Fast String Searching With Suffix Trees

« A Floating Point Problem Data Compression with the Burrows-Wheeler Transform »

Posted in August 1st, 1996
by Mark Nelson in Computer Science, Data Compression, Magazine Articles

Published in **Dr. Dobb's Journal** August, 1996

I think that I shall never see

A poem lovely as a tree.

Poems are made by fools like me,

But only God can make a tree.

- **Joyce Kilmer**

A tree's a tree. How many more do you need to look at?

-**Ronald Reagan**

The problem

Matching string sequences is a problem that computer programmers face on a regular basis. Some programming tasks, such as data compression or DNA sequencing, can benefit enormously from improvements in string matching algorithms. This article discusses a relatively unknown data structure, the *suffix tree*, and shows how its characteristics can be used to attack difficult string matching problems.

Imagine that you've just been hired as a programmer working on a DNA sequencing project. Researchers are busy slicing and dicing viral genetic material, producing fragmented sequences of nucleotides. They send these sequences to your server, which is then expected to locate the sequences in a database of genomes. The genome for a given virus can have hundreds of thousands of nucleotide bases, and you have hundreds of viruses in your database. You are expected to implement this as a client/server project that gives real-time feedback to the impatient PhD.s. What's the best way to go about it?

It is obvious at this point that a brute force string search is going to be terribly inefficient. This type of search would require you to perform a string comparison at every single nucleotide in every genome in your database. Testing a long fragment that has a high hit rate of partial matches would make your client/server system look like an antique batch processing machine. Your challenge is to come up with an efficient string matching solution.

The intuitive solution

Since the database that you are testing against is invariant, preprocessing it to simplify the search seems like a good idea. One preprocessing approach is to build a search trie. For searching through input text, a straightforward approach to a search trie yields a thing called a *suffix trie*. (The suffix trie is just one step away from my final destination, the *suffix tree*.) A trie is a type of tree that has N possible branches from each node, where N is the number of characters in the alphabet. The word 'suffix' is used in this case to refer to the fact that the trie contains all of the suffixes of a given block of text (perhaps a viral genome.)

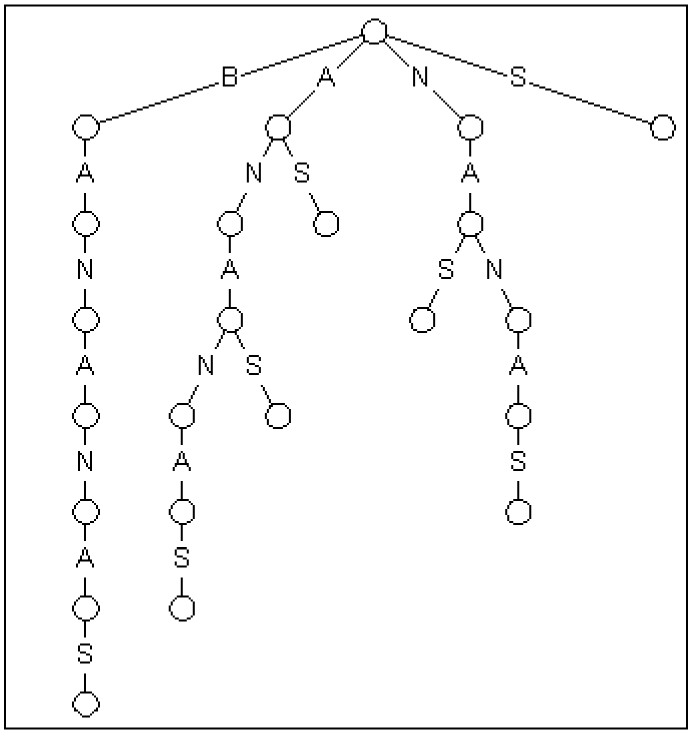


Figure 1

The Suffix Trie Representing "BANANAS"

Figure 1 shows a Suffix trie for the word BANANAS. There are two important facts to note about this trie. First, starting at the root node, each of the suffixes of BANANAS is found in the trie, starting with BANANAS, ANANAS, NANAS, and finishing up with a solitary S. Second, because of this organization, you can search for any substring of the word by starting at the root and following matches down the tree until exhausted.

The second point is what makes the suffix trie such a nice construct. If you have a input text of length *N*, and a search string of length *M*, a traiditonal brute force search will take as many as *N*M* character comparison to complete. Optimized searching techniques, such as the Boyer-Moore algorithm can guarantee searches that require no more than *M+N* comparisons, with even better average performance. But the suffix trie demolishes this performance by requiring just *M* character comparisons, regardless of the length of the text being searched!

Remarkable as this might seem, it means I could determine if the word BANANAS was in the collected works of William Shakespeare by performing just seven character comparisons. Of course, there is just one little catch: the time needed to construct the trie.

The reason you don't hear much about the use of suffix tries is the simple fact that constructing one requires $O(N^2)$ time and space. This quadratic performance rules out the use of suffix tries where they are needed most: to search through long blocks of data.

Under the spreading suffix tree

A reasonable way past this dilemma was proposed by Edward McCreight in 1976, when he published his paper on what came to be known as the *suffix tree*. The suffix tree for a given block of data retains the same topology as the suffix trie, but it eliminates nodes that have only a single descendant. This process, known as path compression, means that individual edges in the tree now may represent sequences of text instead of single characters.

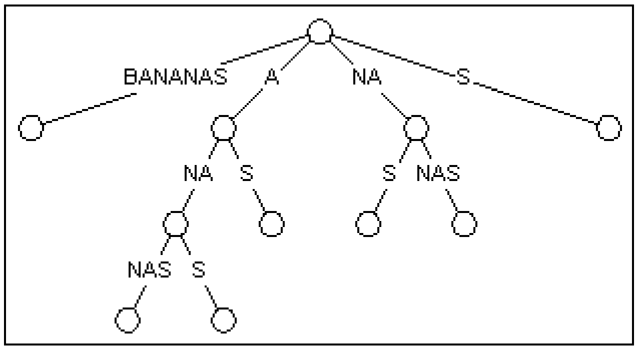


Figure 2

The Suffix Trie Representing "BANANAS"

Figure 2 shows what the suffix trie from Figure 1 looks like when converted to a suffix tree. You can see that the tree still has the same general shape, just far fewer nodes. By eliminating every node with just a single descendant, the count is reduced from 23 to 11.

In fact, the reduction in the number of nodes is such that the time and space requirements for constructing a suffix tree are reduced from O(N^2) to O(N). In the worst case, a suffix tree can be built with a maximum of 2N nodes, where N is the length of the input text. So for a one-time investment proportional to the length of the input text, we can create a tree that turbocharges our string searches.

Even you can make a tree

McCreight's original algorithm for constructing a suffix tree had a few disadvantages. Principle among them was the requirement that the tree be built in reverse order, meaning characters were added from the end of the input. This ruled the algorithm out for on line processing, making it much more difficult to use for applications such as data compression.

Twenty years later, Esko Ukkonen from the University of Helsinki came to the rescue with a slightly modified version of the algorithm that works from left to right. Both my sample code and the descriptions that follow are based on Ukkonen's work, published in the September 1995 issue of Algorithmica.

For a given string of text, T, Ukkonen's algorithm starts with an empty tree, then progressively adds each of the N prefixes of T to the suffix tree. For example, when creating the suffix tree for BANANAS, B is inserted into the tree, then BA, then BAN, and so on. When BANANAS is finally inserted, the tree is complete.

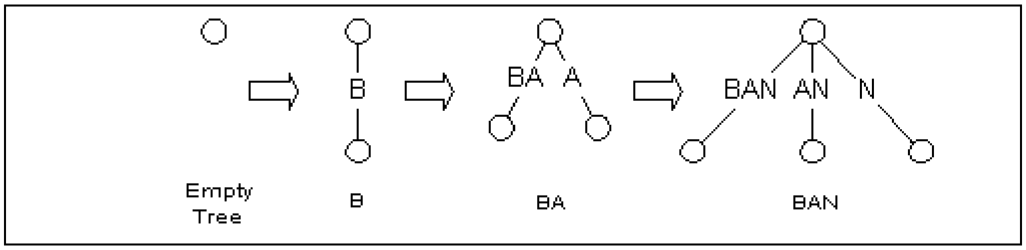


Figure 3

Progressively Building the Suffix Tree

Suffix tree mechanics

Adding a new prefix to the tree is done by walking through the tree and visiting each of the suffixes of the current tree. We start at the longest suffix (BAN in Figure 3), and work our way down to the shortest suffix, which is the empty string. Each suffix ends at a node that consists of one of these three types:

- A leaf node. In Figure 4, the nodes labeled 1,2, 4, and 5 are leaf nodes.
- An explicit node. The non-leaf nodes that are labeled 0 and 3 in Figure 4 are explicit nodes. They represent a point on the tree where two or more edges part ways.
- An implicit node. In Figure 4, prefixes such as BO, BOO, and OO all end in the middle of an edge. These positions are referred to as *implicit* nodes. They would represent nodes in the suffix trie, but path compression eliminated them. As the tree is built, implicit nodes are sometimes converted to explicit nodes.

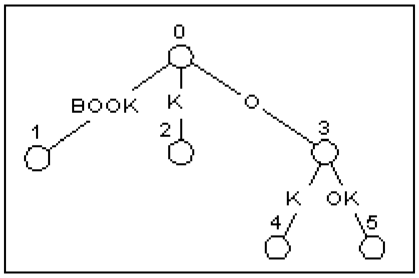


Figure 4
BOOKKEEPER after adding BOOK

In Figure 4, there are five suffixes in the tree (including the empty string) after adding BOOK to the structure. Adding the next prefix, BOOKK to the tree means visiting each of the suffixes in the existing tree, and adding letter K to the end of the suffix.

The first four suffixes, BOOK, OOK, OK, and K, all end at leaf nodes. Because of the path compression applied to suffix trees, adding a new character to a leaf node will always just add to the string on that node. It will never create a new node, regardless of the letter being added.

After all of the leaf nodes have been updated, we still need to add character 'K' to the empty string, which is found at node 0. Since there is already an edge leaving node 0 that starts with letter K, we don't have to do anything. The newly added suffix K will be found at node 0, and will end at the implicit node found one character down along the edge leading to node 2.

The final shape of the resulting tree is shown in Figure 5.

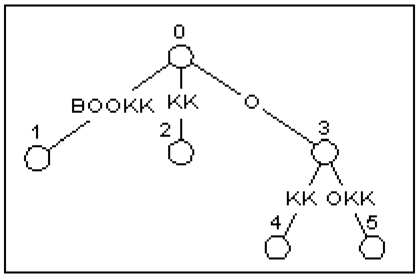


Figure 5

The same tree after adding BOOKK

Things get knotty

Updating the tree in Figure 4 was relatively easy. We performed two types of updates: the first was simply the extension of an edge, and the second was an implicit update, which involved no work at all. Adding BOOKKE to the tree shown in Figure 5 will demonstrate the two other types of updates. In the first type, a new node is created to split an existing edge at an implicit node, followed by the addition of a new edge. The second type of update consists of adding a new edge to an explicit node.

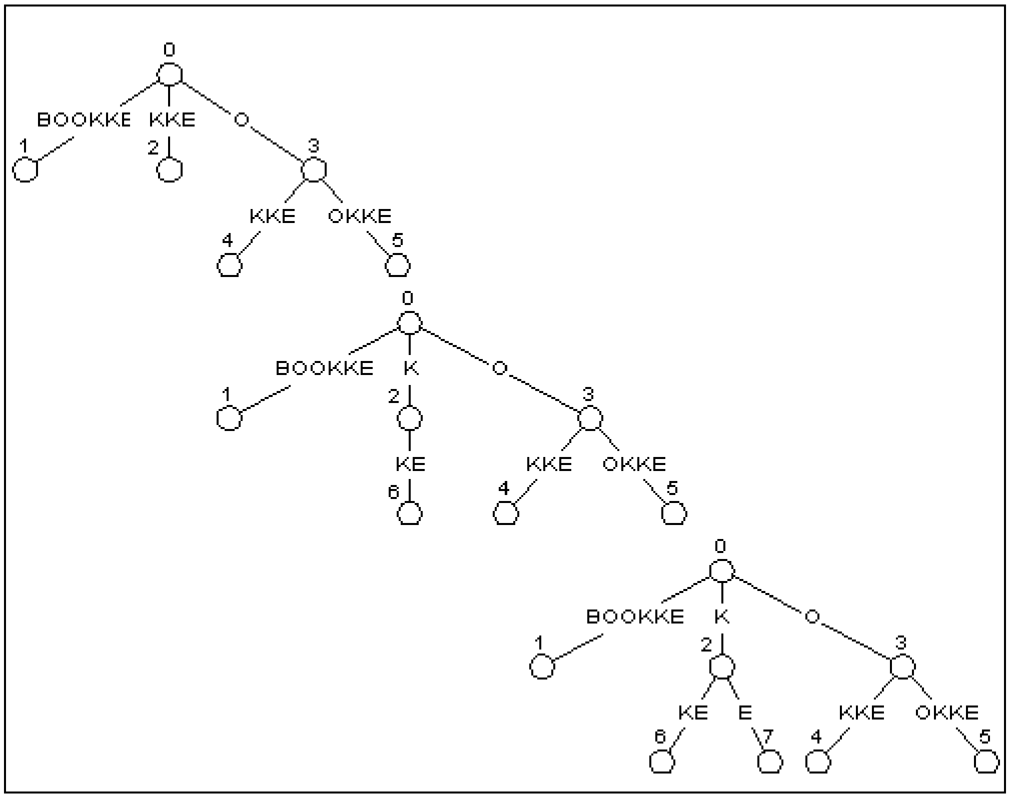


Figure 6

The Split and Add Update

When adding BOOKKE to the tree in Figure 5, we once again start with the longest suffix, BOOKK, and work our way to the shortest, the empty string. Updating the longer suffixes is trivial as long as we are updating leaf nodes. In Figure 5, the suffixes that end in leaf nodes are BOOKK, OOKK, OKK, and KK. The first tree in Figure 6 shows what the tree looks like after these suffixes have been updated using the simple string extension.

The first suffix in Figure 5 that doesn't terminate at a leaf node is K. When updating a suffix tree, the first non-leaf node is defined as the active point of the tree. All of the suffixes that are longer than the suffix defined by the active point will end in leaf nodes. None of the suffixes after this point will terminate in leaf nodes.

The suffix K terminates in an implicit node part way down the edge defined by KKE. When testing non-leaf nodes, we need to see if they have any descendants that match the new character being appended. In this case, that would be E.

A quick look at the first K in KKE shows that it only has a single descendant: K. So this means we have to add a descendent to represent Letter E. This is a two step process. First, we split the edge holding the arc so that it has an explicit node at the end of the suffix being tested. The middle tree in Figure 6 shows what the tree looks like after the split.

Once the edge has been split, and the new node has been added, you have a tree that looks like that in the third position of Figure 6. Note that the K node, which has now grown to be KE, has become a leaf node.

Updating an explicit node

After updating suffix K, we still have to update the next shorter suffix, which is the empty string. The empty string ends at explicit node 0, so we just have to check to see if it has a descendant that starts with letter E. A quick look at the tree in Figure 6 shows that node 0 doesn't have a descendant, so another leaf node is added, which yields the tree shown in Figure 7.

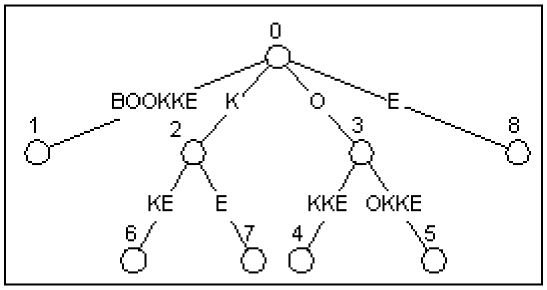


Figure 7

Generalizing the algorithm

By taking advantage of a few of the characteristics of the suffix tree, we can generate a fairly efficient algorithm. The first important trait is this: once a leaf node, always a leaf node. Any node that we create as a leaf will never be given a descendant, it will only be extended through character concatenation. More importantly, every time we add a new suffix to the tree, we are going to automatically extend the edges leading into every leaf node by a single character. That character will be the last character in the new suffix.

This makes management of the edges leading into leaf nodes easy. Any time we create a new leaf node, we automatically set its edge to represent all the characters from its starting point to the end of the input text. Even if we don't know what those characters are, we know they will be added to the tree eventually. Because of this, once a leaf node is created, we can just forget about it! If the edge is split, its starting point may change, but it will still extend all the way to the end of the input text.

This means that we only have to worry about updating explicit and implicit nodes at the active point, which was the first non-leaf node. Given this, we would have to progress from the active point to the empty string, testing each node for update eligibility.

However, we can save some time by stopping our update earlier. As we walk through the suffixes, we will add a new edge to each node that doesn't have a descendant edge starting with the correct character. When we finally do reach a node that has the correct character as a descendant, we can simply stop updating. Knowing how the construction algorithm works, you can see that if you find a certain character as a descendant of a particular suffix, you are bound to also find it as a descendant of every smaller suffix.

The point where you find the first matching descendant is called the end point. The end point has an additional feature that makes it particularly useful. Since we were adding leaves to every suffix between the active point and the end point, we now know that every suffix longer than the end point is a leaf node. This means the end point will turn into the active point on the next pass over the tree!

By confining our updates to the suffixes between the active point and the end point, we cut way back on the processing required to update the tree. And by keeping track of the end point, we automatically know what the active point will be on the next pass. A first pass at the update algorithm using this information might look something like this (in C-like pseudo code) :

PLAIN TEXT

C:

```
Update( new_suffix )
{
  current_suffix = active_point
  test_char = last_char in new_suffix
  done = false;
  while ( !done ) {
    if current_suffix ends at an explicit node {
      if the node has no descendant edge starting with test_char
        create new leaf edge starting at the explicit node
      else
        done = true;
    } else {
      if the implicit node's next char isn't test_char {
        split the edge at the implicit node
        create new leaf edge starting at the split in the edge
      } else
        done = true;
    }
    if current_suffix is the empty string
      done = true;
    else
       current_suffix = next_smaller_suffix( current_suffix )
  }
  active_point = current_suffix
}
```

The Suffix Pointer

The pseudo-code algorithm shown above is more or less accurate, but it glosses over one difficulty. As we are navigating through the tree, we move to the next smaller suffix via a call to `next_smaller_suffix()`. This routine has to find the implicit or explicit node corresponding to a particular suffix.

If we do this by simply walking down the tree until we find the correct node, our algorithm isn't going to run in linear time. To get around this, we have to add one additional pointer to the tree: the *suffix pointer*. The suffix pointer is a pointer found at each internal node. Each internal node represents a sequence of characters that start at the root. The suffix pointer points to the node that is the first suffix of that string. So if a particular string contains characters 0 through N of the input text, the suffix pointer for that string will point to the node that is the termination point for the string starting at the root that represents characters 1 through N of the input text.

Figure 8 shows the suffix tree for the string ABABABC. The first suffix pointer is found at the node that represents ABAB. The first suffix of that string would be BAB, and that is where the suffix pointer at ABAB points. Likewise, BAB has its own suffix pointer, which points to the node for AB.

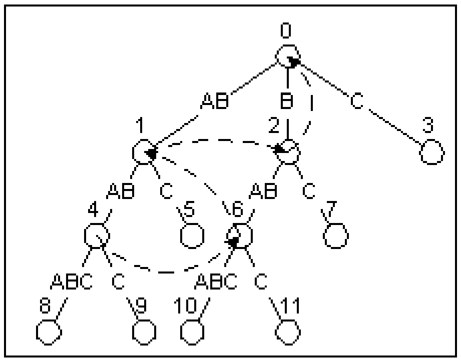


Figure 7

The suffix tree for ABABABC with suffix pointers shown as dashed lines

The suffix pointers are built at the same time the update to the tree is taking place. As I move from the active point to the end point, I keep track of the parent node of each of the new leaves I create. Each time I create a new edge, I also create a suffix pointer from the parent node of the *last* leaf edge I created to the *current* parent edge. (Obviously, I can't do this for the first edge created in the update, but I do for all the remaining edges.)

With the suffix pointers in place, navigating from one suffix to the next is simply a matter of following a pointer. This critical addition to the algorithm is what reduces it to an O(N) algorithm.

Tree houses

To help illustrate this article, I wrote a short program, STREE.CPP, that reads in a string of text from standard input and builds a suffix tree using fully documented C++. A second version, STREED.CPP, has extensive debug output as well. Links to both are available at the bottom of this article.

Understanding STREE.CPP is really just a matter of understanding the workings of the data structures that it contains. The most important data structure is the `Edge` object. The class definition for `Edge` is:

PLAIN TEXT

C++:

```
class Edge {
    public :
        int first_char_index;
        int last_char_index;
        int end_node;
        int start_node;
        void Insert();
        void Remove();
        Edge();
        Edge( int init_first_char_index,
              int init_last_char_index,
              int parent_node );
        int SplitEdge( Suffix &s );
        static Edge Find( int node, int c );
        static int Hash( int node, int c );
};
```

Each time a new edge in the suffix tree is created, a new Edge object is created to represent it. The four data members of the object are defined as follows:

first_char_index, **last_char_index**:

Each of the edges in the tree has a sequence of characters from the input text associated with it. To ensure that the storage size of each edge is identical, we just store two indices into the input text to represent the sequence.

start_node:

The number of the node that represents the starting node for this edge. Node 0 is the root of the tree.

end_node:

The number of the node that represents the end node for this edge. Each time an edge is created, a new end node is created as well. The end node for every edge will not change over the life of the tree, so this can be used as an edge id as well.

One of the most frequent tasks performed when building the suffix tree is to search for the edge emanating from a particular node based on the first character in its sequence. On a byte oriented computer, there could be as many as 256 edges originating at a single node. To make the search reasonably quick and easy, I store the edges in a hash table, using a hash key based on their starting node number and the first character of their substring. The `Insert()` and `Remove()` member functions are used to manage the transfer of edges in and out of the hash table.

The second important data structure used when building the suffix tree is the `Suffix` object. Remember that updating the tree is done by working through all of the suffixes of the string currently stored in the tree, starting with the longest, and ending at the end point. A `Suffix` is simply a sequence of characters that starts at node 0 and ends at some point in the tree.

It makes sense that we can then safely represent any suffix by defining just the position in the tree of its last character, since we know the first character starts at node 0, the root. The `Suffix` object, whose definition is shown here, defines a given suffix using that system:

PLAIN TEXT

C++:

```
class Suffix {
    public :
        int origin_node;
        int first_char_index;
        int last_char_index;
        Suffix( int node, int start, int stop );
        int Explicit();
        int Implicit();
        void Canonize();
};
```

The `Suffix` object defines the last character in a string by starting at a specific node, then following the string of characters in the input sequence pointed to by the first_char_index and last_char_index members. For example, in Figure 8, the longest suffix "ABABABC" would have an origin_node of 0, a first_char_index of 0, and a last_char_index of 6.

Ukkonen's algorithm requires that we work with these `Suffix` definitions in *canonical* form.
The `Canonize()` function is called to perform this transformation any time a `Suffix` object is modified. The canonical representation of the suffix simply requires that the origin_node in the `Suffix` object be the closest parent to the end point of the string. This means that the suffix string represented by the pair (0, "ABABABC"), would be canonized by moving first to (1, "ABABC"), then (4, "ABC"), and finally (8,"").

When a suffix string ends on an explicit node, the canonical representation will use an empty string to define the remaining characters in the string. An empty string is defined by setting first_char_index to be greater than last_char_index. When this is the case, we know that the suffix ends on an *explicit* node. If first_char_index is less than or equal to last_char_index, it means that the suffix string ends on an *implicit* node.

Given these data structure definitions, I think you will find the code in STREE.CPP to be a straightforward implementation of the Ukkonen algorithm. For additional clarity, use STREED.CPP to dump copious debug information out at runtime.

Acknowledgments

I was finally convinced to tackle suffix tree construction by reading Jesper Larsson's paper for the 1996 IEEE Data Compression Conference. Jesper was also kind enough to provide me with sample code and pointers to Ukkonen's paper.

References

E.M. McCreight. **A space-economical suffix tree construction algorithm**. Journal of the ACM, 23:262-272, 1976.

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

Source Code

Good news - this source code has been updated. It was originally published in 1996, pre-standard, and needed just a few nips and tucks to work properly in today's world. These new versions of the code should be pretty portable - the build properly with g++ 3.x, 4.x and Visual C++ 2003.

stree2006.cpp
    A simple program that builds a suffix tree from an input string.
streed2006.cpp
    The same program with much debugging code added.

The original code is her for the curious, but should not be used:

stree.cpp
    A simple program that builds a suffix tree from an input string.
streed.cpp
    The same program with much debugging code added.

165 users commented in " Fast String Searching With Suffix Trees "

Follow-up comment rss or Leave a Trackback

on December 5th, 2006 at 3:01 am, Martin said:

Nice paper but there is room for improvement.

1) Do use the same example throughout the text instead of using different more or less suited examples to illustrate different points - thus making it hard to follow what is going on. MISSISSIPPI would make a excellent example string.

2) Figure 4 already contains an explicit update which is not explained before carrying on - that is rather confusing.

on December 5th, 2006 at 6:19 am, Mark said:

Sigh, everyone's a critic.

I don't know that this article will ever see a revision, but if it does, I shall keep your comments in mind.

on December 13th, 2006 at 5:45 pm, Calin Culianu said:

Your sample code fails to compile cleanly on newer compilers. I guess that's because it was written back in 1996. At any rate thanks for taking the time to explain this. After reading your article I still don't really understand it -- and I am not sure if that's because I am too stupid your your article isn't clearly enough writte. *Shrugs*

on December 13th, 2006 at 8:16 pm, Mark said:

Hi Calin,

Let me know what compiler you are using, I might take a try at updating the source.

The big problem was that in 1996 there weren't any compilers that came anywhere near conforming to today's standard, which wasn't ratified until 1998.

As for understanding it, I agree that it's hard - I wrote the article because I had such a hard time with it myself. I encourage you to try to work through some of the problems by hand, then see if you can duplicate the results with the debug version of the program and see if you get the same results.

on December 26th, 2006 at 3:00 pm, Mark said:

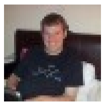
Updated source code released, see the end of the article with the links!

on February 20th, 2007 at 2:21 pm, Samuel said:

I'd like to say thank you.
This article and the source-code is definetly a big help in my diploma thesis where I need a suffix tree as a tool.

on February 20th, 2007 at 3:27 pm, Mark said:

Thanks, Samuel, nice to hear.



on February 21st, 2007 at 3:32 am, Graham Reeds said:

I like your code - currently implementing the BWT algorithm with just RLE as a preprocessor for XML transfer.

A couple of points:
* Make a link to this article from the BWT one - you mention suffix trees help speed up BWT compilation, but I had to go searching and found this site from Wikipedia.
* Lots of comments in the source - you tried Doxygen which would make the comments more searchable.

on February 23rd, 2007 at 4:55 pm, tony said:

hey mark, how can i use your code to tell me the index of a substring in a text?

on February 25th, 2007 at 12:22 pm, Mark said:

>hey mark, how can i use your code to tell me the index of a substring in a text?

Tony, that's kind of the whole point of the article, right?

Use the code in walk_tree as an indication of how to navigate the tree.

Once you walk the tree, searching for a match to your text, you will end up at either a leaf or an interior node.

If you are at leaf, you have found the only occurrence of the string, and can determine where it is by looking at the members of Suffix.

If you are at an interior node, you have found the root of a tree that will provide all the locations of the string in the text.

Good exercise to write this search routine!

on February 27th, 2007 at 6:36 am, Mark said:

Hi Graham,

Believe it or not, when I wrote this article, Doxygen didn't handle C code very well - mostly because it didn't exist yet!

I don't spend too much time going back and fixing up old articles, mostly because of my belief in the inspiration words of Samuel Johnson:

"No man but a blockhead ever wrote, except for money."

on March 2nd, 2007 at 12:28 pm, Shivam said:

How can I find the position(s) of the search string ?? What if there are wildcharacters in the pattern, can you recommend some references ?? Thanks

on March 2nd, 2007 at 1:23 pm, Mark said:

Shivam, do you realize that string searching occupies at least one chapter in virtually every algorithms book?

A search on Google for "string search algorithms" gets you 3.5 million hits.

If you can't find some decent references on this, you aren't trying very hard.

Sorry to be harsh, but seriously, I can help you with specific problems related to the data compression on my site, but general questions that can be easily answered, well, you need to manage yourself.

on March 15th, 2007 at 11:02 pm, cariaso said:

Mark, while I agree with your sentiments for Shivam, I can't resist pointing out:

"string searching occupies at least one chapter in every single algorithms book is dedicated to string searching"

If a book is dedicated to string searching, it seems a bit redundant to say that string searching occupies at least one chapter.

This article was tremendously valuable a decade ago. Since then there have been advances, especially coming from bioinformatics. Anyone interested in the topic should investigate:

An updated C implementation
http://www.icir.org/christian/libstree/

general notes
http://homepage.usask.ca/~ctl271/857/suffix_tree.shtml

big strings in small memory
http://csdl2.computer.org/persagen/DLAbsToc.jsp?
resourcePath=/dl/trans/tk/&toc=comp/trans/tk/2005/01/k1toc.xml&DOI=10.1109/TKDE.2005.3

a pairwise bioinformatics related tool
http://mummer.sourceforge.net/

a generalized bioinformatics tool
http://bibiserv.techfak.uni-bielefeld.de/mga/

on March 17th, 2007 at 1:11 pm, Mark said:

Thanks for the excellent links and comments Michael! As for the proofreading, well, that's what comes of having no editor. i think I will violate the integrity of my blog comments by correcting it without leaving a trail.

on March 28th, 2007 at 1:40 am, eNG-sIONG said:

Dear All,
I found this, may be useful, its a Standard Template Library for Extra Large Data Sets implemented by Roman Dementiev that support a multiple type of container.
Theres no suffix tree yet. :(

http://stxxl.sourceforge.net/
Eng Siong

on April 11th, 2007 at 5:31 pm, Diamante said:

Cool

on July 2nd, 2007 at 4:19 am, Ha Luong said:

Hi Mark,
I have read your article and I couldn't understand the "active point". The active point is the first non-leaf ? (And I don't know the first non-leaf) . Could you please show me what the active point is in your figure 6 ?
Thanks so much ,
Ha Luong

on July 2nd, 2007 at 6:55 am, Mark said:

Ha Luong,

I think my wording in that paragraph is not as good as it could be.

When I refer to the "first suffix that doesn't terminate in a leaf node", I am talking about the first suffix when considering the list of suffices:

BOOKK
OOKK
OKK
KK
K

Which is the list of suffixes you are now attempting to insert into the tree. Looking at Figure 5, you can see the suffix "K" terminates in the middle of the "KK" leaf node, which makes it the active point.

If you don't know what a non-leaf node is, you need to brush up on your data structures.

on July 12th, 2007 at 6:03 pm, David Hou said:

Hi Mark, thank you for the excellent paper. It saves me from Esko Ukkonen's(that one is sooo hard, German folks are too smart ....).

I still don't quite understand how the "end point" works, which is one crucial part of the whole thing. Could you kindly provide a sample to explain it more detail? thanks a lot.

on July 15th, 2007 at 7:31 am, Mark said:

When performing the insertion of a new string, the two most important points are the active point and the end point.

The active point is discussed in the response above. When inserting a string into the tree, we look at the list of suffixes we are inserting. Some of the suffixes will be added to the tree by simple addition of a new character on an existing leaf node. The first suffix that can't be added by this simple extension is used to mark the active point.

I'll go back to the process of adding BOOKKE to Figure 5.

First, BOOKKE is added through simple extension of leaf node 1.
OOKKE is added through simple extension of leaf node 5.
OKKE is added through simple extension of leaf node 4.
KKE is added through extension of leaf node 2.

At this point, the tree looks like the first part of figure 6. You can see that when we attempt to add "KE" to the tree, things are not so simple. There is no leaf node that terminates in "K", so we can't just extend that leaf.

The suffix "K" is part of the tree, but it is an implicit node that is in the middle of the "KKE" branch terminating at node 2. (Part 1 of figure 6). The active point of the tree is now that implicit "K" node on the node 2 branch. (And that is where we will perform the split operation.)

Once you enter into the split operation at the active node, you can walk through the pseudocode to see what the elusive endpoint is:

```
if the implicit node's next char isn't test_char {
split the edge at the implicit node
create new leaf edge starting at the split in the edge
} else
done = true;
```

The end point is the first node where you don't have to perform a split as you add new suffixes.

Unfortunately I didn't include a good example showing this, so I need to add that to my to-do list.

on July 18th, 2007 at 11:18 am, David Hou said:

Thank you Mark, i feel much clear now and I am trying to implement it in C#, oh... it's hard...

also, I found another tuition introduces Ukkonen's suffix tree and there's a demo is provide here: http://www.csse.monash.edu.au/~lloyd/tildeAlgDS/Tree/Suffix/#demoForm , the interesting thing is, when I run same string on your code and their code, I got different result... ... and idea about it ?

on July 22nd, 2007 at 2:13 pm, Mark said:

Well, if you could show me a short demo string that comes up with different results for the two programs, I'd certainly be interested in seeing it!

on July 23rd, 2007 at 7:30 pm, arao said:

the book by Prof. Dan Gusfield is a very good one on this.

and Esko Ukkonen is a German?? I thought he was from Finland...

on July 30th, 2007 at 3:19 pm, Abhey Shah said:

On the Mac I found I needed to recast T[i] into chars before printing, or it would just give me numbers.

on July 30th, 2007 at 5:15 pm, Mark said:

That seems kind of strange - T is typed as char. Casting at as a char should have no effect. Perhaps your compiler is using unsigned chars as default?

on August 1st, 2007 at 9:45 am, Abhey Shah said:

I tried gcc3.3 on linux as well as gcc4.0 on the mac, same thing, that was just doing g -lstdc streed2006.cpp.
Anyway, if anybody else runs into the same thing sed 's/streed2007.cpp

on August 1st, 2007 at 9:48 am, Abhey Shah said:

and of course the command lines get f***ed: got to love web 2.0. anyway I'm sure if anybody runs into the same thing they'll be able to piece together the edit commands.

on August 27th, 2007 at 5:07 am, Amit P said:

i have been looking at the updated code and need to adapt it to take many different input strings and not just the one, i tried creating a loop for active and having each loop hold a different string however this has still not worked any ideas would be appreciated.

thanks

on August 28th, 2007 at 11:18 am, Amit P said:

i added another line of input for T after addprefix loop, after the add input had the program recalculate N and added the for loop underneath, i will now try to write code that will input multiple strings from a loop. this did allow me to enter multiple strings into the same suffix tree.

on September 4th, 2007 at 7:13 am, Adam said:

I see that IBM's Many-eyes data visualization community just put up a new visualization type called a Word tree (visual suffix tree).

http://services.alphaworks.ibm.com/manyeyes/page/Word_Tree.html

on September 4th, 2007 at 7:49 am, Mark said:

That looks like a very nice visualization tool, it gives a nice understanding of how a suffix tree actually works.

on September 17th, 2007 at 4:01 pm, Scott said:

>> On the Mac I found I needed to recast
>> T[i] into chars before printing, or it
>> would just give me numbers.

>
> That seems kind of strange - T is typed
> as char. Casting at as a char should
> have no effect. Perhaps your compiler is
> using unsigned chars as default?

I noticed the same issue with Visual C++ 2005. I believe the 'problem' was that overload resolution was finding the operator int() cast method in the Aux class, and was using that to cast to int and thus calling operator

on September 17th, 2007 at 4:03 pm, Scott said:

Yuck, my post got mangled due to embedded less-than characters. Anyway, the overloaded int cast in the Aux class was preventing the Aux output method from being called.

on October 12th, 2007 at 2:33 am, till said:

```
Enter string: aa
Start End Suf First Last String
0 1 -1 0 1 aa
Would you like to validate the tree?y
Suffix : aa
comparing: aa to aa
Suffix 0 count wrong!
Leaf count : 1 Error!
Branch count : 1 OK
```

on October 12th, 2007 at 9:19 am, Mark said:

@till:

I suspect this is a corner case for n=2. You agree?

on November 7th, 2007 at 2:19 am, Rock Linker said:

hi mark

appreciate for your sharing! Thank you so much!

btw: in your article, there is an conception "end point", in your former response you also explained that, however, I still have some questions about that. According to your explaination, "end point" is the first node that need not to be splited, but you know, end point is shorter than the active point, and active point must be splited, how can an "end point" node exist in the suffix tree? I am a little confused. Thank you! wait for your kindly response. Thanks a lot!

on November 7th, 2007 at 12:21 pm, Mark said:

If we order the suffixes of a string in decreasing length:

Rock
ock
ck
k

The active point will be the first of these suffixes which does not terminate on a leaf node.

And like I said in the article, every suffix longer than the suffix at the active point terminates in a leaf. Every suffix short than the suffix at the active point does not.

As you are updating the suffix tree, you know you have reached the end point when an element in the suffix tree has the same descendant as the string you need to add at that point.

I don't know that there is a relationship between the end point and the active point. You seem to think there is some relationship between the end point and the active point, and I don't think that is true.

I suggest that in order to understand this, you just run through a few examples of moderate length to watch the active point, the end point, and the tree change in response to the addition of new suffixes.

on November 15th, 2007 at 1:53 am, Bryan said:

I still don´t understand the linear time thing.
Every time we add a new character we need to go through all the suffix pointers. That means that the time spent in every step depemds on how many suffix pointers are. And the number of suffix pointers is variable.
If I am wrong, please explain me, because I don´t undertand.

on November 15th, 2007 at 7:07 am, Mark said:

@Bryan:

Hi Bryan,

The place where you are off base is here:

>Every time we add a new character we need to
>go through all the suffix pointers.

If you look at either the pseudocode or the actual code, you'll see that we don't go through all the pointers, and that is what makes the algorithm efficient.

When it comes time to insert a new character, we start work at the active point:

PLAIN TEXT

C++:

```
Update( new_suffix )
{
  current_suffix = active_point
  test_char = last_char in new_suffix
  done = false;
  while ( !done ) {
    if current_suffix ends at an explicit node {
```

Because we keep track of the active point, we know in advance where the next search is going to start, so we most definitely do not search through all the suffix pointers in order to insert a new suffix.

Try working through some insertions by hand and you'll see how it works more clearly.

on November 21st, 2007 at 2:03 am, Bryan said:

Hi Mark:

Thank you for your soon reply. I´m going to try the insertions by hand. My doubt was more focused on how many times the while will be executed in the worst case. I´m going to check the article again and I hope I´ll get a better understanding of the algorithm.

Thank you for taking the time to read and reply.

on November 26th, 2007 at 6:47 pm, Bryan said:

Hi Mark:

I´ve seen the light. At the end, the while has been executed as many times as branches are in the tree. And since the amount of branches is equal to the amount of leafs, the time remains linear.

Thank you for your time.

Bytes

on November 29th, 2007 at 11:03 am, Tim Rowe said:

Many thanks for the clearest explanation of the algorithm that I've managed to find so far! I'm still struggling a little to understand it, but I think that's because it's genuinely tricky (or I'm genuinely thick) rather than any problems with the explanation. Off to try some examples on paper.

on November 30th, 2007 at 8:43 am, Tim Rowe said:

What is the licensing situation on that code? Are we free to use it with acknowledgement in the code? Acknowledgement visible to the user? Apply to you for a licence?

on December 1st, 2007 at 10:33 am, Mark said:

@Tim:

See the link at the top of the page: Liberal Code Use Policy.

But remember, this is demonstration code - you may be able to find something substantially more optimized elsewhere.

on December 14th, 2007 at 11:43 am, hlbnet said:

I do not see how a suffix tree can be build in linear time (so you understand I'm not an expert).
I KNOW it is true (read everywhere), but I definitively don't understand how it is possible.
If I follow your explanations, to construct our suffix tree, we have to browse all the characters of the text (here is the O(n) I think) and for each, we have to append the new character to all suffixes that are already in our tree. Here, I see a loop on the already registered suffixes nested in the main loop on the characters. And the inner loop becomes bigger each time a new suffix is added.
Is there something magic somewhere that makes the whole algorithm finally linear ?

on December 14th, 2007 at 11:54 am, Mark said:

@hlbnet:

The reason the algorithm runs in linear time is that appending a new suffix to the tree can be done in constant time. The reason the suffix can be appended in constant time is because we keep track of the active point.

If you look at the final version of the simplifed algorithm, you'll see that there is an outer loop that iterates over all the characters in the input sequence. But there's no inner loop. So the number of operations needed to create the whole tree is N*something, where something has an upper bound that is not dependent on N.

on December 17th, 2007 at 10:11 am, hlbnet said:

I'm not fully convinced just reading the algo, since I see a

PLAIN TEXT

C:

```
while ( !done )
```

in the

PLAIN TEXT

C:

```
Udpdate( newSuffix )
```

function (here is the inner loop).
I see that there are several stop conditions in the loop, making it stop rather "quickly". But it is still very hard for me to figure out the average number of loops that will be performed in a real example.
I definitively need to think of it more in depth.
Thank you for your article and answers, it is very usefull for beginners like me !

on February 14th, 2008 at 3:07 am, lena said:

Hi Mark..
Thanks on your article..a very big help for my thesis..;)

on March 3rd, 2008 at 4:15 pm, Fan said:

Hi Mark,

Thank you for the good article.

I have one question about how to update suffix tree, would you please give me some suggestions?
Thank you very much.

I have modified your source code in order to insert a new string to the suffix tree, update the suffix tree if new string doesn't exist in the suffix tree. Unfortunately, I was stuck on updating the suffix tree.

Here is an example, first inserts "hello" to the suffix tree which looks like:
Start End Suf First Last String
0 2 -1 1 4 ello
0 1 -1 0 4 hello
0 4 0 2 2 l
0 6 -1 4 4 o
4 3 -1 3 4 lo
4 5 -1 4 4 o

then inserts "bok" to the suffix tree, first character 'b' is added without any problem,
Start End Suf First Last String
0 2 -1 1 4 ello
0 1 -1 0 4 hello
0 4 0 2 2 l
0 6 -1 4 4 o
4 3 -1 3 4 lo
4 5 -1 4 4 o
0 7 -1 5 7 bok

then 'o' is being added to the suffix tree, it was found in the suffix tree, which indicates by start node 0, end node 6. continuing on next character 'k', it first should append 'k' to leaf node 'o' and update its first and last character in the string from (4, 4) to (6, 7), then creates a new edge to represent 'k' and insert into the suffix tree. How should I modify the code on those steps?

The correct suffix tree should look like the following structure:

Start End Suf First Last String
0 2 -1 1 4 ello
0 1 -1 0 4 hello
0 4 0 2 2 l
0 6 -1 4(6) 4(7) ok
4 3 -1 3 4 lo
4 5 -1 4 4 o
0 7 -1 5 7 bok
0 8 -1 7 7 k

on March 4th, 2008 at 11:01 pm, Mark Nelson said:

@Fan:

Well, I'm not sure what you are doing makes any sense. If you already have "hello" in the tree, you can't have "bok" by itself, it has to be a suffix of some existing string in the tree. So when adding the "b", you should have "hellob" in the tree.

Your modified algorithm doesn't.

So whatever you are trying to make is not really a suffix tree.
=Mark

on April 21st, 2008 at 6:54 pm, PT said:

Does any know of a good Trie / suffix tree implementation in c#?

on April 21st, 2008 at 8:39 pm, Mark Nelson said:

@PT:

I'd take a look at codeproject.com, and perhaps search krugle.com.

on April 23rd, 2008 at 1:29 am, sarba said:

@sarba
Hi Mark,
I want to modify your code to find all exact repeat positions of an input string .For example String S=ATAGGATAGC .I want to find repeat (here ATA) and there positions (here 0,5).Can you give me any suggestion about how to modify the code?

on April 23rd, 2008 at 7:46 am, Mark Nelson said:

@sabra: I'm not sure I understand the question.

on April 24th, 2008 at 12:29 pm, sarba said:

hi mark,
Thanks for your attention.ok forget the above question for time being . More simply can you explain me how can I find longest common substring of input string. Suppose I have two string
a. ATATGCATCAG
b. GCATGCACCGA
I want to find longest common substring of the two sequences. What I did is as follow,
I concatenate two string like this S=ATATGCATCAG#GCATGCACCGA then made suffix tree

.Theoretically the edge from root node to the depth est internal node contains the longest common substring.So how can I find the depthest intenal node and its "edge level" from root.

on May 21st, 2008 at 4:24 am, prodvit said:

Hi mark. I read your article and I found it very useful. I'm interesting in suffix trees for building a web search results clustering algorithm, like STC (Zamir et Etzioni, 1998).

I'm interested in understanding the difference between suffix trees and suffix arrays, in particular I would understand advantage in using one rather than other in clustering problems.

Can you help me?

Thanks
Massimiliano

on May 21st, 2008 at 5:57 am, Mark Nelson said:

@prodvit:

I wish I could give you a good answer here, but I don't know a lot about suffix arrays - I need to get up to speed there myself. All I know for sure is that suffix trees are advertised as having good construction costs, and once constructed, should be just as useful as suffix trees.

on May 28th, 2008 at 3:24 am, void said:

Just wanted to let you know that I thought your article was good. Clear and more understandable than alot of other articles on suffix trees I've looked at.

Good work, thanks for sharing.

on May 28th, 2008 at 7:23 am, Mark Nelson said:

@void:

Thanks for the kind words. For some reason this is a tough algorithm to explain, to understand, and to illustrate. Some day in the far off future I'm going to take another crack at it and see if I can do better!

on June 5th, 2008 at 4:34 am, giorgos said:

Hi mark,
thanks for your article!
Is it possible to decribe what happens with the string MISSISSIPPI ?
Actually, i dont understand the step adding the MISSIS suffix.
i read the following link and its really confusing
http://www.allisons.org/ll/AlgDS/Tree/Suffix/

on June 5th, 2008 at 5:15 am, Mark Nelson said:

@girgos,

If you're having trouble with the description on the web site shown in your comment, why not ask the owner of that site for clarification?

on June 5th, 2008 at 11:06 am, giorgos said:

Thanks for your quick response, Mark!

I read your example with Bookeeper, but when i tried to create the tree for MISSISSIPPI, i had some problems.
It would be great and you would help me a lot if you gave us a succinct example with the string MISSISSIPPI too.

Thanks,
Giorgos

on June 16th, 2008 at 6:48 am, xyzzy said:

Hi, thanks for the nice article. It would be better if you add a few examples though, like how strees can be used for finding the longest common substring etc.

on June 16th, 2008 at 7:52 am, xyzzy said:

Ok. Here's one solution I found. When you find a character that's already an implicit node, mark a $ under it so that the next character encountered will be added to this $ also.

on July 1st, 2008 at 3:38 pm, Tamer said:

Hi,

Nice work. I am working on a research project that requires building a large suffix tree (around 20k string, each with 4 to 8 letters). I worked on your code, and i guess the main problem is in the hash array. I mean if i can enlarge the hash array, i can store the large number of edges in the tree.
Could you please provide me with some way to change the hashing function so that to be able to generate a number between 0 and 20k or 30k.

Thanks,
Tamer

on December 5th, 2008 at 9:27 am, Tonda said:

Hello,
this si the best explanation of creating suffix tree I found so far (however I still fully don't understand it, maybe because I am not native English speaker).

I have one question. I have 1MB file with characters representing part of DNA. I need to find longest repeated substring (LRS) but these substrings can be over each other.

Example: BANANAS
1) LRS without overlapping: AN or NA
2) LRS with overlapping: ANA

I need to find the second one. But if I understand at least a little to suffix trees, they allow me to find only the first mentioned LRS. Aren't they? Could you give me some direction, how to find the second one?

Thank you.

on December 6th, 2008 at 1:54 pm, Mark Nelson said:

@Tonda:

To be honest, I don't have a good answer for you. It looks like a tough problem.

on December 8th, 2008 at 7:03 am, pschloss said:

Thanks for your article and code. I was wondering whether you have developed a function to add strings to a pre-existing suffix tree to build a generalized suffix tree. I know you can just concatenate the strings together with unique separators, but this seems to run into problems when you have more than 256 strings and you would seem to generate and process some very long edges that aren't "real". Any suggestions? This seems to get minimal mention in Gusfield and other sources.

on January 30th, 2009 at 10:02 pm, links for 2009-01-30 « My Weblog said:

[...] Fast String Searching With Suffix Trees (tags: cs) [...]

on March 17th, 2009 at 1:52 pm, bbi5291 said:

@Mark Nelson: When you say that each new suffix can be appended in constant time, don't you mean that adding a character to the end of a current suffix takes constant time?
It's certainly not true that it is guaranteed to take constant time only to add a character to the tree as a whole. But it still takes linear time overall, right? Because whenever we do A LOT of work for one character, it was due to not having done much during previous iterations. Or you could say that the active point moves a linear number of times during the whole algorithm, and that each time it moves a constant amount of work is done, so the algorithm is linear overall.

on March 18th, 2009 at 6:38 am, Mark Nelson said:

@bbi5291:

Yes, you are correct, when I say "each new suffix can be added in constant time" I am misusing the term suffix - what I actually mean is "each new character can be added in constant time".

But you are incorrect in thinking that adding a character to the tree is not constant time - only linear overall. Adding a single character is definitely constant time only. The various operations that can occur each time a character is added do not change regardless of the length of the previously seen input. They don't involve scanning back through the the tree. They are, in fact, constant, which is why the algorithm is guaranteed to run in linear time regardless of the input.

We don't do "a lot" of work for just one character. (Mostly because we maintain a suffix pointer at each node - this lets us navigate to the next smallest node with a single dereference.)

on March 19th, 2009 at 12:06 pm, bbi5291 said:

@Mark Nelson:
Seems that we do agree on runtime complexities, then, we just misunderstood each other - yes, I understand that the operation of extending a single suffix by a single character is constant time. When I said "adding a character" what I meant was the process undertaken to transform the suffix tree of the first N characters to the one for the first N+1 characters. Here "a lot of work" can be done at once, but it's still amortized constant, right?

Now, one thing that bothers me is the canonicalization. This can require walking through a linear number of edges at once; how do we know that this is amortized constant as well?

on April 11th, 2009 at 5:09 pm, rodger87 said:

hi

I am working on common pattern searching algorithms. I was using suffix tree,but i wasn't able to implement it for the search of common pattern without giving any input string.
Like, I have 15 sequences and i wanted to make suffix tree of each sequence and search for the common pattern in all the 15 sequences

on April 11th, 2009 at 6:26 pm, Mark Nelson said:

@rodger87:

Sorry don't really understand the question.

From the problem you describe, you need to create and search 15 suffix trees.

- Mark

on April 16th, 2009 at 10:09 pm, rodger87 said:

This is my algorithm:

1 set number of characters to be processed ws = 8000
(note: we assume 8000 characters are processed at one time)
2 compute length of longest common pattern (overlap size).
3 for each sequence, Si, in database do
4 set overlap string Os to empty
5 while not end of sequence Si do

6 set Stmp = |Os| + ws characters of Si
7 construct a suffix tree, ST, for the subsequence Stmp
8 use multiple patterns search against the suffix tree ST
9 record the search result
10 determine the content of overlap string Os
11 update position for next ws characters from Si
12 end while
13 end for

I am not able to code it....in perl...

on April 18th, 2009 at 1:20 am, Illya Havsiyevych said:

Hello,

I've ported your source code to Java
http://illya-keeplearning.blogspot.com/2009/04/suffix-trees-java-ukkonens-algorithm.html

Thanks for reference implementation,
illya

on April 18th, 2009 at 6:53 am, Mark Nelson said:

@Illya:

Thanks, that will be very helpful!

- Mark

on May 16th, 2009 at 11:10 am, Tagz | "Fast String Searching With Suffix Trees at Mark Nelson" | Comments said:

[...] [upmod] [downmod] Fast String Searching With Suffix Trees at Mark Nelson (marknelson.us) 0 points posted 10 months, 1 week ago by jeethu tags search suffix algorithm [...]

on May 16th, 2009 at 2:51 pm, Zubair said:

Hi Mark,

Thnx for writing such a beautiful article. I m working on building a "weighted suffix tree" for particular application. Therefore I have to modify the whole algorithm of developing the suffix tree and its search algorithm. Wht i believe is some minor changes in the current code and

INTELLIGENT guidance will lead me to a better solution. I have already implemented the prototype for my work but the efficiency of the implementation in term of processing is quite low. I would like for your coordination or help in ma work . I need to update the suffix tree for a certain period of time by giving new strings to tree. If a particular suffix exists all it need to do …….(lets assume nothing to do). If new suffix arrives it have to update the tree on run time ……. Thats only the part of problem . Hope u can guide me regarding this As I m still a student :). Further more if ask for a personal contact with you regarding ma work :) ……

on May 19th, 2009 at 5:51 am, Mark Nelson said:

@zubair:

sorry i won't be able to give you personal guidance on this, perhaps you could contact others who have posted here.

on May 24th, 2009 at 2:53 pm, Illya Havsiyevych said:

Mark,

FYI refactored Java code with some pictures and test runs.
http://illya-keeplearning.blogspot.com/2009/05/suffix-trees-refactored-java-code.html

Main ideas:
- reuse java hash map;
- better OOP.

Thanks,
illya

on June 8th, 2009 at 1:16 am, Illya Havsiyevych said:

FYI,
Suffix Trees Java Applet - build and visualize your tree.
http://illya-keeplearning.blogspot.com/2009/06/suffix-trees-java-applet.html

Thanks,
illya

on June 13th, 2009 at 11:59 am, Maria said:

Hi Mark,

I am trying to understanding how the construction of a suffix tree works. I have tried to continue the construction of the tree for BANANAS starting with the steps on Figure.3, but I have a problem:

as a next step in Fig.3 I want to build a tree for BANA, and I think I just need to add an 'A' to the three edges of the tree for BAN (I don't add another edge). So, my tree for BANA will have again three edges (no further branching) starting from the root and having as strings BANA, ANA, and NA. My problem is that I am not sure what happens with the suffix 'A'. It does not appear (in what I have done) explicitly on a separate edge. If we had a suffix TRIE we would have an edge for 'A'. Now in the suffix TREE we have an implicite node, which implicitely separates 'N' and 'A' of the edge 'NA'. My question is: if I query for 'A', will the system "take into account" this implicit node between 'N' and 'A', and thus see the 'A'? Should I think about this in this way? I may be have trouble with what "implicit" mean.

The same problem is with the tree for BOOKK on Figure 5. I don't understand why we don't have an edge for the suffix 'K' (the smallest suffix of BOOKK, apart from the empty one). If what I have written above for the identification (matching) of the suffix 'A' is correct, then 'K' will be "identified" on one of the edges having 'KK' as a string or substring. Will it be the edge 'KK' (the one between nodes 0 and 2)? Or may be on the edge 'BOOKK' (the one between nodes 0 and 1)?

Thanks a lot in advance.

on June 13th, 2009 at 4:24 pm, Mark Nelson said:

@Maria:

Check out Ilya's java app and you can see the results of construction in real time. That might be helpful.

on June 15th, 2009 at 12:35 pm, xutao said:

Hi Mark,
Good article and useful code. However when I walk_tree for "banana$", it took 2816 iterations! It is certainly nowhere near linear. Please help

on June 15th, 2009 at 3:00 pm, xutao said:

The following code is to help query the tree with a string. If the query is a substring of a suffix, it will return the position substring.

example code following tree construction
[c]

int start, end;

search_tree(q, start, end);

cout= strlen(query)) {

cout

on June 15th, 2009 at 3:01 pm, xutao said:

PLAIN TEXT

C:

```
/*
input a string to search query []
output start_index and end_index on the string (tree) search against
*/
void search_tree(char query [], int  & start_index,
    int  & end_index)

{
    int start_node = 0;
    int qp=0; //query position
    start_index = -1;
    end_index=-1;

    bool stop =false;
    while(!stop){
        Edge edge = Edge::Find(start_node, query[qp]);
        if ( edge.start_node == -1) {
            stop=true;

            break;
        }
        if (start_node ==0) start_index = edge.first_char_index;
        print_edge(edge);
        for (int i = edge.first_char_index; i <=edge.last_char_index; i++){
            if(qp>= strlen(query)) {
                //cout<<"whole query matched"<<endl;
                stop=true;
                break;
            }
            else if (query[qp] == T[i]){
                //cout<<query[qp]<<" ";
                qp++;
                end_index = i;
            }
            else{
                //cout<<"partially matched"<<endl;
                stop=true;
                break;
            }
        }
        if (!stop){ //proceed with next node
            start_node = edge.end_node;
            if(start_node==-1) stop=true;
```

```
                cout<<"next node     "<<start_node;
            }
        }
}
```

on July 3rd, 2009 at 3:47 pm, Illya Havsiyevych said:

Hello,

FYI,
Some other Suffix Trees based Java Applets:
* Generalized Suffix Tree - http://illya-keeplearning.blogspot.com/2009/06/generalized-suffix-trees-java-applet.html
* Diff - http://illya-keeplearning.blogspot.com/2009/07/suffix-trees-based-diff-java-applet.html

Thanks,
illya

on November 25th, 2009 at 1:39 pm, Legistrate said:

I found your implementation the most useful of the various implementations out there, but the divergence from Gusfield was very confusing at first. Then there are also some things that Gusfield doesn't seem to address directly.

For example, If you just wanted to implement suffix links, how would you know what the characters are when you walk up one edge to arrive at the parent (y in the text). You need to know more than just the one character leading from that parent to the edge so you can walk back down after you traverse the suffix link with possibly multiple nodes.

I ask because in your code I finally realized that you don't actually walk down the edge to either a newly created inner node or to new children. In this way you have the Suffix start track the position of the char that indicated the edge to the correct node. While I'm still fuzzy on why exactly that works, in general I would say it does.

The start being larger than stop signals an external node, and the converse an internal node. So in a phase, that means extensions are applied Rule 1, Rule 2(splits), Rule 2(new kids), Rule 3. But when I use the string "mississippi$" I have an unexpected split in the last phase. In Phase 10 adding (the last)'i' Suffix start is 9. The active point splits ppi$ into p: pi$ and i$, and the active point moves to the root. Thus start is incremented by one(10) and the next extension should be explicit(start==stop) which it is since 'i' is already an internal node off the root. Rule 3 breaks the loop and we move to the next phase('$'). But here, the stop has increased too, and now the algorithm thinks that the active point should be an implicit node. This is not true, so a split is performed on the explicit node, and there is now an extra empty child in the final tree.

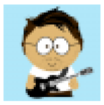
Could you maybe mention a little of how you devised a way to have a start counter for tracking the relevant index in the string as you move from extension to extension rather than the for loop covering each extension?

on November 25th, 2009 at 4:06 pm, Legistrate said:

Hmm, well it looks like I made an implementation mistake. The algorithm does correctly handle my Phase 11('$'). I am still unsure of how you discovered the proper way to track the position of the active extension, but as the code works, I guess I could read through it a few more times. I do however thing that the suffix pointers added are not always necessary(ie extras), but that could be a mistake too :P.

on December 9th, 2009 at 8:31 pm, Its Entirely True » C# Suffix Tree said:

[…] I was unable to find a C# implementation of the suffix tree so I ported one I found at Mark Nelson's Blog. The project of the C# port is located here […]



on February 10th, 2010 at 5:12 am, ziman said:

Thanks much for this overview, it definitely helped me much to get an A at today's exam! :)

on February 16th, 2010 at 12:13 pm, suffix tree resources « KcodeL said:

[…] http://marknelson.us/1996/08/01/suffix-trees/ […]

on March 5th, 2010 at 4:54 am, ayan_2587 said:

hello..
i have a small doubt here..
i did not understand the "canonize" function here.. why are we using it and what is exactly its role over here..
please help !!!

on November 19th, 2010 at 5:58 pm, cparker7 said:

I have a C# implementation based on your documentation and source code. I have implemented FindAll() based on your comments: from point at which match is found, traverse subtree, and each leaf indicates a match. There is a bit of computation which must be done based on the path from root to leaf. I have that all working.

There is a bug which I cannot figure out. There are certain trees which generate only N leaf nodes, where there are N + 1 actual matches. The simplest test case I can generate is the string "anasan", and the query string "a" . The tree that I am generating looks like:

(0-0:'a')=>3
(1-5:'nasan')=>1
(3-5:'san')=>4
(1-5:'nasan')=>2
(3-5:'san')=>5
where each link has the template:
({first}-{last}:'string')=>{node}

When searching for 'a', we find the edge from 0-3, and then traverse to all the leaves.

Thoughts?

on December 5th, 2010 at 7:00 pm, Mark Nelson said:

@cparker7:

I'm not sure I follow your notation, but it doesn't appear that you have created a valid suffix tree. For starters, you should have a leaf for character 'n', the smallest suffix.

Each edge in the tree has a parent and a child node, and a string on the edge. So we could represent the suffix tree in Figure 2 like this:

0-1:BANANAS
0-2:A
0-3:NA
0-4:S
2-5:NA
2-6:S
3-7:S
3-8:NAS
5-9:NAS
5-10:S

Use that notation to describe your tree and let's see what it looks like.

- Mark

- Mark

on December 17th, 2010 at 5:12 pm, Bla said:

This code that you shared here is a really bad design and poor implementation.

on March 17th, 2011 at 9:01 am, tueken said:

After reading and thinking for quite a while, i start to understand how the algorithms works, especially the concept of "active point". Your explanation is great. Thanks a lot.

on April 2nd, 2011 at 1:50 am, whiteBomb said:

Everyone's a critic, cool~

on April 5th, 2011 at 5:13 pm, Xplode said:

Two more steps to the fastest string search algorithm ...

on April 17th, 2011 at 2:12 am, whiteBomb said:

Hi Mark!
I want to use vector to store all the edges, since I insist on using Hash table is extremely difficult for me(may be some guys) to REMOVE edges.

what's your point on using vector and hash table? Please tell me about their efficiency or convenience.

Thank you!
Waiting for your kindly reply.

on April 18th, 2011 at 5:12 am, Mark Nelson said:

@whiteBomb:

Look into the suffix array data structure:

http://en.wikipedia.org/wiki/Suffix_array

on May 1st, 2011 at 11:39 am, Pravesh Parekh said:

Well this is the first time that I am looking at a suffix tree and I must say that it was a nice and refreshing to look at your page. The diagram clearly made me understand how to represent a suffix tree followed by edge compression. I do not need the coding of this so I haven't gone into the complex issues that you later brought up but I am pretty sure that they would be great as well. One thing that you could include here is how the tree was made: the removing one letter and traversal. All good otherwise. Cheers!

on August 16th, 2011 at 8:46 pm, Rayan Yousif said:

Hi Mark , really i'd like to thank you for all that you've offer ,i have been interested in searching algorithms , and i'm looking for a new or an improved algorithm for large strings indexing (string of tera-bytes) , can you advice me to any? Does suffix tree suitable more ?
Thank You again ..

on August 17th, 2011 at 8:07 am, Mark Nelson said:

@Rayan:

A suffix tree is normally an in-memory structure, so if you are indexing terabytes, you might run into issues of memory exhaustion.

- Mark

on August 29th, 2011 at 3:49 pm, Ivan said:

I find it a bit confusing when you stated "Any node that we create as a leaf will never be given a descendant". In Figure 6-1, the node '2' is later added two descendant '6 and '7' in Figure 6-3. I guess you may argue that the node 6 in Figure 6-3 is actually the previous node '2', but wouldn't it be better if you use the same number for the nodes? To better illustrate, see here:

https://skitch.com/iveney/fw3ci/fast-string-searching-with-suffix-trees

on August 29th, 2011 at 4:10 pm, Mark Nelson said:

@ivan:

I agree - that is not explained clearly.

When I am speaking of a leaf node, I identify it by the suffix that terminates at that node - not the number. The number is a convenience used in some places, but the suffix is what is important.

In the picture you identified, the node with number 2 corresponds to suffix KKE. After the construction process, KKE is still a leaf node, but the edge was split, so there is a new interior node. That node has number 2, which is obviously a confusing point.

- Mark

on August 31st, 2011 at 9:20 am, Suffix tree: an illustration of the algorithm « Use You Imagination said:

[...] and found Ukkonen's linear time online algorithm quite hard to understand. Mark Nelson has a good article about this, however, in my opinion the writing is not so great and some part is [...]

on October 20th, 2011 at 2:35 pm, Burrow Wheeler transform, Suffix Arrays and FM Index « Homologus said:

[...] version of constructing suffix trees was presented in a paper by Edward McCreight in 1976. I found this link most helpful on suffix [...]

on November 29th, 2011 at 1:29 pm, Tony Bruguier said:

Mark,

Thanks for writing this article. It's quite helpful. I think I understand almost everything, except one sentence. You say:

"Knowing how the construction algorithm works, you can see that if you find a certain character as a descendant of a particular suffix, you are bound to also find it as a descendant of every smaller suffix."

How can we guarantee this?

Thanks
Tony (different from another Tony above)

on December 1st, 2011 at 11:35 am, Mark Nelson said:

The sentence I have given describes an invariant property of the suffix tree. If you find that character 'D' is a descendant of 'ABC', it means that the tree *must* also contain BCD and CD. If this was not the case, it wouldn't be a suffix tree.

The algorithm must be written to guarantee this invariant holds.

"How" is what the article is all about!

- Mark

on December 13th, 2011 at 2:47 pm, Garret Wilson said:

Mark, I cannot express how invaluable this article has been to me in implementing a syntax tree. I needed a Java syntax tree implementation, and after days of wrestling with tutorials (mostly yours), I am starting to wrap my head around it. I am by no means an algorithm expert, but after the mental sweat I've poured into this, I've come away with a few insights that make your algorithm a bit clearer for me. Please don't take any of these thoughts negatively---they merely reflect how my mind has come to understand the algorithm.

The Suffix class was a little confusing from its name, as it represents a suffix from the root, yet there is some optional "leftover" part past the active node. In my implementation I called this class State following Ukkonen.

The Suffix/State and Edge end offsets you use are inclusive. In 2011, in my mind anyway, many programmers are accustomed to end positions being exclusive, and indeed it makes many of the length calculations cleaner and more intuitive.

Several equivalent variables were arbitrarily intermingled, and logic could have been further refactored and isolated. For example, in Edge::SplitEdge() the new near edge is created extending from suffix.origin_node---which should always be equivalent to the old edge's edge.start_node. Using the Suffix origin node when splitting an edge obscures the fact that edge-splitting logically is an operation completely independent of suffixes. That is, to split an edge I merely need to provide the edge to split and a length position at which to split (even though this length may have originally come from the suffix). This loosens the coupling of edge-splitting logic from the overall suffix-tree building operation, making the isolated routine more understandable and testable on its own.

In the comments to stre2006.cpp, saying "... we ... set first_char_index > last_char_index ... to flag [an explicit node]..." was confusing to me, and I looked for code that set first_char_index > last_char_index as a flag. What you mean, of course, is that the algorithm sometimes advances first_char_index > last_char_index, and we interpret this situation as such a flag. The difference is subtle, and may only reflect my particular way of thinking about this.

Lastly, whenever there is an infinite for(;;) loop, I pause to see if there is some more concise representation I am missing in my iteration logic. In fact, if I understand this correctly, it turns out that in the present implementation, when going to the next shorter suffix from the origin node (active.first_char_index++), first_char_index may be greater than last_char_index, but only by two characters (or, with an end-exclusive implementation such as mine, by only one character). When this occurs, we not only know that there is no shorter suffix in this round, but that the previous iteration must also have been explicit, which means that it must have added a new edge. Once we note this, we can exit the loop immediately. I accomplish this by turning the for(;;) loop into a do{} while(state.nextSmallerSuffix()) loop, in which nextSmallerSuffix() returns false if start>end after advancing start++ (again, using exclusive end positions). The current implementation in this situation needlessly loops back around and does the same check it made before, finding the edge that was added the previous time around before breaking. I could be wrong about this, but so far my implementation is passing the validation tests.

Again, these observations are less criticisms of your approach than awe that I was able to understand this at all, thanks mainly to reading your article. I have finished my Java implementation. After tidying up the code and added methods to make the class useful in actual string processing, I'll post a link here. Thanks again for publishing this.

on December 13th, 2011 at 3:03 pm, Mark Nelson said:

@Garret:

>Again, these observations are less criticisms of your approach

No worries, as the years go by I always see plenty of room for improvement, and I would really like to rewrite this article someday. I finally did a rewrite of an LZW article after over 20 years, and I was happy with that - maybe this one is next!

Thanks for your comments, I hope they help others trying to work through this stuff.

- Mark

on December 15th, 2011 at 1:50 pm, Garret Wilson said:

(In my comment above, I of course meant "suffix tree" instead of "syntax tree".)

I have posted an overview of suffix trees and their application, along with links to my implementation in Java:

http://www.garretwilson.com/blog/2011/12/15/suffix-trees-java.xhtml

I referenced your article. Thanks again.

Garret

on January 4th, 2012 at 11:24 pm, Jason Young said:

Mark,

Thank you for the code and kind explanation.

I think I can't fully understand the concept of the suffix tree..

In your code, what is the role of origin_node of Suffix?

thank you.

Jason.

on January 18th, 2012 at 12:32 am, Sergey Makarenko said:

Mark,
Thanks for a great article.
I believe that in streed2006.cpp signature of

PLAIN TEXT
C:

```
ostream &amp;operator&lt;&lt;( ostream &amp;s, Aux
const&amp;a )
```

should include const for the second parameter. Otherwise numbers are outputted instead of characters. (At least for MS Visual Studio 2010).

on January 29th, 2012 at 12:38 am, suffix tree « demonstrate 的 blog said:

[...] suffix tree，但是没见到后文了。网上能找到的一些实现如比较早的，C 的版本感觉局限性比较大（libstree 与这个），有两个 C++ [...]

on April 20th, 2012 at 2:39 am, to do or learn | Pearltrees said:

[...] Even you can make a tree McCreight's original algorithm for constructing a suffix tree had a few disadvantages. Principle among them was the requirement that the tree be built in reverse order, meaning characters were added from the end of the input. This ruled the algorithm out for on line processing, making it much more difficult to use for applications such as data compression. In fact, the reduction in the number of nodes is such that the time and space requirements for constructing a suffix tree are reduced from O(N 2) to O(N). In the worst case, a suffix tree can be built with a maximum of 2N nodes, where N is the length of the input text. Fast String Searching With Suffix Trees [...]

on May 23rd, 2012 at 4:50 am, Can someone please explain Ukkonen's suffix tree algorithm in plain english? | PHP Developer Resource said:

[...] excessive use of mathematical symbology. The closest to a good explanation that I've found is http://marknelson.us/1996/08/01/suffix-trees/, but he glosses over various points and some aspects of the algorithm remain [...]

on July 24th, 2012 at 12:37 pm, eateshkandpal said:

Hi Mark,

This was a very helpful explanation about suffix trees i have already gone through it twice. I wonder if few comments could be marked which too add up to the article for eg. the one explaining 'active points'.

on August 5th, 2012 at 3:56 am, random() » Blog Archive » Suffix Trees, Suffix Arrays and Ukkonen's Algorithm said:

[...] Mark Nelsons c++ implementation [...]

on October 15th, 2012 at 12:39 am, sush said:

Can you please explain, how can the suffix tree be used to find all the unique substrings of a string?
I have many strings and my task is to find unique substrings of all those strings and then take their union. Thus, finally I need to find all the unique substrings in the given set of strings. Can this whole task be implemented using suffix trees efficiently(in terms of time)? If yes, can you please explain the algorithm for that?
I have searched the net, but I didn't find any good solution for my problem. That's the reason I am asking here.
I am waiting for a kind and detailed reply.

Thanks

on October 15th, 2012 at 12:45 am, sush said:

I forgot to mention, it would be really nice if you rewrite the article and the source codes as they are around 6 years old!

Thanks.

on October 15th, 2012 at 6:14 am, Mark Nelson said:

Actually, it is 16 years old, and yes, it certainly could use a an update.

I don't think a suffix tree is the structure you need to find all the unique substrings in a given string. It doesn't have that information.

- Mark

on October 16th, 2012 at 12:45 am, sush said:

So, can you please suggest a data structure for my problem with your experience? I thought of using trie but, my single string's length can go upto 2000. So, trie would not be space efficient.

What I am thinking about is, to use the suffix tree to find the prefixes of all suffixes(which are itself the substrings). Can't this be done by traversing the from root to leaf and taking all the prefixes of that suffix? The time complexity would be O(n^2). So far, I have read that no algorithm can give substrings of a string in less than O(n^2) time. So, wouldn't it be feasible to do it with suffix trees?

on October 16th, 2012 at 6:27 am, Mark Nelson said:

If you want to generate all the substrings of string s, I would suggest that you do so using a simple algorithm that walks through the string and stores all the substrings it finds in a set. This below needs cleanup, it is just giving you the idea:

PLAIN TEXT

C:

```
unordered_set<string> results;

for (i = 1 to s.size() -1 )
   for ( j = 1 to s.size() - 1 ) {
         string sub = s.substr(i,j);
         results.insert(sub);
   }
}
```

on October 16th, 2012 at 10:39 am, sush said:

As I told you earlier, I need an O(n^2) algorithm but what you are suggesting is O(n^2lgn). Apart from this, I have m number of strings. So, for finding unique substrings for all of them even using the best algorithm of O(n^2)would take overall O(n^2 *m) time. Thus, I am just thinking about creating a generic suffix tree and then doing it in much lesser time. But for that I need to know what is the way to process all unique substrings from a given string's suffix tree. And that's what I am asking you.
The algorithm you told is almost like brute force and is known to me.

Please provide any ideas for solving my problem using suffix trees.
Thanks.

on October 16th, 2012 at 1:49 pm, Mark Nelson said:

@Sush

>As I told you earlier,

And as I told you earlier, I don't think that a suffix tree is going to do you any good.

The algorithm I gave you will operate in O(n^2) under normal circumstances - it uses an unordered_set to store the substrings. unordered_set is a hashed container, so insertions will normally take place in constant time. Any hashed container obviously has much worse behavior in the worst case, but that's usually not what we care about. It would be very hard to construct a pathological case.

It seems that somehow you have decided that both the suffix tree construction algorithm and I are responsible for solving your problem. I assure you that neither of us is.

- Mark

on October 16th, 2012 at 2:26 pm, sush said:

OKay.
Thanks for the kind info. I am trying to code in C. So, can you please tell me that is there any container like unsorted_set in C? Or some other replacement for it?

on October 16th, 2012 at 2:29 pm, Mark Nelson said:

>So, can you please tell me that is there any container like unsorted_set in C?

Not in the C standard library.

I'm sure you can find many different container libraries to choose from. However, for this project, the complexity of what you are trying to do is going to be fairly larger in C than in C++, Java, or another language with built-in containers.

- Mark

on October 17th, 2012 at 2:21 am, sush said:

Yes! Thinking about the same, I am planning to code it in C++. So far, I know that there is a set in C++ which inserts in O(lgn). Can you just point out if there is some other container which inserts in O(1) in C++? Or will I need to get some library in C++ also?
And one more point which is bugging me is that, in your code you have used "s.substr(i,j)". Lets say k=j-i. Thus length of the substring is k. So, wouldn't it take O(k) time to extract the substring out of the original string(or even copying the substring to some other string and then using it)? So, is there a way in C++ to do it in constant time?
If this step is done in constant time and we have some container in C++ which inserts in constant time, then your whole algorithm will do it in O(n^2) time.
I know your blog is to provide suffix tree implementation and not to solve my problem but it would be really kind of you if you just answer these questions.

Thanks.

on October 17th, 2012 at 5:11 am, Mark Nelson said:

>Can you just point out if there is some other
>container which inserts in O(1) in C++?

C++ 98 introduced the set container to the standard library. Usually based on a red-black tree, it offers lg(n) performance for most types of access.

TR1 added unordered_set, which is a hashed container. It has been supported in most compilers for quite some time. Depending on which version of the compiler you are using, it may be in the std::tr1 namespace, or in the std namespace. By design, it has constant insertion and lookup times except for pathological conditions.

>So, wouldn't it take O(k) time to extract the substring

There are all sorts of string operations that are O(n) where n is the length of the string. Mostly people don't care too much about this because strings are usually short. So there is little incentive to optimize.

If you are dealing with some huge computational burden imposed by string operations, you might want to subclass std::string and create your own methods. But the times when this is necessary will be very rare.

- Mark

on October 17th, 2012 at 10:02 am, sush said:

Thanks a lot mark. I got a lot of help from you. I know this is a blog but are there other means to contact you(via email or something)? I would really like to show the problems I face in future, so that if you wish you can help me with your experience.

on October 18th, 2012 at 7:02 am, sush said:

Just FYI, I got the solution for my problem. It will be solved by using trie. I will add all the strings to the trie and then get the unique substrings.

on December 14th, 2012 at 10:08 am, Algorithm of the Year 1973 & Project Euler 26 « John Ruddy said:

[…] I had a relatively large set of strings generated. I used Mark Nelsons implementation of the Suffix Tree in C# in order to complete this. By generating a suffix tree for each of the strings I managed to build […]

on January 22nd, 2013 at 2:35 pm, hh said:

Thank you very much for this great post. I wish more people would be capable of explaining algorithms in such an understandable way. Kind of reminded me of the Sleator/Tarjan Splay tree (not the data structure, but the transparency of the explanation).

I had much fun implementing this in the Scala programming language: https://github.com/Sciss/ContextSnake/tree/v0.0.1

It's still quite close to your C++ code, so not exactly functional style, but in the next refactoring step it will be more 'scalaish'. Like Garret, I used exclusive stop indices which make things a bit cleaner, and there are a few redundant things that can be omitted.

(Next step will be implementing a probabilistic estimator of the next character of a given suffix. Any ideas how to add this to the existing structure? I will need to efficiently look up all the outgoing next characters of an explicit node. Perhaps just a hash-map of that node's index.)

on March 12th, 2013 at 3:18 am, Ukkonen's suffix tree algorithm in plain English? | Everyday I'm coding said:

[…] 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 […]

on March 15th, 2013 at 11:01 am, suffix tree simple explaination | simple2013 said:

[…] http://marknelson.us/1996/08/01/suffix-trees/ […]

on April 25th, 2013 at 12:09 pm, azal said:

Hi there,

Thank you so much for writing this comprehensive article!
I am working on a problem where, given a string of length n, I need to calculate the occurrence counts of all the "consecutive" substrings of length of m=1..n.
For instance if the string is 0 0 1 0 1 1
then I would need to calculate the number of times I see:
0
1
00, 01,10,11
001,010, 101,011
0010,0101,1011
00101,01011
001011

So one approach would be to construct the tree in O(n) and then search for each pattern of length m in O(m). However, since there are n-m+1 patterns for each m, and m ranges from 1..n the overall computational complexity would be quite high! (I believe I can probably do better if I construct a suffix array from the suffix tree, to use for subsequent search).

But my main question is this:

In a 2005 power-point presentation titled "Suffix tree and suffix array techniques for pattern analysis in strings" (which can be found here: http://www.cs.helsinki.fi/u/ukkonen/Erice2005.ppt) Ukkonen mentions
"Thm: Suffix tree Tree(T) gives complete occurrence counts of all substrings motifs of T in O(n) time, (although T mauve O(n^2) substrings!)"
But he's left no reference. I was wondering if this follows directly from the construction algorithm described here, i.e. during construction (of the tree) one can simultaneously obtain/update the counts.

I'd appreciate any input/insight on this.

Thanks!

on April 25th, 2013 at 12:23 pm, Mark Nelson said:

I'm not familiar with that reference, you might try going directly to Ukkonen.

- Mark

on April 25th, 2013 at 3:24 pm, azal said:

Thanks for getting back to me so quickly :)

I just wanted to make sure that it's not like the STree (as construced here) happens to somehow contain the occurrence counts well, with no additional processing.

But that doesn't seem to be the case anyway.

Thanks again.

on April 26th, 2013 at 3:29 am, Victor Liang said:

Hi Mark,

Thanks for your great work. But I have a place confused in your code STREED2006.cpp.

In the function

AddPrefix(Suffix &active, int last_char_index),

there is a line about adding new edge,

Edge *new_edge = new Edge(last_char_index, T.N, parent_node).

My question is why the new edge is defined starting from last_char_index to the T.N?

In my understanding, last_char_index is the character I am reading from the beginning of the input string, T.N seems the end position of the input string. Why should I insert the entire string at first? Why the insertion process is not done by adding one character step by step incrementally?

Thanks.

on June 1st, 2013 at 4:32 pm, Ukkonens suffix tree algorithm in plain English? - Tech Forum Network said:

[…] 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 […]

on November 7th, 2013 at 3:02 pm, Ukkonen's suffix tree algorithm in plain English? | Ask Programming & Technology said:

[…] 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 […]

on October 7th, 2014 at 4:12 pm, saadtaame said:

Awesome article and code is clean and readable as well. I want to propose a way in which program might be optimized. Since there are at most 2N nodes in the suffix tree, the number of edges is going to be no more than 2N-1. So you can use a table for edges as you do for nodes (this gives constant time access to edges).

on October 7th, 2014 at 4:21 pm, saadtaame said:

Forgot to mention that you can use the end of string character '' as a distinguished character. This way, the user does not have to worry about adding a special character at the end. I think that all you need to change is the less sign to <= sign in the main.

on November 10th, 2014 at 5:12 pm, Data Structures and Algorithms Tutorials | TheShayna.Com said:

[…] with Explanation 4 […]

on November 13th, 2014 at 12:18 pm, Memory Leak in suffix tree c++ | Ngoding said:

[…] used the library streed2006.cpp from source. The code has memory leak in deletion of edges. I cleared the number of edges from hashtable using […]

on November 16th, 2014 at 11:44 pm, nithinuppalapati said:

Hi,

Can i traverse the suffix tree created using the above code in postorder?
Also i would like to know the memory of the suffix tree after its construction

on November 22nd, 2014 at 1:47 pm, How to: Ukkonen's suffix tree algorithm in plain English? | SevenNet said:

[…] 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 […]

on December 4th, 2014 at 3:54 pm, the-tech-blog.com said:

Great items fro you, man. I've be aware yoir stuff prior to and
you're simply extremely wonderful. I really like what you have bought right
here, really like whst you are saying and the way during which yoou ssay
it. You're making it entertaining and you still care for to stay
it wise. I cant wait to readd far more from you. Thatt is really a great web site.

on December 6th, 2014 at 2:22 am, Solution: Ukkonen's suffix tree algorithm in plain English? #dev #it #computers | Technical information for you said:

[...] 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 [...]

on December 9th, 2014 at 4:44 pm, Fixed Ukkonen's suffix tree algorithm in plain English? #dev #it #asnwer | Good Answer said:

[...] 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 [...]

on February 13th, 2015 at 12:04 am, student101 said:

Thanks for the great article and sample code Mark! It is extremely efficient compared to many other implementations I've run across. I really appreciate it. I'm working on an academic project now and the suffix tree will really help out. I'm working through your code now to determine the most efficient way to find repeated substrings using your implementation as the basis. Thanks again!

on February 13th, 2015 at 2:09 am, Mark Nelson said:

I'm glad it is still proving useful after all these years!

- Mark

on March 8th, 2015 at 4:24 am, Ukkonen's suffix tree algorithm in plain English? - Technology said:

[...] 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 [...]

on March 29th, 2015 at 11:23 pm, Ukkonen's suffix tree algorithm in plain English? - TecHub said:

[...] 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 [...]

on April 6th, 2015 at 12:46 am, gcapell said:

Thanks very much for the article.

Go implementation at https://github.com/gcapell/suffixtree in case it's useful to anyone.

on April 19th, 2015 at 8:10 am, Naive approaches facing too little time | The donraksaid:

[...] inevitable truth was that my algorithm sucked. Looking for other possibilities, I stumbled upon the suffix tree, which can be used to find all substrings in O(n), but I don't have the time and the [...]

on April 26th, 2015 at 9:19 am, baiwenlei said:

hi, mark

Thanks for this great article. I read the code, there is something seems make no sense.

```
int Edge::SplitEdge( Suffix &s )
{
Remove();
Edge *new_edge =
new Edge( first_char_index,
first_char_index + s.last_char_index - s.first_char_index,
s.origin_node );
new_edge->Insert();
Nodes[ new_edge->end_node ].suffix_node = s.origin_node; // --> what does this mean? why set new node's suffix pointer to his parent? This doesn't make any sense.
first_char_index += s.last_char_index - s.first_char_index + 1;
start_node = new_edge->end_node;
Insert();
return new_edge->end_node;
}
```

Any idea about the comment? thanks.

on July 27th, 2015 at 7:52 am, Getting started with competitive coding | Sameer Chaudhari said:

[...] Tree : Tutorial, Tutorial, Intro, Construction [...]

on November 26th, 2015 at 1:07 am, How can we use Ukkonen's suffix tree to identify all the common substrings within a document. vc++ | Questions said:

[…] was able to generate ukkonen suffix tree for a document based on http://marknelson.us/1996/08/01/suffix-trees/ . And search for a given substring. But still I could not find a way to identify all the common […]

on April 29th, 2017 at 11:06 am, Ukkonen’s suffix tree algorithm in plain English? – Xclusive Developers Blog said:

[…] 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 […]

on May 21st, 2017 at 11:55 pm, 动态规划与# 39；S后缀树算法在平原英语吗？ – CodingBlog said:

[…] 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 […]