Ukkonen's suffix tree algorithm in plain English?

I feel a bit thick at this point. I've spent days trying to fully wrap my head around suffix tree construction, but because I don't have a mathematical background, many of the explanations elude me as they start to make excessive use of mathematical symbology. The closest to a good explanation that I've found is *Fast String Searching With Suffix Trees*, but he glosses over various points and some aspects of the algorithm remain unclear.

A step-by-step explanation of this algorithm here on Stack Overflow would be invaluable for many others besides me, I'm sure.

For reference, here's Ukkonen's paper on the algorithm:
http://www.cs.helsinki.fi/u/ukkonen/SuffixT1withFigs.pdf

My basic understanding, so far:

- I need to iterate through each prefix P of a given string T
- I need to iterate through each suffix S in prefix P and add that to tree
- To add suffix S to the tree, I need to iterate through each character in S, with the iterations consisting of either walking down an existing branch that starts with the same set of characters C in S and potentially splitting an edge into descendent nodes when I reach a differing character in the suffix, OR if there was no matching edge to walk down. When no matching edge is found to walk down for C, a new leaf edge is created for C.

The basic algorithm appears to be $O(n^2)$, as is pointed out in most explanations, as we need to step through all of the prefixes, then we need to step through each of the suffixes for each prefix. Ukkonen's algorithm is apparently unique because of the suffix pointer technique he uses, though I think *that* is what I'm having trouble understanding.

I'm also having trouble understanding:

- exactly when and how the "active point" is assigned, used and changed
- what is going on with the canonization aspect of the algorithm
- Why the implementations I've seen need to "fix" bounding variables that they are using

EDIT (April 13, 2012)

Here is the completed source code that I've written and output based on jogojapan's answer below. The code outputs a detailed description and text-based diagram of the steps it takes as it builds the tree. It is a first version and could probably do with optimization and so forth, but it works, which is the main thing.

[Redacted URL, see updated link below]

EDIT (April 15, 2012)

The source code has been completely rewritten from scratch and now not only works correctly, but it supports automatic canonization and renders a nicer looking text graph of the output. Source code and sample output is at:

https://gist.github.com/2373868

algorithm | search | suffix-tree

edited **Jun 10 at 17:49**

Peter Mortensen
5,383 7 27 62

asked **Feb 26 at 11:30**

Nathan Ridley
6,272 3 29 66
87% accept rate

Did you take a look at the description given in Dan Gusfield's book? I found that to be helpful. – jogojapan Feb 27 at 2:10

I don't have a copy of that book unfortunately... – Nathan Ridley Feb 27 at 21:27

About the implementation and tests at gist.github.com/2373868. The code works not properly on string 'dedededeede$', because in your output suffix tree we cannot find suffix 'de$'. The reason is that the linked

dedddddeeddd$, because in your output suffix tree we cannot find suffix "dd$". The reason is that the linked node for active node node #3 is not null but node #1 (check the output for iteration 12). – mar_one May 15 at 11:54

1 The gist does not specify the license - can I change your code and republish under MIT (obviously with attributions)? – Yurik Oct 11 at 4:26

Yep, go for your life. Consider it public domain. As mentioned by another answer on this page, there's a bug that needs fixing anyway. – Nathan Ridley Oct 11 at 15:27

Was this post useful to you? Yes No

4 Answers

The following is an attempt to describe the Ukkonen algorithm by first showing what it does when the string is simple (i.e. does not contain any repeated characters), and then extending it to the full algorithm.

First, a few preliminary statements.

1. What we are building, is *basically* like a search trie. So there is a root node, edges going out of it leading to new nodes, and further edges going out of those, and so forth
2. **But**: Unlike in a search trie, the edge labels are not single characters. Instead, each edge is labeled using a pair of integers `[from,to]`. These are pointers into the text. In this sense, each edge carries a string label of arbitrary length, but takes only O(1) space (two pointers).

Basic principle

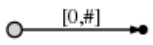
I would like to first demonstrate how to create the suffix tree of a particularly simple string, a string with no repeated characters:

```
abc
```

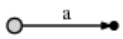
The algorithm **works in steps, from left to right**. There is **one step for every character of the string**. Each step might involve more than one individual operation, but we will see (see the final observations at the end) that the total number of operations is O(n).

So, we start from the *left*, and first insert only the single character `a` by creating an edge from the root node (on the left) to a leaf, and labeling it as `[0,#]`, which means the edge represents the substring starting at position 0 and ending at *the current end*. I use the symbol `#` to mean *the current end*, which is at position 1 (right after `a`).

So we have an initial tree, which looks like this:



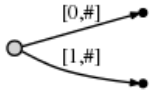
And what it means is this:



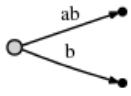
Now we progress to position 2 (right after `b`). **Our goal at each step** is to insert **all suffixes up to the current position**. We do this by

- expanding the existing a-edge to ab
- inserting one new edge for b

In our representation this looks like



And what it means is:



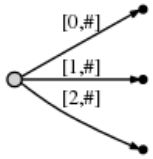
We observe two things:

We observe two things:

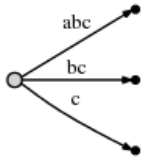
- The edge representation for `ab` is **the same** as it used to be in the initial tree: `[0,#]`. Its meaning has automatically changed because we updated the current position `#` from 1 to 2.
- Each edge consumes O(1) space, because it consists of only two pointers into the text, regardless of how many characters it represents.

Next we increment the position again and update the tree by appending a `c` to every existing edge and inserting one new edge for the new suffix `c`.

In our representation this looks like



And what it means is:



We observe:

- The tree is the correct suffix tree *up to the current position* after each step
- There are as many steps as there are characters in the text
- The amount of work in each step is O(1), because all existing edges are updated automatically by incrementing `#`, and inserting the one new edge for the final character can be done in O(1) time. Hence for a string of length n, only O(n) time is required.

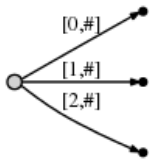
First extension: Simple repetitions

Of course this works so nicely only because our string does not contain any repetitions. We now look at a more realistic string:

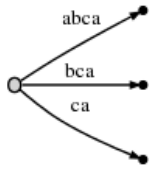
```
abcabxabcd
```

It starts with `abc` as in the previous example, then `ab` is repeated and followed by `x`, and then `abc` is repeated followed by `d`.

Steps 1 through 3: After the first 3 steps we have the tree from the previous example:



Step 4: We move `#` to position 4. This implicitly updates all existing edges to this:



and we need to insert the final suffix of the current step, `a`, at the root.

Before we do this, we introduce **two more variables** (in addition to `#`), which of course have been there all the time but we haven't used them so far:

- The **active point**, which is a triple `(active_node,active_edge,active_length)`
- The `remainder`, which is an integer indicating how many new suffixes we need to insert

The exact meaning of these two will become clear soon, but for now let's just say:

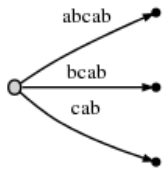
- In the simple `abc` example, the active point was always `(root,'\0x',0)`, i.e. `active_node` was the root node, `active_edge` was specified as the null character `'\0x'`, and `active_length` was zero. The effect of this was that the one new edge that we inserted in every step was inserted at the root node as a freshly created edge. We will see soon why a triple is necessary to represent this information.
- The `remainder` was always set to 1 at the beginning of each step. The meaning of this was that the number of suffixes we had to actively insert at the end of each step was 1 (always just the final character).

Now this is going to change. When we insert the current final character `a` at the root, we notice that there is already an outgoing edge starting with `a`, specifically: `abca`. Here is what we do in such a case:

- We do **not** insert a fresh edge at the root node.
- Instead we simply notice that the suffix `a` is already in our tree. It ends in the middle of a longer edge, but we are not bothered by that. We just leave things the way they are.
- We **set the active point** to `(root,'a',1)`. That means the active point is now somewhere in the middle of outgoing edge of the root node that starts with `a`, specifically, after position 1 on that edge. We notice that the edge is specified simply by its first character `a`. That suffices because there can be *only one* edge starting with any particular character (confirm that this is true after reading through the entire description).
- We also increment `remainder`, so at the beginning of the next step it will be 2.

Observation: When the final **suffix we need to insert is found to exist in the tree already**, the tree itself is **not changed** at all (we only update the active point and `remainder`). The tree is then not an accurate representation of the suffix tree *up to the current position* any more, but it **contains** all suffixes (because the final suffix `a` is contained *implicitly*). Hence, apart from updating the variables (which are all of fixed length, so this is O(1)), there was **no work** done in this step.

Step 5: We update the current position `#` to 5. This automatically updates the tree to this:



And **because `remainder` is 2**, we need to insert two final suffixes of the current position: `ab` and `b`. This is basically because:

- The `a` suffix from the previous step has never been properly inserted. So it has *remained*, and since we have progressed one step, it has now grown from `a` to `ab`.
- And we need to insert the new final edge `b`.

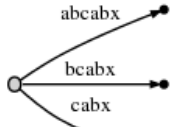
In practice this means that we go to the active point (which points to behind the `a` on what is now the `abcab` edge), and insert the current final character `b`. **But:** Again, it turns out that `b` is also already present on that same edge.

So, again, we do not change the tree. We simply:

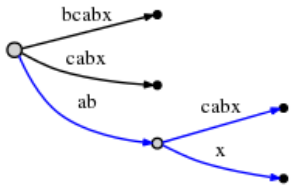
- Update the active point to `(root,'a',2)` (same node and edge as before, but now we point to behind the `b`)
- Increment the `remainder` to 3 because we still have not properly inserted the final edge from the previous step, and we don't insert the current final edge either.

To be clear: We had to insert `ab` and `b` in the current step, but because `ab` was already found, we updated the active point and did not even attempt to insert `b`. Why? Because if `ab` is in the tree, **every suffix** of it (including `b`) must be in the tree, too. Perhaps only *implicitly*, but it must be there, because of the way we have built the tree so far.

We proceed to **step 6** by incrementing `#`. The tree is automatically updated to:



Because **`remainder` is 3**, we have to insert `abx`, `bx` and `x`. The active point tells us where `ab` ends, so we only need to jump there and insert the `x`. Indeed, `x` is not there yet, so we split the `abcabx` edge and insert an internal node:



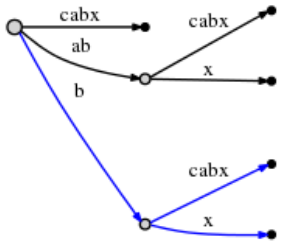
The edge representations are still pointers into the text, so splitting and inserting an internal node can be done in O(1) time.

So we have dealt with `abx` and decrement `remainder` to 2. Now we need to insert the next remaining suffix, `bx`. But before we do that we need to update the active point. The rule for this, after splitting and inserting an edge, will be called **Rule 1** below, and it applies whenever the `active_node` is root (we will learn rule 3 for other cases further below). Here is rule 1:

After an insertion from root,

- `active_node` remains root
- `active_edge` is set to the first character of the new suffix we need to insert, i.e. `b`
- `active_length` is reduced by 1

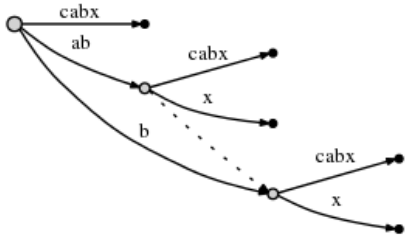
Hence, the new active-point triple `(root,'b',1)` indicates that the next insert has to be made at the `bcabx` edge, behind 1 character, i.e. behind `b`. We can identify the insertion point in O(1) time and check whether `x` is already present or not. If it was present, we would end the current step and leave everything the way it is. But `x` is not present, so we insert it by splitting the edge:



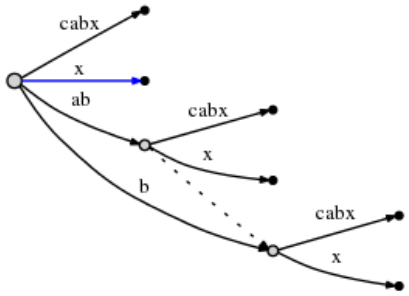
Again, this took O(1) time and we update `remainder` to 1 and the active point to `(root,'x',0)` as rule 1 states.

But there is one more thing we need to do. We'll call this **Rule 2:**

> If we split an edge and insert a new node, and if that is *not the first node* created during the current step, we connect the previously inserted node and the new node through a special pointer, a **suffix link**. We will later see why that is useful. Here is what we get, the suffix link is represented as a dotted edge:



We still need to insert the final suffix of the current step, `x`. Since the `active_length` component of the active node has fallen to 0, the final insert is made at the root directly. Since there is no outgoing edge at the root node starting with `x`, we insert a new edge:



As we can see, in the current step all remaining inserts were made.

We proceed to **step 7** by setting `#` =7, which automatically appends the next character, `a`, to all leaf edges, as always. Then we attempt to insert the new final character to the active point (the root), and find that it is there already. So we end the current step without inserting anything and update the active point to `(root,'a',1)`.

In **step 8**,, `#` =8, we append `b`, and as seen before, this only means we update the active point to `(root,'b',2)` and increment `remainder` without doing anything else, because `b` is already present. **However,** we notice (in O(1) time) that the active point is now at the end of an edge. We reflect this by re-setting it to `(node1,'\0x',0)`. Here, I use `node1` to refer to the internal node the `ab` edge ends at.

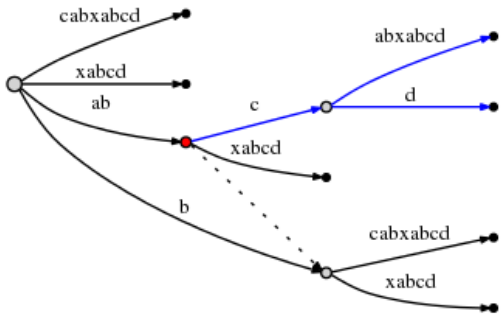
Then, in **step `#` =9**, we need to insert 'c' and this will help us to understand the final trick:

Second extension: Using suffix links

As always, the `#` update appends `c` automatically to the leaf edges and we go to the active point to see if we can insert 'c'. It turns out 'c' exists already at that edge, so we set the active point to `(node1,'c',1)`, increment `remainder` and do nothing else.

Now in **step `#` =10**, `remainder is 4`, and so we first need to insert `abcd` (which remains from 3 steps ago) by inserting `d` at the active point.

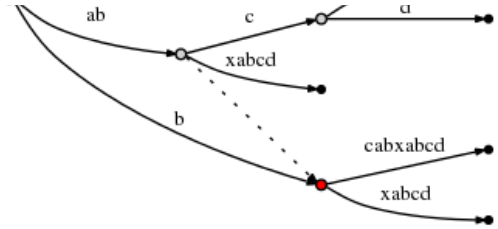
Attempting to insert `d` at the active point causes an edge split in O(1) time:



The `active_node`, from which the split was initiated, is marked in red above. Here is the final rule, **Rule 3:**

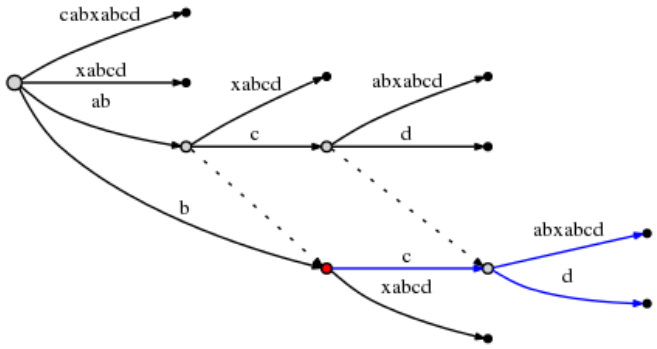
> After splitting an edge from an `active_node` that is not the root node, we follow the suffix link going out of that node, if there is any, and reset the `active_node` to the node it points to. If there is no suffix link, we set the `active_node` to the root. `active_edge` and `active_length` remain unchanged.

So the active point is now `(node2,'c',1)`, and `node2` is marked in red below:



Since the insertion of `abcd` is complete, we decrement `remainder` to 3 and consider the next remaining suffix of the current step, `bcd`. Rule 3 has set the active point to just the right node and edge so inserting `bcd` can be done by simply inserting its final character `d` at the active point.

Doing this causes another edge split, and **because of rule 2**, we must create a suffix link from the previously inserted node to the new one:

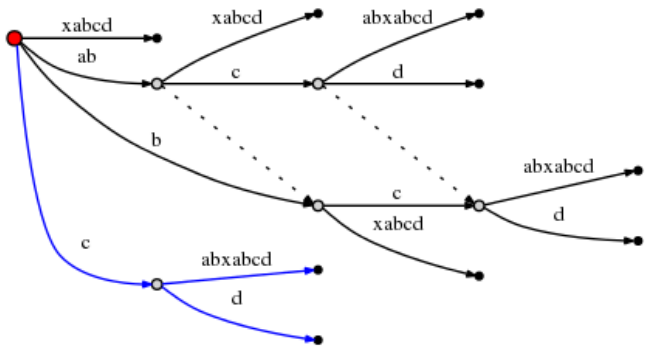


We observe: Suffix links enable us to reset the active point so we can make the next *remaining insert* at O(1) effort. Look at the graph above to confirm that indeed node at label `ab` is linked to the node at `b` (its suffix), and the node at `abc` is linked to `bc`.

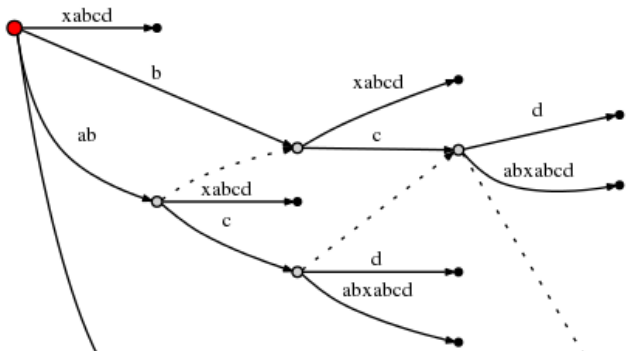
The current step is not finished yet. `remainder` is now 2, and we need to follow rule 3 to reset the active point again. Since the current `active_node` (red above) has no suffix link, we reset to root. The

active point is now `(root,'c',1)`.

Hence the next insert occurs at the one outgoing edge of the root node whose label starts with `c`: `cabxabcd`, behind the first character, i.e. behind `c`. This causes another split:

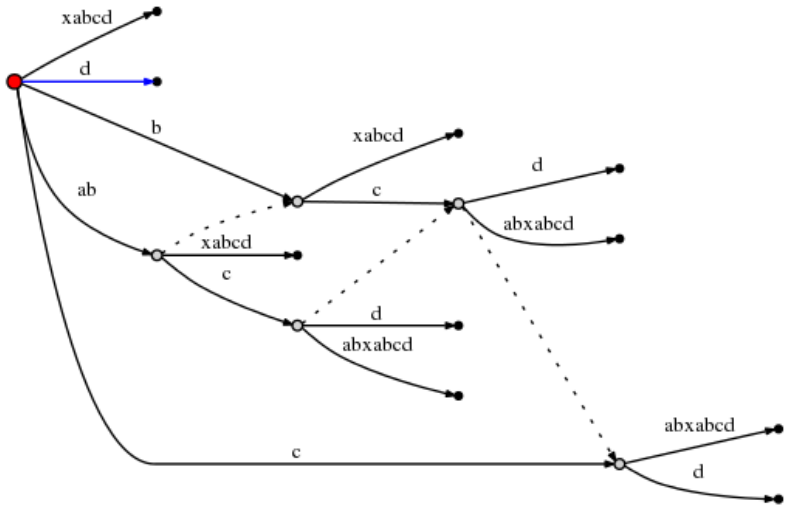


And since this involves the creation of a new internal node,we follow rule 2 and set a new suffix link from the previously created internal node:



(I am using Graphviz Dot for these little graphs. The new suffix link caused dot to re-arrange the existing edges, so check carefully to confirm that the only thing that was inserted above is a new suffix link.)

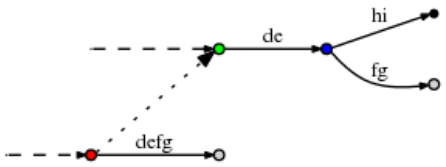
With this, `remainder` can be set to 1 and since the `active_node` is root, we use rule 1 to update the active point to `(root,'d',0)`. This means the final insert of the current step is to insert a single `d` at root:



That was the final step and we are done. There are number of **final observations**, though:

- In each step we move `#` forward by 1 position. This automatically updates all leaf nodes in O(1) time.
- But it does not deal with a) any suffixes *remaining* from previous steps, and b) with the one final character of the current step.
- `remainder` tells us how many additional inserts we need to make. These inserts correspond one-to-one to the final suffixes of the string that ends at the current position `#`. We consider one after the other and make the insert. **Important:** Each insert is done in O(1) time since the active point tells us exactly where to go, and we need to add only one single character at the active point. Why? Because the other characters are *contained implicitly* (otherwise the active point would not be where it is).
- After each such insert, we decrement `remainder` and follow the suffix link if there is any. If not we go to root (rule 3). If we are at root already, we modify the active point using rule 1. In any case, it takes only O(1) time.
- If, during one of these inserts, we find that the character we want to insert is already there, we don't do anything and end the current step, even if `remainder` >0. The reason is that any inserts that remain will be suffixes of the one we just tried to make. Hence they are all *implicit* in the current tree. The fact that `remainder` >0 makes sure we deal with the remaining suffixes later.
- What if at the end of the algorithm `remainder` >0? This will be the case whenever the end of the text is a substring that occurred somewhere before. In that case we must append one extra character at the end of the string that has not occurred before. In the literature, usually the dollar sign `$` is used as a symbol for that. **Why does that matter?** --> If later we use the completed suffix tree to search for suffixes, we must accept matches only if they *end at a leaf*. Otherwise we would get a lot of spurious matches, because there are *many* strings *implicitly* contained in the tree that are not actual suffixes of the main string. Forcing `remainder` to be 0 at the end is essentially a way to ensure that all suffixes end at a leaf node. **However,** if we want to use the tree to search for *general substrings*, not only *suffixes* of the main string, this final step is indeed not required, as suggested by the OP's comment below.

- So what is the complexity of the entire algorithm? If the text is n characters in length, there are obviously n steps (or n+1 if we add the dollar sign). In each step we either do nothing (other than updating the variables), or we make `remainder` inserts, each taking O(1) time. Since `remainder` indicates how many times we have done nothing in previous steps, and is decremented for every insert that we make now, the total number of times we do something is exactly n (or n+1). Hence, the total complexity is O(n).
- However, there is one small thing that I did not properly explain: It can happen that we follow a suffix link, update the active point, and then find that its `active_length` component does not work well with the new `active_node`. For example, consider a situation like this:



(The dashed lines indicate the rest of the tree. The dotted line is a suffix link.)

Now let the active point be `(red,'d',3)`, so it points to the place behind the `f` on the `defg` edge. Now assume we made the necessary updates and now follow the suffix link to update the active point according to rule 3. The new active point is `(green,'d',3)`. However, the `d`-edge going out of the green node is `de`, so it has only 2 characters. In order to find the correct active point, we obviously need to follow that edge to the blue node and reset to `(blue,'f',1)`.

In a particularly bad case, the `active_length` could be as large as `remainder`, which can be as large as n. And it might very well happen that to find the correct active point, we need not only jump over one internal node, but perhaps many, up to n in the worst case. Does that mean the algorithm has a hidden $O(n^2)$ complexity, because in each step `remainder` is generally O(n), and the post-adjustments to the active node after following a suffix link could be O(n), too?

No. The reason is that if indeed we have to adjust the active point (e.g. from green to blue as above), that brings us to a new node that has its own suffix link, and `active_length` will be reduced. As we follow down the chain of suffix links we make the remaining inserts, `active_length` can only decrease, and the number of active-point adjustments we can make on the way can't be larger than `active_length` at any given time. Since `active_length` can never be larger than `remainder`, and `remainder` is O(n) not only in every single step, but the total sum of increments ever made to `remainder` over the course of the entire process is O(n) too, the number of active point adjustments is also bounded by O(n).

edited **Apr 13 at 5:45**

answered **Mar 1 at 9:13**

jogojapan

9,431 2 14 32

15 Sorry this ended up a little longer than I hoped. And I realize it explains an number of trivial things we all know, while the difficult parts might still not be perfectly clear. Let's edit it into shape together. – jogojapan Mar 1 at 9:14

12 And I should add that this is *not* based on the description found in Dan Gusfield's book. It's a new attempt at describing the algorithm by first considering a string with no repetitions and then discussing how repetitions are handled. I hoped that would be more intuitive. – jogojapan Mar 1 at 9:16

70 Oh, to everyone else, vote this man up! Such answers deserve a significant number of votes! – Nathan Ridley Mar 1 at 14:42

13 This is why I love stack overflow. Epic answer. – Tom Anderson Apr 11 at 23:15

19 Outstanding answer. It helps that the question was actually interesting and well presented and hence would encourage a quality answer, but this is above and beyond. Would love to see more of this on SO. Kudos @jogojapan. – OJ. Apr 11 at 23:28

show **23** more comments

feedback

I have had quite a lot problems to implement this data structure myself. In the end I found this article and managed to implement it. A great plus for it is that you have a step-by-step example for a quite long string so you get to see what you should do. Please take a look at the article and I will be more then

happy to give tips where needed. I am hesitant to gove yet another full-blown explanation as there are alredy quite a few round the internet.

answered **Feb 26 at 11:59**

izomorphius

7,261 1 7 28

1 Yeah I've seen that one but again there's a lot of text and symbology that I am having trouble with. I think I need an illustrated picture book to understand it properly :P – Nathan Ridley Feb 26 at 12:03

There are tons of pictures in this article. Please be patient and try to read it. At first the author tries to explain why is the suffix tree good and then shows in great detail how to build it(pictures included on each step). Still there are a few things you can't evade - you need to know what is suffix and compressed trie at least. – izomorphius Feb 26 at 12:06

I intend to write my own blog post once I have my head wrapped around this, and I'm going to illustrate every aspect of the algorithm, step by step. – Nathan Ridley Feb 26 at 12:07

I would happily help you with that. Also should you have any question on a given part of the algorithm I will be glad to answer – izomorphius Feb 26 at 12:10

Ok I just re-read the article and the bit that lost me is when he started talking about "proper" suffixes without explaining what he meant by "proper". – Nathan Ridley Feb 26 at 12:12

show **1** more comment

feedback

My intuition is as follows:

After k iterations of the main loop you have constructed a suffix tree which contains all suffixes of the complete string that start in the first k characters.

At the start, this means the suffix tree contains a single root node that represents the entire string (this is the only suffix that starts at 0).

After len(string) iterations you have a suffix tree that contains all suffixes.

During the loop the key is the active point. My guess is that this represents the deepest point in the suffix tree that corresponds to a proper suffix of the first k characters of the string. (I think proper means that the suffix cannot be the entire string.)

For example, suppose you have seen characters 'abcabc'. The active point would represent the point in the tree corresponding to the suffix 'abc'.

The active point is represented by (origin,first,last). This means that you are currently at the point in the tree that you get to by starting at node origin and then feeding in the characters in string[first:last]

When you add a new character you look to see whether the active point is still in the existing tree. If it is then you are done. Otherwise you need to add a new node to the suffix tree at the active point, fallback to the next shortest match, and check again.

Note 1: The suffix pointers give a link to the next shortest match for each node.

Note 2: When you add a new node and fallback you add a new suffix pointer for the new node. The destination for this suffix pointer will be the node at the shortened active point. This node will either already exist, or be created on the next iteration of this fallback loop.

Note 3: The canonization part simply saves time in checking the active point. For example, suppose you always used origin=0, and just changed first and last. To check the active point you would have to follow the suffix tree each time along all the intermediate nodes. It makes sense to cache the result of following this path by recording just the distance from the last node.

Can you give a code example of what you mean by "fix" bounding variables?

Health warning: I also found this algorithm particularly hard to understand so please realise that this intuition is likely to be incorrect in all important details...

edited **Feb 26 at 20:41**

answered **Feb 26 at 20:16**

Peter de Rivaz

2,290 1 2 12

feedback

An implementation in java can be found at https://gist.github.com/3355993

answered **Aug 15 at 4:48**

bicepjai

37 10

feedback

---

protected by Mysticial Aug 15 at 4:56

This question is protected to prevent "thanks!", "me too!", or spam answers by new users. To answer it, you must have earned at least 10 reputation on this site.

Not the answer you're looking for? Browse other questions tagged algorithm search suffix-tree or ask your own question.