abcbc" respectively.

Now the class q' becomes the parent of q and r in the suffix tree, and the parent of q' is the old parent of q .

Finally we have to consider the edges in the automaton. First of all we should add edges to the new class r from every class of every suffix of the original string that does not have an edge labeled with the added letter x . These can be found by starting from the class of the whole original string and following the suffix links until a class p with edge labeled x appears. Following the edge from p also gives us the class q .

If we have to split q , there will be additional changes to the edges. The edges from q will stay the same, and these will also be copied to be the edges from q' . The nodes with edges to q that now should point to the shorter strings in q' instead can be found among p and its parents by following suffix links.

Implementation

Below is an implementation of the construction procedure in C++. It might appear on the surface that this is an $O(n^2)$ algorithm, but in fact the inner loops are run only $O(n)$ times. I will skip this proof.

```
struct SuffixAutomaton {
    vector<map<char,int>> edges; // edges[i] : the labeled edges from node
    i
    vector<int> link;           // link[i] : the parent of i
    vector<int> length;         // length[i] : the length of the longest
    string in the ith class
    int last;                  // the index of the equivalence class of
    the whole string

    SuffixAutomaton(string s) {
        // add the initial node
        edges.push_back(map<char,int>());
        link.push_back(-1);
        length.push_back(0);
        last = 0;

        for(int i=0;i<s.size();i++) {
            // construct r
            edges.push_back(map<char,int>());
            length.push_back(i+1);
            link.push_back(0);
            int r = edges.size() - 1;

            // add edges to r and find p with link to q
            int p = last;
            while(p >= 0 && edges[p].find(s[i]) == edges[p].end()) {
                edges[p][s[i]] = r;
                p = link[p];
            }
            if(p != -1) {
                int q = edges[p][s[i]];
                if(length[p] + 1 == length[q]) {
                    // we do not have to split q, just set the correct suffix link
                    link[r] = q;
                } else {
                    // we have to split, add q'
                    edges.push_back(edges[q]); // copy edges of q
                    length.push_back(length[p] + 1);
                    link.push_back(link[q]); // copy parent of q
                }
            }
            last = r;
        }
    }
};
```

```

    int qq = edges.size()-1;
    // add qq as the new parent of q and r
    link[q] = qq;
    link[r] = qq;
    // move short classes pointing to q to point to q'
    while(p >= 0 && edges[p][s[i]] == q) {
        edges[p][s[i]] = qq;
        p = link[p];
    }
}
}
}
last = r;
}
}
};

```

Note that this does not calculate the terminating nodes of the suffix automaton. This can however be done simply by starting from the last node and following the suffix links:

```

vector<int> terminals;
int p = last;
while(p > 0) {
    terminals.push_back(p);
    p = link[p];
}

```

Applications

Typical applications of the suffix automaton are based on some sort of dynamic programming scheme on top of the automaton, but let's start with the trivial ones that can be implemented on the plain automaton.

Problem: Find whether a given string w is a substring of s .

Solution: Simply run the automaton.

```

SuffixAutomaton a(s);
bool fail = false;
int n = 0;
for(int i=0; i<w.size(); i++) {
    if(a.edges[n].find(w[i]) == a.edges[n].end()) {
        fail = true;
        break;
    }
    n = a.edges[n][w[i]];
}
if(!fail) cout << w << " is a substring of " << s << "\n";

```

Problem: Find whether a given string w is a suffix of s .

Solution: Construct the list of terminal states, run the automaton as above and check in the end if the n is among the terminal states.

Let's now look at the dp problems.

Problem: Count the number of distinct substrings in s .

Solution: The number of distinct substrings is the number of different paths in the automaton. These can be calculated recursively by calculating for each node the number of different paths starting from that node. The number of different paths starting from a node is the sum of the corresponding numbers of its direct successors, plus 1 corresponding to the path that does not leave the node.

Problem: Count the number of times a given word w occurs in s .

Solution: Similar to the previous problem. Let p be the node in the automaton that we end up while running it for w . This time the number of times a given word occurs is the number of paths starting from p and ending in a terminal node, so one can calculate recursively the number of paths from each node ending in a terminal node.

Problem: Find where a given word w occurs for the first time in s .

Solution: This is equivalent to calculating the longest path in the automaton after reaching the node p (defined as in the previous solution).

Finally let's consider the following problem where the suffix links come handy.

Problem: Find all the positions where a given word w occurs in s .

Solution: Prepend the string with some symbol '\$' that does not occur in the string and construct the suffix automaton. Let's then add to each node of the suffix automaton its children in the suffix tree:

```
children=vector<vector<int>>(link.size());
for(int i=0;i<link.size();i++) {
    if(link[i] >= 0) children[link[i]].push_back(i);
}
```

Now find the node p corresponding to the node w as has been done in the previous problems. We can then dfs through the subtree of the suffix tree rooted at p by using the children vector. Once we reach a leaf, we know that we have found a prefix of s that ends in w , and the length of the leaf can be used to calculate the position of w . All of the dfs branches correspond to different prefixes, so no unnecessary work is done and the complexity is $O(|s| + |w| + \text{size of output})$.

THE END

suffix automata, guide

▲ +180 ▼ ☆ [quasisphere](#) 📅 13 months ago 💬 [13](#)



Comments (13)

[Write comment?](#)



jcg

13 months ago, # | ☆

▲ +9 ▼

maybe a few more complex examples of applications would be nice... thank you for the post..

→ [Reply](#)



Cybuster

13 months ago, # | ☆

▲ 0 ▼

thanks for writing this, as you said there are only few options to learn from :))

→ [Reply](#)



reality

13 months ago, # | ☆

▲ 0 ▼

how to solve <http://codeforces.com/problemset/problem/128/B> with suffix automata?

→ [Reply](#)

13 months ago, # ^ | ☆

← Rev. 2 ▲ +8 ▼

Build suffix automaton and count for each state how many times does it occur in string. After that with dfs on DAG you can count for every state how many paths can you get from it. Finally you can find the k th lexicographically path in DAG using this information (in state you check every output edge and try check if the answer is in there by comparing amount of path there and k . If k is bigger, you



should subtract this amount from it otherwise you will go in that

adamant

should subtract this amount from it otherwise you will go in that state).

Also can be solved by reversing string and obtaining suffix tree for initial string.

My code: 7361297. Pretty old one with bad style, sorry.

→ [Reply](#)

13 months ago, # ^ | ☆

▲ 0 ▼

Find for every node how many different substrings one can reach from that node. This can be done recursively: First of all calculate for every node `i` how many paths from that node there are to terminal nodes. Call this `occurrences[i]`. This is the number of occurrences of each of the substrings in the equivalence class in the original string. Next calculate for every node the number of substrings that can be reached from this node, call it `words[i]`. For node `i` this would be `words[i] = occurrences[i] + sum(words[j], j is a successor of node i)`. Now do a search in the lexicographical order by using the values `words[i]`. See the submission below.

<http://codeforces.com/contest/128/submission/13583093>

→ [Reply](#)



quasisphere



Colorless_coder

10 months ago, # | ☆

← Rev. 3 ▲ 0 ▼

Great post

→ [Reply](#)



saisumit

9 months ago, # | ☆

▲ +1 ▼

Have a look at this <https://saisumit.wordpress.com/2016/01/26/suffix-automaton/>

→ [Reply](#)



omarfarhat97

9 months ago, # | ☆

← Rev. 2 ▲ 0 ▼

mohieddine Interesting (D/N) Finite Automaton..

→ [Reply](#)

9 months ago, # | ☆

← Rev. 2 ▲ 0 ▼

Nice article. Thank you. By the way I had a problem and I have written about it here - <http://codeforces.com/blog/entry/23593>

Could anyone help me please?

EDIT: Got the mistake.

→ [Reply](#)

7 months ago, # | ☆

▲ 0 ▼

Number of distinct substring is also equal to

$\sum_{i=1}^{size} length[i] - length[link[i]]$ because each node corresponds to different substrings with lengths in range $[length[link[i]] + 1, length[i]]$.

→ [Reply](#)



ACMath

5 months ago, # | ☆

▲ 0 ▼

Wow, thanks a lot for the post. Now i am clearly understand the subject.

→ [Reply](#)



arrch

5 months ago, # | ☆

▲ 0 ▼

Please help. I dont get this, "At each stage we remember all the locations in s where we could be based on the previous letters. When the next letter comes in we just check which of the locations have the correct next letter



spk

comes in, we just check which of the locations have the correct next letter and form our next set of possible locations."

Thanks !

→ [Reply](#)

5 months ago, # | ☆

▲ 0 ▼

Does anyone know if it's possible to solve [this](#) problem with suffix automaton? And if so, how?



dcms2

If the queries were disjoint, I'd know how to solve it; we could just build an automaton for each query. The problem is if two queries overlap, how to remove letters from the beginning of the previous query string from the automaton? Assuming the queries are sorted by start index.

Thanks

→ [Reply](#)

[Codeforces](#) (c) Copyright 2010-2016 Mike Mirzayanov
The only programming contests Web 2.0 platform
Server time: Nov/08/2016 00:55:22^{UTC-2} (c2).
Desktop version, switch to [mobile version](#).

