

Toggle Navigation [Akshay Jaggi](#)

Search

Everything ▼

Go

- [Home](#)
- [Blog](#)
- [Resume](#)
- [Contact](#)

Suffix Automata

- [Home](#)
- [Blog](#)
- Suffix Automata

Posted by: [Akshay Jaggi](#) 2 years, 7 months ago

([4 comments](#))

Abstract

I found out about Suffix Automata (SA) while reading accepted solutions of a problem we were unable to solve in one of the Codeforces-GYM practice sessions. Most of the solutions were using Suffix Arrays, but this particular solution. It was neat, short and precise. I started looking up more on Suffix Automata. It has been used mostly by Russian coders, and the only available and highly quoted resource I found is in Russian. It took me about a week on the Yandex translated resource, and countless re-reads before I was finally able to make sense of what was written.

Suffix Automata is one of the few very elegant data structures I know of. Coarsely, you can consider it as a data structure which contains information about all the substrings of a given string. You can solve a wide range of problems using it and it is linear in memory and construction time. For example, from the definition, you know you can solve the trivial problem of checking if the given strings occur as a substring of a given text, in linear time.

This blog post is long overdue now. I have been waiting to write this for three months.

Definition

A **suffix automaton**, also known as Directed-Acyclic-Word-Graph (DAWG), for a string **S** is the **minimal** deterministic finite automaton (DFA) that accepts all suffixes of the given string **S**.

Observation 1: If you mark all the states of the DFA as accepting, you can match any substring of the given string **S**.


Observation 2: If you want to match the empty string ϵ , mark the initial state as accepting, otherwise, mark it as non-accepting.

Note 1: If you don't know what a DFA or an NFA is, I would recommend that you stop reading now and take up a course on Formal Methods. Although describing it here would help with the description of SA, I'm not doing it for the purposes of brevity and clarity.


Note 2: The **minimality** condition is a very important property, as we will see ahead.

Note 3: I'm using the usual definition of a DFA where DFA is a five-tuple $(Q, \Sigma, \delta, q_0, F)$. Therefore, the initial state will be shown as q_0 , etc.

Examples

 $S = \text{"aaa"}$

$S = \text{"aaa"}$

 $S = \text{"ab"}$

$S = \text{"ab"}$

$S = \text{"abbb"}$

More Definitions and Properties

Endpos

Let t be a substring of string S . Then, $\text{endpos}(t)$ is defined as:

$\text{endpos}(t)$: The set of all positions in the string S where the substring t ends.

For example, let

$S = \text{"abcbdc"}$

then,

$\text{endpos}(\text{"bc"}) = \{3, 6\}$
 $\text{endpos}(\text{"abc"}) = \{3\}$

Endpos-Equivalence

Two substrings, $t1$ and $t2$, are endpos-equivalent if and only if

$\text{endpos}(t1) = \text{endpos}(t2)$

For example, in the previous string S ,

$\text{endpos}(\text{"c"}) = \text{endpos}(\text{"bc"}) = \{3, 6\}$
 $\text{endpos}(\text{"bc"}) \neq \text{endpos}(\text{"abc"})$

Thus, "bc" and "c" are endpos equivalent, but "bc" and "abc" are not.

Observation 3: We can divide the set of all non-empty substrings of String S into different endpos-equivalence classes.

Axiom 1: For each endpos-equivalence class, we have a state in the Suffix Automata. All substrings belonging to a given equivalence class, end at the state defined by that equivalence class (when matched by the DFA).

For now just take it as an axiom and remember it by heart. (I'm still figuring out a way to say this better.)

This is the most important part and this is the part where, I believe, the actual beauty of the SA construction lies. As you will see ahead, SA construction involves two different data structures, which individually look quite independent, but they help in building each other incrementally. This is the part that links both the data structures.

Lemma 1: Two non-empty substrings $t1$ and $t2$, with $\text{length}(t1) \leq \text{length}(t2)$, are endpos-equivalent if and only if $t1$ occurs in S only as a suffix of $t2$.

Proof: This is simple to see. If the ending positions of substrings $t1$ and $t2$ are the same, and $\text{length}(t1) \leq \text{length}(t2)$, then $t1$ can only occur as a suffix of $t2$. Other way around, if $t1$ only occurs in S as a suffix of $t2$, ending positions of $t1$ would be the same as ending positions of $t2$.

Lemma 2: Given two non-empty substrings $t1$ and $t2$, with $\text{length}(t1) \leq \text{length}(t2)$, then

$\text{endpos}(t2) \subseteq \text{endpos}(t1)$... if $t1$ is a suffix of $t2$
 $\text{endpos}(t1) \cap \text{endpos}(t2) = \emptyset$... otherwise

Proof: Suppose $\text{endpos}(t1)$ and $\text{endpos}(t2)$ have at least one element common, say z . This means that they both end at z . And since, $\text{length}(t1) \leq \text{length}(t2)$, therefore $t1$ is a suffix of $t2$. Now, since $t1$ is a suffix of $t2$, $t1$ ends at all the places where $t2$ ends, and maybe a few more places. Therefore, $\text{endpos}(t2) \subseteq \text{endpos}(t1)$.

Lemma 3: Suppose the length of the minimum length substring in a given equivalence class is a and the length of the maximum length substring is b . Then,

- The equivalence class contains one and only one substring for each of the lengths in the range $[a, b]$.
- Every substring, say t , in the equivalence class is a proper suffix of all the other substrings in the equivalence class with length greater than $\text{length}(t)$.

For example,

$\{\text{"abcd"}, \text{"bcd"}, \text{"cd"}\}$ is a valid equivalence class.
 $\{\text{"abcd"}, \text{"cd"}\}$ is **not** a valid equivalence class.
 $\{\text{"abcd"}, \text{"cd"}, \text{"bd"}\}$ is **not** a valid equivalence class.

Proof: Let Q be some endpos equivalence class with more than one string (the proof is trivial for those with one string). From lemma 1, we have that two **different** strings are endpos-equivalent if and only if one is a **proper** suffix of the other. Therefore, two different strings of the same length cannot be in an equivalence class.

Now, we need to show that there is one string for each of the lengths between a and b , in Q . Let c be such that $a \leq c \leq b$. Let w be the string of length b in Q . Therefore, from lemma 2, we have

$\text{endpos}(w) \subseteq \text{endpos}(\text{suffix of } w \text{ of length } c) \subseteq \text{endpos}(\text{suffix of } w \text{ of length } a)$

But, we already know that

$\text{endpos}(w) = \text{endpos}(\text{suffix of } w \text{ of length } a)$

as both belong in the same equivalence class Q .

Therefore,

$\text{endpos}(w) = \text{endpos}(\text{suffix of } w \text{ of length } c) = \text{endpos}(\text{suffix of } w \text{ of length } a)$

Suffix Link Tree

Suffix Link Tree is the second data structure I was talking about.

Suffix Link

Let's take an equivalence class Q . Let w be the longest string in this equivalence class, and let this equivalence class cover a range of lengths $[a, b]$. Therefore, $\text{length}(w) = b$ and all suffixes of w of length from a to $b-1$ are also part of this equivalence class.

But what about the suffix of length $a-1$ of w ? It lies in a different equivalence class. Suffix Link links these two equivalence classes.

More precisely, $\text{link}(Q)$ points to the endpos-equivalence class defined by "the longest suffix of w which does not belong to Q ".

Observation 4: The suffix links form a tree, with the root as q_0 (the initial state of the suffix automata).

The root is the equivalence class defined by the empty string.

$\text{endpos}("") = \{ 0, 1, 2, \dots, \text{strlen}(S) \}$

For any given state, as we travel along the suffix links, we keep on reducing the string length, finally reaching the root node. Lastly, there are no cycles, as only one suffix link exists for any node (parent pointers).

Let's see the suffix automata and the suffix link tree for $S = \text{"abcdbc"}$.

Suffix Link Tree

Suffix Automata

Lemma 4: Let Q be a node, and let $Q_1, Q_2, Q_3 \dots Q_N$ be its children. That is, $\text{link}(Q_1) = Q$, $\text{link}(Q_2) = Q$, and so on. Then,

$$\begin{aligned} \text{endpos}(Q_1) \cup \text{endpos}(Q_2) \dots \cup \text{endpos}(Q_N) &\subset \text{endpos}(Q) \\ \text{endpos}(Q_i) \cap \text{endpos}(Q_j) &= \emptyset \quad (i \neq j) \end{aligned}$$

From the definition of suffix links and lemma 2, we have $\text{endpos}(Q_i) \subset \text{endpos}(Q)$ for all i . Thus, $\text{endpos}(Q_1) \cup \text{endpos}(Q_2) \dots \cup \text{endpos}(Q_N) \subset \text{endpos}(Q)$.

Let, $\text{endpos}(\mathbf{Q_i}) \cap \text{endpos}(\mathbf{Q_j}) \neq \emptyset$. Then, from Lemma 2, we know that one is contained in the other. Without loss of generality, let $\text{endpos}(\mathbf{Q_i}) \subset \text{endpos}(\mathbf{Q_j})$ (proper subset since $i \neq j$). From the definition of suffix links, we know that $\text{endpos}(\mathbf{Q_i}) \subset \text{endpos}(\text{link}(\mathbf{Q_i})) \subset \text{endpos}(\text{link}(\text{link}(\mathbf{Q_i}))) \dots$ and so on. Thus, $\mathbf{Q_j}$ must be one of $\text{link}(\text{link} \dots \text{link}(\mathbf{Q_i}))$. But, we have $\text{endpos}(\mathbf{Q_j}) \subset \text{endpos}(\mathbf{Q})$. Thus, $\text{endpos}(\mathbf{Q_j}) \subset \text{endpos}(\text{link}(\mathbf{Q_i}))$, which is a contradiction.

Observation 5: If we build a tree from the available sets of endpos-equivalence, such that the parent is a union of it's (disjoint) children, it will be **similar** in structure to the suffix link tree, apart from the condition that it will instead follow the property,

$$\text{endpos}(\mathbf{Q_1}) \cup \text{endpos}(\mathbf{Q_2}) \dots \cup \text{endpos}(\mathbf{Q_N}) = \text{endpos}(\mathbf{Q})$$

To see the difference, see the suffix link tree for $\mathbf{S} = \text{"aba"}$.

Building Algorithm

The algorithm is **online**. We build the Suffix Automata incrementally, adding one character every time.

Some definitions:

- Every state represents an equivalence class, which matches the substrings that belong to this equivalence class.
- Let \mathbf{v} be some state in the machine. Let $\text{longest}(\mathbf{v})$ be the longest substring that matches at \mathbf{v} , and let its length be $\text{len}(\mathbf{v})$. Let the length of the shortest substring that matches at \mathbf{v} be $\text{minlen}(\mathbf{v})$. Then,

$$\circ \text{minlen}(\mathbf{v}) = \text{len}(\text{link}(\mathbf{v})) + 1$$

- **last** points to the state that matches the entire string till this point (i.e., till the point the SA is built).

To ensure linearity in memory consumption, the data that we persist at every state is:

- Length of the longest string that matches at this state: len
- Suffix link: link
- DFA-transitions: next

Initial State

The SA consists of a single state, $\mathbf{q_0}$.

- $\text{len}(\mathbf{q_0}) = 0$

- $link(q_0) = \text{nullptr}$
- $next(q_0) = \{ \}$
- $last = q_0$

Adding a Character

Suppose we are adding character c to the Suffix Automata.

- Create a new state, say cur . Set $len(cur) = len(last) + 1$.
- Let $iter_variable$ be $last$. Iterate till $iter_variable$ has **no** transition on character c or $iter_variable$ is nullptr :
 - Add a transition on character c to cur in state $iter_variable$

$iter_variable.next(c) = cur$

- Travel along the suffix link

$iter_variable = link(iter_variable)$

- Let p be the value of $iter_variable$ at which there was a transition on seeing character c .
- If $p == \text{nullptr}$, assign $link(cur) = q_0$.
- Else, let q be the state to which there is a transition at state p on seeing character c . (i.e., $q = p.next(c)$)
- If $len(q) == len(p) + 1$, assign $link(cur) = q$.
- Else, make a copy of state q , say $clone$.
- Set $len(clone) = len(p) + 1$. Set $link(cur) = clone$ and $link(q) = clone$.
- Let $iter_variable$ be p . Iterate till $iter_variable$ has a transition on character c or $iter_variable$ is nullptr :
 - Change the transition on character c to $clone$ in state $iter_variable$.

$iter_variable.next(c) = clone$

- Set $last$ to cur .

To mark the Accepting States of our DFA, after building the complete SA, we can start from $last$, and travel along the suffix links, marking all encountered states as Accepting. (Why?)

Proof of Correctness and Linearity

I'll add the proofs in part-two of this blog post. I'm purposely not adding them here, to keep this part concise. I shall wait for some feedback, before I proceed with them.

Implementation

This is just a sample implementation. It can be optimised further for various purposes.

```
class SAutomata
{
    private:
        class SNode
        {
            public:
                int link, len;
                map<char,int> next;
                SNode()
                {
                    link = -1;
                    len = 0;
                }
                SNode(const SNode& other)
                {
                    link = other.link;
                    len = other.len;
                    next = other.next;
                }
        };
        vector nodes;
        int last, cur;

    public:
        SAutomata(string S)
            : nodes(2*S.size()),
              dp(2*S.size(), -1),
              last(0),
              cur(1)
        {
            for(int i=0; i<S.size(); i++)
                add_char(S[i]);
        }
        void add_char(char A)
        {
            trace(A);
```



```

int nw = cur++;
nodes[nw].len = nodes[last].len+1;
int p=last;
for(; p!=-1 && nodes[p].next.find(A)==nodes[p].next.end(); p=nodes[p].link)
{
    nodes[p].next[A]=nw;
}
if(p==-1)
{
    nodes[nw].link = 0;
}
else if(nodes[nodes[p].next[A]].len == nodes[p].len + 1)
{
    nodes[nw].link = nodes[p].next[A];
}
else
{
    int nxt = nodes[p].next[A];
    int clone = cur++;
    nodes[clone] = nodes[nxt];
    nodes[clone].len = nodes[p].len + 1;
    nodes[nxt].link = clone;
    nodes[nw].link = clone;
    for(; p!=-1
        && nodes[p].next.find(A)!=nodes[p].next.end()
        && nodes[p].next[A]==nxt;
        p=nodes[p].link)
    {
        nodes[p].next[A]=clone;
    }
}
last = nw;
}
};

```

Sample Problem

Let's see a very basic problem.

Count the number of distinct substrings that occur in a given string.

You can solve this problem in two ways.

1. The number of different substrings is equal to the number of different possible paths we can take in the Suffix Automata. The number of distinct paths can be quite high ($O(n^2)$), but we can use DP to save the number of paths possible from this state, and thus get the answer in $O(n)$.
2. Using Suffix Links. Each node accepts substrings from minlen to len, i.e. len-minlen+1 substrings, and each of these is unique.

Try to get this problem accepted on [SPOJ](#) using both 1 and 2.

You can check my submission [here](#). dfs() uses approach 1, while dfs2() uses approach 2.

Road Ahead

The part two of this blog post would be more focussed on questions and examples, the proof of correctness and linearity of the construction algorithm, converting the Suffix Automata to Suffix Tree and vice versa.

Till then, you can try out a few problems. For example, longest common substring in $O(n)$; the count, total length, positions of occurrence, and lexicographically k^{th} smallest of all distinct substrings in a given string, etc.

I look forward to some feedback, before I proceed with the second part.

References

1. http://e-maxx.ru/algo/suffix_automata (Translated version: https://translate.yandex.com/translate?url=http%3A%2F%2Fe-maxx.ru%2Falgo%2Fsuffix_automata&lang=ru-en)
2. https://en.wikipedia.org/wiki/Suffix_automaton
3. <http://static.googleusercontent.com/media/research.google.com/en//pubs/archive/35395.pdf>

- Tags:
- [Suffix](#),
- [Automata](#),
- [data-structures](#),
- [competitive programming](#),
- [strings](#),
- [DAWG](#)

Current rating: 4.7

- ○ ● 1

- ☐ 2
- ☐ 3
- ☐ 4
- ☐ 5

Rate

[Share on Twitter](#) [Share on Facebook](#)

- [Grant-Table User-Space Device | Progress Report →](#)

Comments



Arafat Dad Khan 2 years, 7 months ago

This seems to be wonderful article and Its surprising to see that this is not really as hard as i thought it would be.

Just one last request. I understood how the construction takes place and the three cases but can you please elaborate on the reasoning behind the construction. Also the proof of correctness and linearity and a few other examples would be a wonderful addition to this.

I know that this is too much to ask but if you are free sometime then it would be really cool if you could add a few things.

Thanks a lot

This made my day.

[Link](#) | [Reply](#)

Current rating: 5

- - ☐ 1
 - ☐ 2
 - ☐ 3
 - ☐ 4
 - ☐ 5

Rate

Comment deleted 6 months ago



Ramesh 1 year, 3 months ago

Vaah, akshay bhai, bahut khub. Very simple and good guide by you. Thanks again.

[Link](#) | [Reply](#)

Currently unrated

- - 1
 - 2
 - 3
 - 4
 - 5

Rate



amir 10 months ago

Thanks for very informative post

[Link](#) | [Reply](#)

Currently unrated

- - 1
 - 2
 - 3
 - 4
 - 5

Rate

New Comment

Name

required

Email

required (not published)

Website

optional

Comment

Recaptcha

Comment

Recent Posts

- [ताकत-ऐ-लब्ज - I](#)
- [Grant-Table User-Space Device | Progress Report](#)
- [Suffix Automata](#)

Archive

2016

- [November](#) (1)
- [May](#) (2)

Categories

- [Competitive Programming](#) (1)
- [Data Structures](#) (1)
- [FreeBSD](#) (1)
- [Google Summer of Code](#) (1)
- [Hindi](#) (1)
- [Open Source](#) (1)
- [Poetry](#) (1)
- [Xen](#) (1)

Tags

- [data-structures](#) (1)
- [competitive programming](#) (1)
- [Google Summer of Code](#) (1)
- [FreeBSD](#) (1)
- [Xen](#) (1)
- [Hindi](#) (1)
- [Poetry](#) (1)
- [Suffix](#) (1)
- [Automata](#) (1)
- [strings](#) (1)
- [DAWG](#) (1)

Authors

- [Akshay Jaggi](#) (3)

Feeds

[RSS](#) / [Atom](#)

- [Blog](#)
- [Resume](#)
- [Contact](#)

Powered by [Mezzanine](#) and [Django](#) | Theme by [Bootstrap](#)