EECS764 - ANALYSIS OF ALGORITHMS

RESEARCH PROJECT REPORT

STRING MATCHING PROBLEMS AND THEIR ALGORITHMS

In this paper, a short review of various strings matching algorithms are being discussed. This paper mainly focuses on the construction, improvement and applications of Suffix tree algorithm in diversified fields.

RAGAPRABHA CHINNASWAMY
2830383

TABLE OF CONTENT

1. Introduction

String matching is a very important subject in the wider domain of text processing. Although data are memorized in various ways, text remains the main form to exchange information. This is particularly evident in literature or linguistics where data are composed of huge corpus and dictionaries. This applies as well to computer science where a large amount of data is stored in linear files. And this is also the case, for instance, in molecular biology because biological molecules can often be approximated as sequences of nucleotides or amino acids. Furthermore, the quantity of available data in these fields tends to double every eighteen months. This is the reason why algorithms should be efficient even if the speed and capacity of storage of computers increase regularly. The present day pattern-matching algorithms match the pattern exactly or approximately within the text. The exact matching problem is the following: we are given a string and a pattern, which are sequences over an alphabet, and we have to find all the occurrences of the pattern in the string. While exact string matching is more commonly used in computer science, it is often not useful in biology. One reason for this is that biological sequences are experimentally determined, and may include errors: a single error can render an exact match useless, where approximate matches are less susceptible to errors and other sequence differences.

1.1. Motivation of Research

Large string datasets are common in a number of emerging text and biological database applications. Common queries over such datasets include both exact and approximate string matches. These queries can be evaluated very efficiently by using a suffix tree index on the string dataset. A suffix tree is a data structure that exposes the internal structure of a string in a deeper way than does the fundamental preprocessing algorithms. Suffix trees can be used to solve the exact matching problem in linear time, but their real virtue comes from their use in linear time solutions to many string problems more complex than exact matching. Moreover, suffix trees provide a bridge between exact matching problems and inexact matching problems and it has its application in diversified fields.

2. Problem Formulation

Model of computation:

String searching algorithms find a place where one or several strings (also called patterns) are found within a larger string or text.

Input: Pattern and the Text

Output: Positions in Text where Patterns start.

Solution formulation:

Applications require two kinds of solution depending on which string, the pattern or the text, is given first. In this paper, we will discuss the two ways of efficiently solving the above problem.

- Preprocess Pattern: KMP Algorithm, Rabin Karp Algorithm, Finite Automata, Boyer Moore Algorithm.
- Preprocess Text: Suffix Tree

All the algorithms work in two phases: i.e., the preprocessing phase and the search phase. In the preprocessing phase, these algorithms process the text and use this information in the search phase to reduce the total number of character comparisons and hence reduce the overall execution time. The efficiency of an algorithm mainly depends on the search phase.

3. Survey of Existing results and comparison

3.1. Naïve Algorithm

This algorithm finds all valid shifts using an iteration that shifts using iteration that compare pattern for each of the possible n-m+1 values of shift. The running time of the algorithm is O((n - m +1)m), which is clearly O(nm). Hence, in the worst case, when the length of the pattern, m are roughly equal, this algorithm runs in the quadratic time.

Examples of Naive Pattern Matching:
The worst case of Naive Pattern Searching occurs in following scenarios.

- When all characters of the text and pattern are same.
  txt[] = "AAAAAAAAAAAAAAAAAA"
  pat[] = "AAAAA"
- Worst case also occurs when only the last character is different.
  txt[] = "AAAAAAAAAAAAAAAAAB"
  pat[] = "AAAAB"

The main disadvantage of naive string matching algorithm involves too many repetition of matching of characters. This method is very inefficient method because it takes only one position movement in each time.

3.2. KMP (Knuth Morris Pratt) Pattern Searching

The disadvantages of naive pattern matching algorithm can be overcome by KMP algorithm. The Naive pattern Searching algorithm doesn't work well in cases where we see many matching characters followed by a mismatching character. Following is the example.

txt[] = "AAAAAAAAAAAAAAAAAB"
pat[] = "AAAAB"

Knuth, Morris and Pratt proposed a linear time algorithm for the string matching problem. A matching time of O(n) is achieved by avoiding comparisons with elements of 'S' that have been previously been involved in comparison with some element of the pattern 'P' to be matched. The basic idea behind KMP's algorithm is: whenever we detect a mismatch (after some matches), we already know some of the characters in the text (since they matched the pattern characters prior to the mismatch). We take advantage

of this information to avoid matching the characters that we know will anyway match. KMP algorithm does some preprocessing over the pattern pat[] and constructs an auxiliary array π[] of size m (same as size of pattern). Here π indicates longest proper prefix which is also suffix. For each sub-pattern pat[0…i] where i = 0 to m-1, π[i] stores length of the maximum matching proper prefix which is also a suffix of the sub-pattern pat[0..i]. There are two main components to KMP Algorithm,

Prefix function π:

The prefix function π for a pattern encapsulates knowledge about how the pattern matches against shifts of itself. This enables avoiding backtracking on the String 'S'.

Preprocessing Algorithm:

In the preprocessing part, the values of π[] can be calculated. To do that, the length of the longest prefix suffix value for the previous index should be tracked. The value of π [1] and len is initialized as 1. If pat[len] and pat[i] match, the len value is incremented by 1 and assign the incremented value to π [i]. If pat[i] and pat[len] do not match and len is not 0, then len is updated to π [len-1]. The key to KMP, of course, is the partial match table.

π [i] = the longest proper prefix of pat[0…i] which is also a suffix of pat[0…i]. The calculation of the Prefix function π for the pattern P = abababababca can be explained as follows:

i	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
P[i]	**a**	**b**	**a**	**b**	**a**	**b**	**a**	**b**	**c**	**a**
π[i]	**0**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**0**	**1**

Step 1: When i = 1,

Prefix	sufffix	Match
-	-	0

Step 2: When i = 2,

Prefix	Suffix	Match
a	b	0

Step 3: When i = 3,

Prefix	Sufffix	Match
a	a	1
ab	ba	0

Step 4: When i = 4,

Prefix	Sufffix	Match
a	b	0
ab	ab	1
aba	bab	2

Step 5: When i = 5,

Prefix	Suffix	Match
a	a	1
ab	ba	0
aba	aba	3
abab	baba	0

Step 6: When i = 6,

Prefix	Suffix	Match
a	b	0
ab	ab	2
aba	bab	0
abab	abab	4
ababa	babab	0

Step 7: when i = 7,

Prefix	Suffix	Match
a	a	1
ab	ba	0
aba	aba	3
abab	baba	0
ababa	ababa	5
ababab	bababa	0

Step 8: When i = 8,

Prefix	Suffix	Match
a	b	0
ab	ab	2
aba	bab	0
abab	abab	4
ababa	babab	0
ababab	ababab	6
abababa	bababab	0

Step 9: When i = 9,

Prefix	Suffix	Match
a	c	0
ab	bc	0
aba	abc	0
abab	babc	0
ababa	ababc	0
ababab	bababc	0
abababa	abababc	0
abababac	bababababc	0

Step 10: When i = 10,

Prefix	Suffix	Match
a	a	1
ab	ca	0
aba	bca	0
abab	abca	0
ababa	babca	0
ababab	ababca	0
abababa	bababca	0
abababab	abababca	0
ababababc	babababca	0

The above table demonstrates that $\Pi[i]$ is the length of the longest proper prefix which is also suffix. Some more examples,

For the pattern "AABAACAABAA"

P[i] : |A|A|B|A|A|C|A|A|B|A|A|
i : |1 |2 |3 |4 |5 |6 |7|8 |9 |10|11|
$\Pi[i]$: |0|1 |0 |1 |2 |0 |1 |2 |3 |4 |5 |

For the pattern "ABCDE"

P[i] : |A|B|C|D|E|
i : |1 |2 |3 |4 |5|
$\Pi[i]$: |0|0 |0 |0 |0 |

For the pattern "AAAAA"

P[i] : |A|A|A|A|A|
i : |1 |2 |3 |4 |5|
$\Pi[i]$: |0|1|2 |3 |4 |

For the pattern "AAABAAA",

P[i] : |A|A|A|B|A|A|A|
i : |1 |2 |3 |4 |5 |6 |7|
$\Pi[i]$: |0|1 |2 |0 |1 |2 |3 |

For the pattern "AAACAAAAAC"

P[i] : |A|A|A|C|A|A|A|A|A|C|
i : |1 |2 |3 |4 |5 |6 |7|8 |9 |10|
$\Pi[i]$: |0|1 |2 |0 |1 |2 |3 |3 |3 |4 |

Hence the running time of compute prefix function is O(m). The complexity of the table algorithm is 0(m), where m is the length of pattern.

Searching Algorithm:

Unlike the Naive algorithm where we slide the pattern by one, we use a value from $\Pi[]$ to decide the next sliding position. Let us see how we do that. When we compare pat[j] with S[i] and see a mismatch, we know that characters pat[0..j-1] match

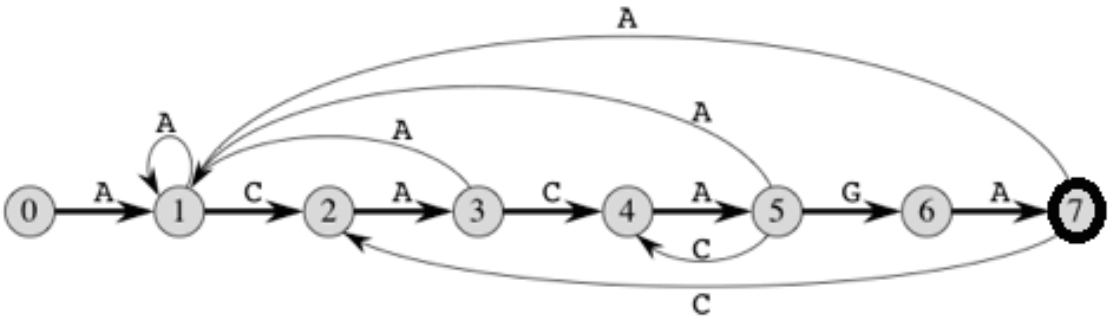
with S[i-j+1…i-1], and we also know that π[j-1] characters of pat[0…j-1] are both proper prefix and suffix which means we do not need to match these π[j-1] characters with S[i-j…i-1] because we know that these characters will anyway match. Thus the running time of the matching function is O(n), which is equal to the length of the string 'S'.

Thus the worst case and average case time complexity of Knuth-Morris-Pratt algorithm is O(n+m). The algorithm never needs to move backwards in the input text T. It makes the algorithm good for processing very large files. It doesn't work so well as the size of the alphabets increases. By which more chances of mismatch occurs. Thus the algorithm performs better when the alphabet size is small.

3.3. Finite Automata Algorithm

In Finite Automata based algorithm, we preprocess the pattern and build a 2D array that represents a Finite Automata. Construction of the FA is the main tricky part of this algorithm. Once the FA is built, the searching is simple. In search, we simply need to start from the first state of the automata and first character of the text. At every step, we consider next character of text, look for the next state in the built FA and move to new state. If we reach final state, then pattern is found in text. Time complexity of the search process is O(n).

The following figure represents the State-Transition Diagram for the String-matching automaton that accepts all strings ending in the string ACACAGA.



State 0 is the start state, and state 7 is the only accepting state. A directed edge from state i to state j labeled a represents δ(i,a) = j. The right-going edges forming the "spine" of the automaton, shown heavy in the figure, correspond to successful matches between pattern and input characters. The left-going edges correspond to failing matches. Some edges corresponding to failing matches are not shown; by convention, if a state i has no outgoing edge labeled a for some a $\in \Sigma$, then δ(i,a) = 0.

The corresponding transition function δ for the pattern string P = ACACAGA is given below. The entries corresponding to successful matches between pattern and input characters are shown in red color.

State	A	C	G
0	1	0	0
1	1	2	0
2	3	0	0
3	1	4	0
4	5	0	0
5	1	4	6
6	7	0	0
7	1	2	0

The following diagram shows the operation of the automaton on the text T = ACACACAGAAC. Under each text character T[i] is given the state $ø(T_i)$ the automaton is in after processing the prefix T_i. One occurrence of the pattern is found, ending in position 9.

i	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*
T[i]	*A*	*C*	*A*	*C*	*A*	*C*	*A*	*G*	*A*	*A*	*C*
State: ø(T_i)	*1*	*2*	*3*	*4*	*5*	*4*	*5*	*6*	*7*	*1*	*2*

Number of states in Finite Automata will be M+1 where M is length of the pattern. The main thing to construct FA is to get the next state from the current state for every possible character. Given a character x and a state k, we can get the next state by considering the string "pat[0..k-1]x" which is basically concatenation of pattern characters pat[0], pat[1] … pat[k-1] and the character x. The idea is to get length of the longest prefix of the given pattern such that the prefix is also suffix of "pat[0..k-1]x". The value of length gives us the next state. For example, let us see how to get the next state from current state 5 and character 'C' in the above diagram. We need to consider the string, "pat[0..5]C" which is "ACACAC". The length of the longest prefix of the pattern such that the prefix is suffix of "ACACAC" is 4 ("ACAC"). So the next state (from state 5) is 4 for character 'C'.

The time complexity of the finite automata construction algorithm is $O(m^{3*NO\_OF\_CHARS})$ where m is length of the pattern and NO_OF_CHARS is size of alphabet (total number of possible characters in pattern and text .

The time complexity of the searching algorithm is O(n). Thus the total running time to find all occurrences of a length-m pattern in a length-n text over an alphabet Σ is O(n + m).

3.4. Rabin Karp Algorithm

The Rabin–Karp algorithm or Karp–Rabin algorithm is a string searching algorithm that uses hashing to find any one of a set of pattern strings in a text. Like the Naive Algorithm, Rabin-Karp algorithm also slides the pattern one by one. But unlike the Naive algorithm, Rabin Karp algorithm matches the hash value of the pattern with the

hash value of current substring of text, and if the hash values match then only it starts matching individual characters. So Rabin Karp algorithm needs to calculate hash values for following strings.

- Pattern itself.
- All the substrings of text of length m.

A hash function is a function which converts every string into a numeric value, called its hash value. The algorithm exploits the fact that if two strings are equal, their hash values are also equal. However, there are two problems with this. First, because there are so many different strings, to keep the hash values small we have to assign some strings the same number. This means that if the hash values match, the strings might not match; we have to verify that they do, which can take a long time for long substrings.

The naive recomputation of the hash value for the substring s[i+1..i+m], would require O(m) time, and since this is done on each loop, the algorithm would require O(mn) time, the same as the most naive algorithms. The trick to solve this is to note that the variable hs already contains the hash value of s[i..i+m-1]. This variable is used in the computation next hash value in constant time. The rolling hash function is used in this recomputation. A rolling hash is a hash function specially designed to enable this operation. One simple example is adding up the values of each character in the substring. The following rule gives the formula to compute the next hash value in constant time:

$$s[i+1..i+m] = s[i..i+m-1] - s[i] + s[i+m]$$

This rolling hash function treats every substring as a number in some base, the base being usually a large prime. For example, if the substring is "hi" and the base is 101, the hash value would be $104 \times 101^1 + 105 \times 101^0 = 10609$ (ASCII of 'h' is 104 and of 'i' is 105).

For example, if the text is "abracadabra" and the pattern being searched is of length 3, the hash of the first substring, "abr", using 101 as base is:

ASCII a = 97, b = 98, r = 114

$$\text{Hash}(\text{“abr”}) = (97 \times 101^2) + (98 \times 101^1) + (114 \times 101^0) = 999{,}509$$

The hash of the next substring, "bra", can be computed from the hash of "abr" by subtracting the number added for the first 'a' of "abr", i.e. 97×101^2, multiplying by the base and adding for the last a of "bra", i.e. 97×101^0. Like so:

$$\text{Hash}(\text{“bra”}) = [101 \times (999{,}509 - (97 \times 101^2))] + (97 \times 101^0) = 1{,}011{,}309$$

If the substrings in question are long, this algorithm achieves great savings compared with many other hashing schemes.

Theoretically, there exist other algorithms that could provide convenient recomputation, e.g. multiplying together ASCII values of all characters so that shifting substring would only entail dividing by the first character and multiplying by the last. The limitation, however, is the limited size of the integer data type and the necessity of using modular arithmetic to scale down the hash results. Meanwhile, naive hash functions

do not produce large numbers quickly, but, just like adding ASCII values, are likely to cause many hash collisions and hence slow down the algorithm.

The average and best case running time of the Rabin-Karp algorithm is O(n+m), but its worst-case time is O(nm). Worst case of Rabin-Karp algorithm occurs when all characters of pattern and text are same as the hash values of all the substrings of txt[] match with hash value of pat[]. For example,

pat[] = "AAA" and txt[] = "AAAAAAA".

3.5. Boyer Moore Algorithm

Like KMP and Finite Automata algorithms, Boyer Moore algorithm also preprocesses the pattern. Boyer Moore is a combination of following two approaches.

- Bad Character Heuristic
- Good Suffix Heuristic

Both of the above heuristics can also be used independently to search a pattern in a text. The two independent approaches can be worked together in the Boyer Moore algorithm. The Naive algorithm slides the pattern over the text one by one. KMP algorithm does preprocessing over the pattern so that the pattern can be shifted by more than one. The Boyer Moore algorithm does preprocessing for the same reason. It preprocesses the pattern and creates different arrays for both heuristics. At every step, it slides the pattern by max of the slides suggested by the two heuristics. So it uses best of the two heuristics at every step. Unlike the previous pattern searching algorithms, Boyer Moore algorithm starts matching from the last character of the pattern.

The idea of bad character heuristic is simple. The character of the text, which doesn't match, with the current character of pattern is called the Bad Character. Whenever a character doesn't match, we slide the pattern in such a way that aligns the bad character with the last occurrence of it in pattern. We preprocess the pattern and store the last occurrence of every possible character in an array of size equal to alphabet size. The table built using this technique is called as bad match table. Example of Bad match table:

Patter = TOOTH
Value = length-index-1

Letter	T	O	H	*
value	1	2	5	5

The bad-character rule considers the character in T at which the comparison process failed (assuming such a failure occurred). The next occurrence of that character to the left in P is found, and a shift which brings that occurrence in line with the mismatched occurrence in T is proposed. If the mismatched character does not occur to the left in P, a shift is proposed that moves the entirety of P past the point of mismatch.

If the character is not present at all, then it may result in a shift by m (length of pattern). Therefore, the bad character heuristic takes O(n/m) time in the best case.

The Bad Character Heuristic may take O(mn) time in worst case. The worst case occurs when all characters of the text and pattern are same. For example,

txt[] = "AAAAAAAAAAAAAAAAA" & pat[] = "AAAAA"

3.6. Algorithm analysis table

Algorithm Name	Comparison Order	Preprocess Time Complexity	Search Time Complexity	Main Characteristic
Brute Force	Not relevant	No preprocessing	O(mn)	Use one by one character shift. Not an optimal one
KMP	Left to Right	O(m)	O(m+n)	Independent of alphabet size, use the notion of border of the string, increases performance, decrease delay and decrease time of comparing.
Finite Automata	Left to Right	O(m)	O(m+n)	Preprocess the pattern and build finite automata
Rabin Karp	Left to Right	O(m)	O(mn)	Use hashing function, very effective for multiple pattern matching in 1D matching.
Boyer Moore	Right to Left	O(m+n)	O(mn)	Use both good suffix shift and bad character shift

4. In-depth research of the problem

4.1. Advantages of preprocessing text

All of the above algorithms preprocess the pattern to make the pattern searching faster. The best time complexity that we could get by preprocessing pattern is O(n) where n is length of the text. In this post, we will discuss an approach that preprocesses the text. A suffix tree is built of the text. After preprocessing text (building suffix tree of text), we can search any pattern in O(m) time where m is length of the pattern. Preprocessing of text may become costly if the text changes frequently. It is good for fixed text or less frequently changing text though.

4.2. Suffix Tree

Why Suffix Tree? Suffix trees are phenomenally useful data structures for solving string problems elegantly and efficiently. If you need to speed up a string processing algorithm from $O(n^2)$ to linear time, proper use of suffix trees is quite likely the answer. In its simplest instantiation, a suffix tree is simply a trie of the n strings that are suffixes of an n-character string S. A trie is a tree structure, where each node represents one character, and the root represents the null string. Thus each path from the root represents a string, described by the characters labeling the nodes traversed. Any finite set of words defines a trie, and two words with common prefixes will branch off from each other at the first distinguishing character. Each leaf represents the end of a string. Tries are useful for testing whether a given query string q is in the set.

4.2.1. Construction

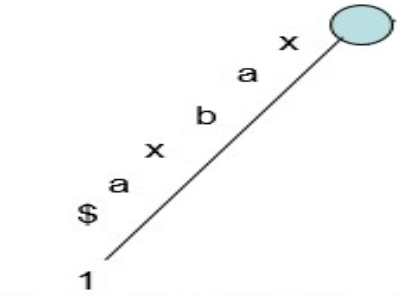
Let S denote a string, the length of which is n. Let S[i,j] denote the substring of S from position i to position j. Before constructing the suffix tree, we concatenate a new character, $ to S. The importance of this character is twofold. First, by adding it to the string, one can avoid that a suffix is a prefix of another suffix, which is undesirable. Second, the generalization is also made easier by this operation. Now, we will define the suffix tree of a string. We always consider a fixed size alphabet. A suffix tree is a rooted, directed tree. It has n leaves labeled from 1 to n, and its edges are labeled by characters of the alphabet. The label of an edge e is denoted by l(e). On a path from the root to the leaf j one can read the suffix S[j,n] of the string and a $ sign. Such a path is denoted by P(j) and its label is referred as the path label of j and denoted by L(j). We call a leaf w reachable from the node v, if there is a directed path from v to w.

Suffix trees can use quite a lot of space. There are long branches, which could be compressed. By compressing long branches one will receive the compact suffix tree of the string. Formally, a compact suffix tree of S is a rooted directed tree with n leaves. Each internal node has at least two children (the root is not an internal node). Each edge has a label with the property that if (u, v) and (u, w) are edges, then the first characters of the label of (u, v) and of (u, w) are distinct. The label of a path is the concatenation of the labels on its edges. A

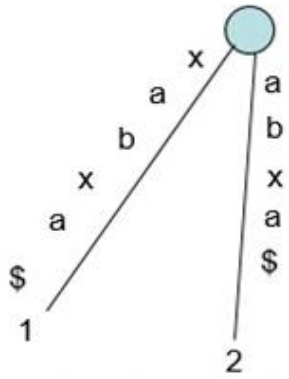
4.2.2. A naive algorithm for constructing a suffix tree

We will give a naive recursive algorithm for building the suffix tree of a string. T_i stands for the suffix tree built in phase i. As initiation, take a root and add an edge labeled by S[1,n]$. In phase i, T_{i-1} is already built. First, add leaf i to the tree. Take the suffix S[i,n] and find the longest path from the root whose label is its prefix. Suppose this label is S[i,k]. If the end of this label does not end in a node, create a new internal node and add a new edge from this node to leaf i labelled by S[k + 1, n]$. This algorithm takes $O(n^2)$ time.

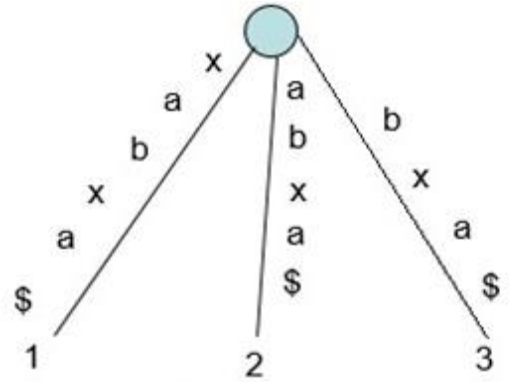
Following are few steps to build suffix tree for the string “xabxa$” based on above algorithm:



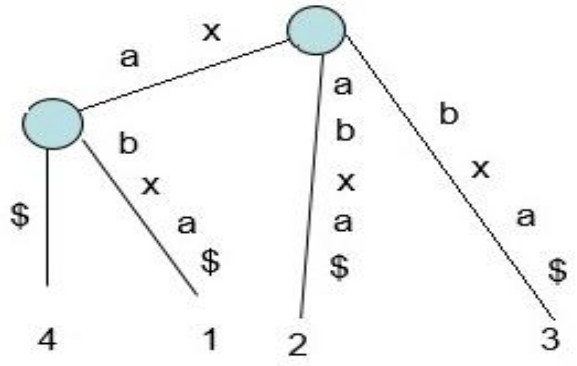
Tree with suffix N1, S[1..6]



Tree with suffixes N1, S[1..6] and N2, S[2..6]



Tree with suffixes N1, N2 and N3



Tree with suffixes N1, N2, N3 and N4

4.2.3. Linear Time Construction of Suffix Tree

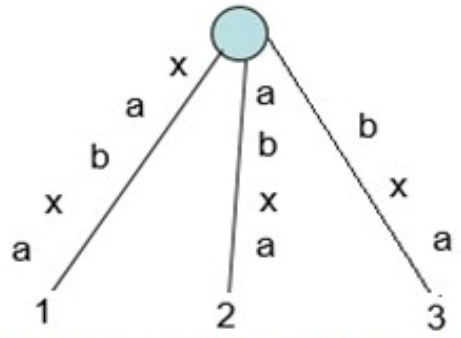
We will present a method for constructing suffix trees in detail, Ukkonen's method. Even though Weiner was the first to show that suffix trees can be built in linear time, however, lJkkonen's method is equally fast and uses far less space (i.e., memory) in practice than Weiner's method - Hence Ukkonen is the method of choice for most problems requiring the construction of a suffix tree.

4.2.4. Implicit suffix tree

Definition: An implicit suffix tree for string S is a tree obtained from the suffix tree for S$ by removing every copy of the terminal symbol $ from the edge labels of the tree, then removing any edge that has no label, and then removing any node that does not have at least two children.

While generating suffix tree using Ukkonen's algorithm, we will see implicit suffix tree in intermediate steps few times depending on characters in string S. In implicit suffix trees, there will be no edge with $ (or # or any other termination character) label and no internal node with only one edge going out of it. To get implicit suffix tree from a suffix tree S$, there are three important step as follows:

- Remove all terminal symbol $ from the edge labels of the tree,
- Remove any edge that has no label
- Remove any node that has only one edge going out of it and merge the edges.



4.2.5. High Level Description of Ukkonen's algorithm

Ukkonen's algorithm constructs an implicit suffix tree T_i for each prefix S[l ..i] of S (of length m). It first builds T_1 using 1st character, then T_2 using 2nd character, then T_3 using 3rd character, T_m using mth character. Implicit suffix tree T_{i+1} is built on top of implicit suffix tree T_i. The true suffix tree for S is built from T_m by adding $. At any time, Ukkonen's algorithm builds the suffix tree for the characters seen so far and so it has on-line property that may be useful in some situations. Time taken is O(m).

Ukkonen's algorithm is divided into m phases (one phase for each character in the string with length m). In phase i+1, tree T_{i+1} is built from tree T_i. Each phase i+1 is further divided into i+1 extensions, one for each of the i+1 suffixes of S[1..i+1]. In extension j of phase i+1, the algorithm first finds the end of the path from the root labeled with substring S[j..i]. It then extends the substring by adding the character S(i+1) to its end (if it is not there already). In extension 1 of phase i+1, we put string S[1..i+1] in the tree. Here S[1..i] will already be present in tree due to previous phase i. We just need to add S[i+1]th character in tree (if not there already). In extension 2 of phase i+1, we put string S[2..i+1] in the tree. Here S[2..i] will already be present in tree due to previous phase i. We just need to add S[i+1]th character in tree (if not there already). In extension 3 of phase i+1, we put string S[3..i+1] in the tree. Here S[3..i] will already be present in tree due to previous phase i. We just need to add S[i+1]th character in tree (if not there already).

In extension i+1 of phase i+1, we put string S[i+1..i+1] in the tree. This is just one character which may not be in tree (if character is seen first time so far). If so, we just add a new leaf edge with label S[i+1].

<u>High Level Ukkonen's algorithm</u>:

```
Construct tree T1
For i from 1 to m-1 do
begin {phase i+1}
        For j from 1 to i+1
                begin {extension j}
                Find the end of the path from the root labeled S[j..i] in the current tree.
                Extend that path by adding character S[i+l] if it is not there already
        end;
end;
```

Suffix extension is all about adding the next character into the suffix tree built so far. In extension j of phase i+1, algorithm finds the end of S[j..i] (which is already in the tree due to previous phase i) and then it extends S[j..i] to be sure the suffix S[j..i+1] is in the tree.

There are 3 suffix extension rules,
Rule 1: If the path from the root labeled S[j..i] ends at leaf edge (i.e. S[i] is last character on leaf edge) then character S[i+1] is just added to the end of the label on that leaf edge.

Rule 2: If the path from the root labeled S[j..i] ends at non-leaf edge (i.e. there are more characters after S[i] on path) and next character is not s[i+1], then a new leaf edge with label s{i+1] and number j is created starting from character S[i+1].
A new internal node will also be created if s[1..i] ends inside (in-between) a non-leaf edge.

Rule 3: If the path from the root labeled S[j..i] ends at non-leaf edge (i.e. there are more characters after S[i] on path) and next character is s[i+1] (already in tree), do nothing.

The key point to note here is that from a given node (root or internal), there will be one and only one edge starting from one character. There will not be more than one edges going out of any node, starting with same character.

Following is a step by step suffix tree construction of string xabxac using Ukkonen's algorithm:

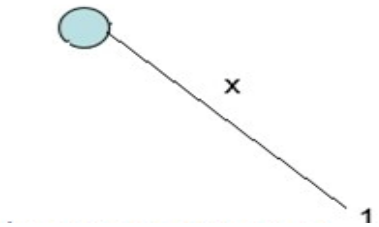


Figure 1: T1 for S[1..1]
Adding suffixes of X(X)
Rule 2 – A new leaf edge

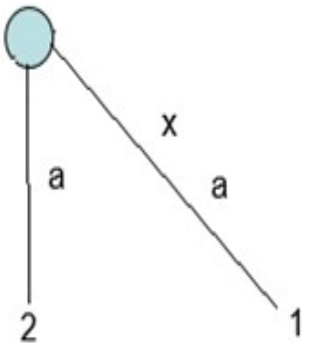


Figure 2: T2 for S[1..2]
Adding suffixes of xa (xa and a)
Rule 1 – Extending path label in existing leaf edge
Rule 2 - A new leaf edge

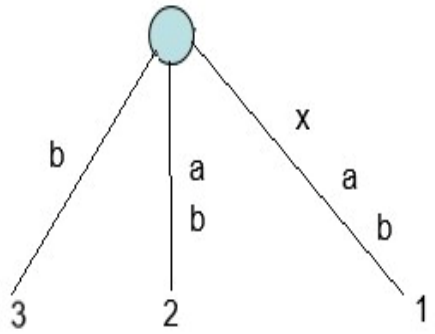


Figure 3: T3 for S[1..3]
Adding suffixes of xab (xab, ab and b)
Rule 1 – Extending path label in existing leaf edge
Rule 2 – A new leaf edge

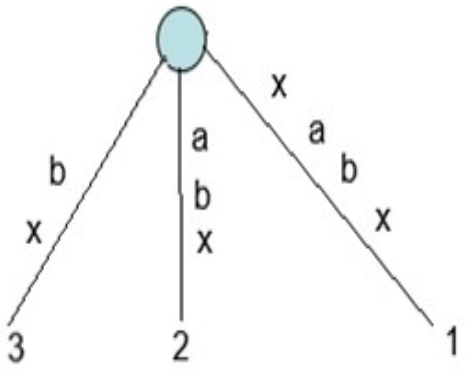


Figure 4: T4 for S[1..4]
Adding suffixes of xabx (xabx, abx, bx and x)
Rule 1 – Extending path label in existing leaf edge
Rule 3: Do nothing (path with label x already present)

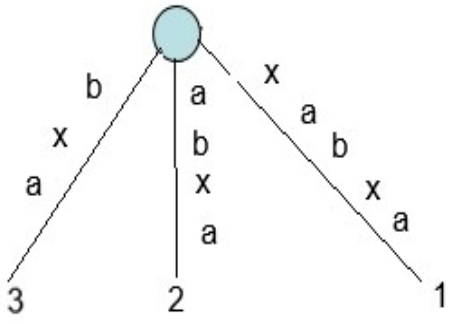


Figure 5: T5 for S[1..5]
Adding suffixes of xabxa (xabxa, abxa, bxa, xa and x)
Rule 1 – Extending path label in existing leaf edge
Rule 3 - Do nothing (path with label xa and a already present)

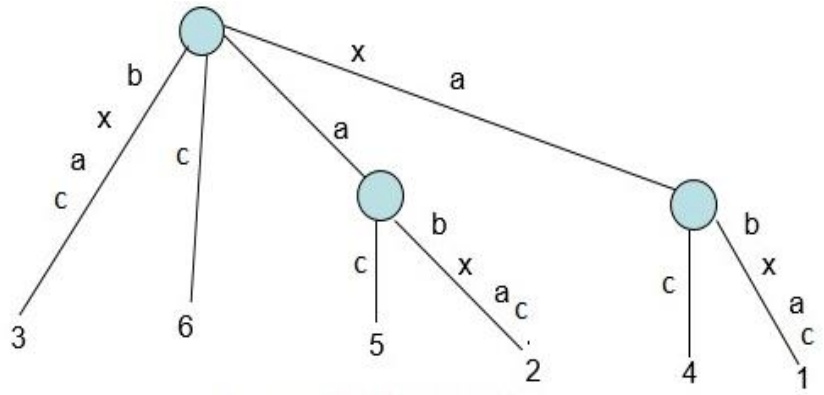


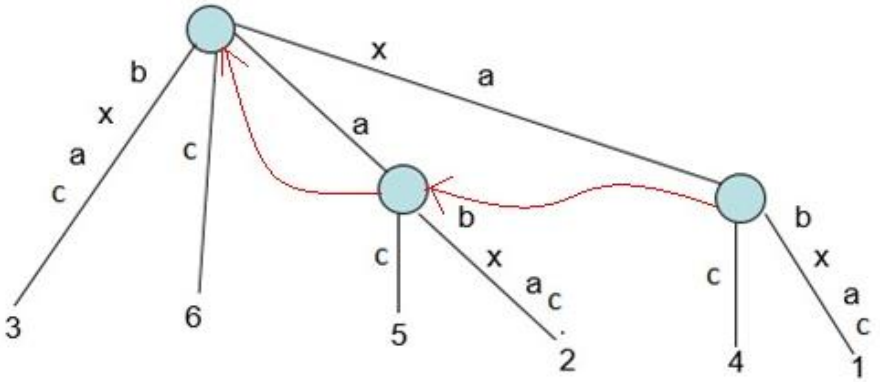
Figure 6: T6 for S[1..6]
Adding suffixes of xabxac (xabxac, abxac, bxac, xac, ac, c)
Rule 1 – Extending path label in existing leaf edge
Rule 2 – Three new leaf edges and two new internal nodes

In Suffix Tree Construction of string S of length m, there are m phases and for a phase j (1 <= j <= m), we add j^{th} character in tree built so far and this is done through j extensions. All extensions follow one of the three extension rules. To do j^{th} extension of phase i+1 (adding character S[i+1]), we first need to find end of the path from the root labeled S[j..i] in the current tree. One way is start from root and traverse the edges matching S[j..i] string. This will take $O(m^3)$ time to build the suffix tree. Using few observations and implementation tricks, it can be done in O(m) which is explained as follows.

4.3. Suffix links: First implementation Speedup

4.3.1. Definition and Construction

One important feature of suffix trees are suffix links. Suffix links are well defined for suffix trees. For an internal node v with path-label xA, where x denotes a single character and A denotes a (possibly empty) substring, if there is another node s(v) with path-label A, then a pointer from v to s(v) is called a suffix link.
If A is empty string, suffix link from internal node will go to root node. There will not be any suffix link from root node.



Suffix Links in red arrows

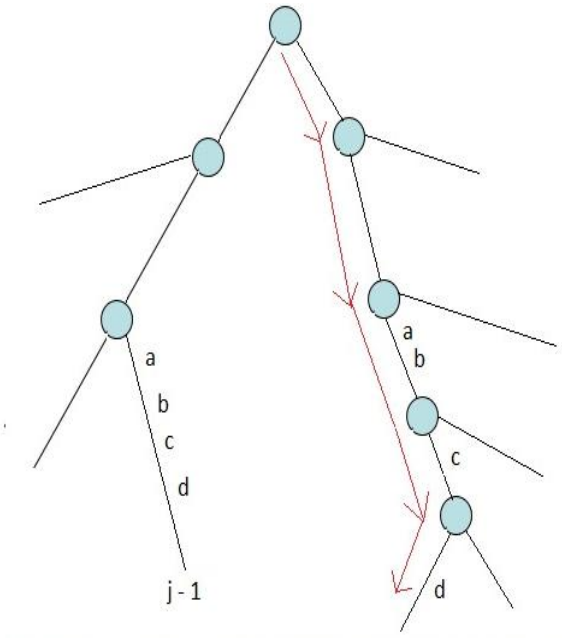
In extension j of some phase i, if a new internal node v with path-label xA is added, then in extension j+1 in the same phase i:

- Either the path labeled A already ends at an internal node (or root node if A is empty)
- OR a new internal node at the end of string A will be created.

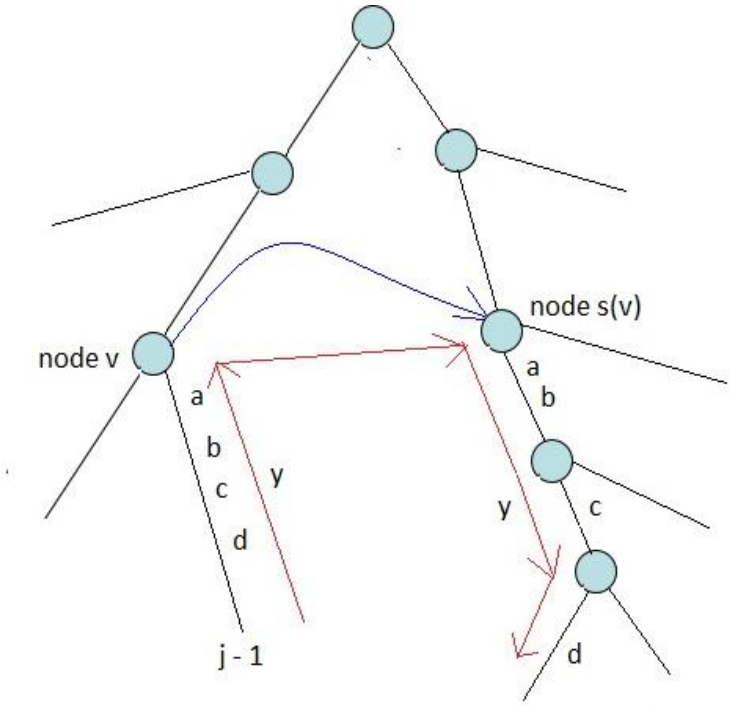
In extension j+1 of same phase i, we will create a suffix link from the internal node created in j^{th}extension to the node with path labeled A. So in a given phase, any newly created internal node (with path-label xA) will have a suffix link from it (pointing to another node with path-label A) by the end of the next extension. In any implicit suffix tree T_i after phase i, if internal node v has path-label xA, then there is a node s(v) in T_i with path-label A and node v will point to node s(v) using suffix link. At any time, all internal nodes in the changing tree will have suffix links from them to another internal node (or root) except for the most recently added internal node, which will receive its suffix link by the end of the next extension.

4.3.2. Usage of suffix Links to speed up the implementation

In extension j of phase i+1, we need to find the end of the path from the root labeled S[j..i] in the current tree. One way is start from root and traverse the edges matching S[j..i] string. Suffix links provide a short cut to find end of the path.



Traversal from root to leaf in extension j of phase i+1, to find end of S[j..i], when suffix is not used

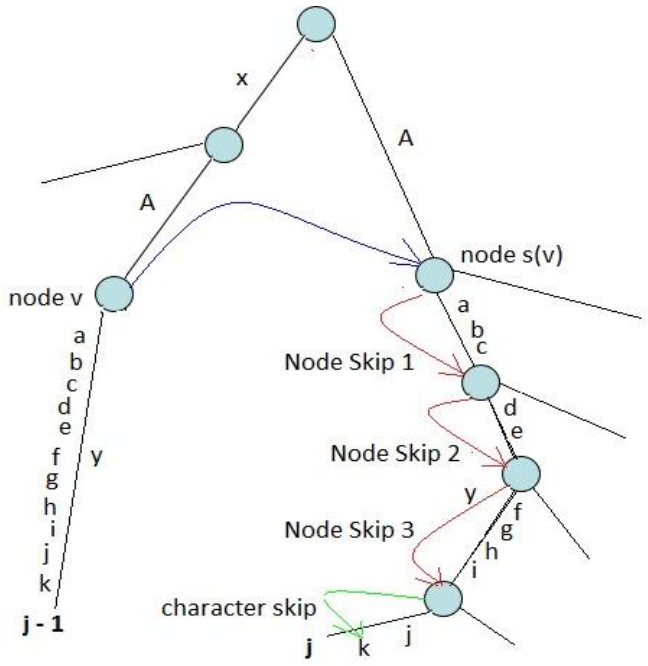


Traversal from root to leaf in extension j of phase i+1, to find end of S[j..i], when suffix link (blue arrow) is used

So we can see that, to find end of path S[j..i], we need not traverse from root. We can start from the end of path S[j-1..i], walk up one edge to node v (i.e. go to parent node), follow the suffix link to s(v), then walk down the path y . This shows the use of suffix link is an improvement over the process. When there is a suffix link from node v to node s(v), then if there is a path labeled with string y from node v to a leaf, then there must be a path labeled with string y from node s(v) to a leaf. In the above figure, there is a path label "abcd" from node v to a leaf, then there is a path will same label "abcd" from node s(v) to a leaf. This fact can be used to improve the walk from s(v) to leaf along the path y. This is called "skip/count" trick that will reduce the worst-case time for the algorithm to $0(m^2)$.

4.3.3. Skip / Count Trick

When walking down from node s(v) to leaf, instead of matching path character by character as we travel, we can directly skip to the next node if number of characters on the edge is less than the number of characters we need to travel. If number of characters on the edge is more than the number of characters we need to travel, we directly skip to the last character on that edge. If implementation is such a way that number of characters on any edge, character at a given position in string S should be obtained in constant time, then skip/count trick will do the walk down in proportional to the number of nodes on it rather than the number of characters on it.



Skip/Count trick: Substring y from node v has length 11.Substring y fom node s(v) is two characters down the last node, after3 node skips

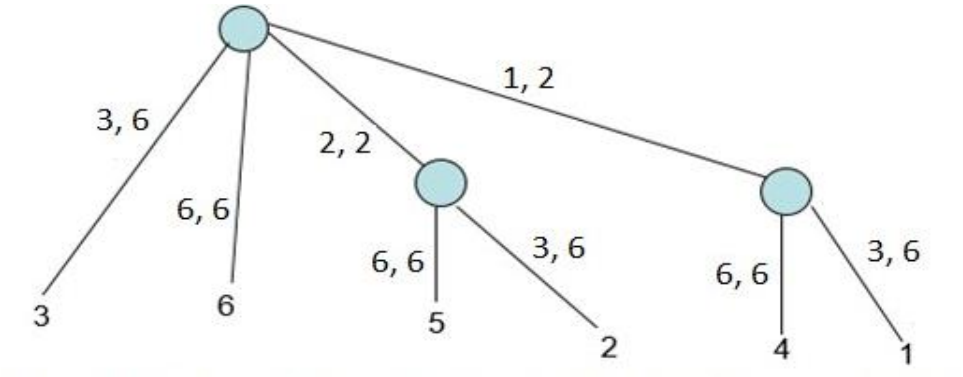
Analysis:

Using suffix link along with skip/count trick, suffix tree can be built in $O(m^2)$ as there are m phases and each phase takes O(m).

4.4. Edge-label compression

We next establish a 0(m) time bound for building a suffix tree. There is, however, one immediate barrier to that goal: The suffix tree may require $\Theta(m^2)$ space. As described so far, the edge-labels of a suffix tree might contain more than $\Theta(m)$ characters in total. Since the time for the algorithm is at least as large as the size of its output, that many characters makes an 0(m) time bound impossible. Thus, an 0(m)-time algorithm for building suffix trees requires some alternate scheme to represent the edge-labels.

So far, path labels are represented as characters in string. Such a suffix tree will take $O(m^2)$ space to store the path labels. To avoid this, we can use two pair of indices (start, end) on each edge for path labels, instead of substring itself. The indices start and end tells the path label start and end position in string S. By using an index pair to specify an edge-label, only two numbers are written on any edge, and since the number of edges is at most 2m - 1, the suffix tree uses only 0(m) symbols and requires only 0(m) space. This makes it more plausible that the tree can actually be built in 0(m) time. Although the fully implemented algorithm will not explicitly write a substring on an edge, we will still find it convenient to talk about "the substring or label on an edge or path" as if the explicit substring was written there.



Suffix tree for string xabxac with edge-label compression
Figure 7: shows the same suffix tree without edge-label compression

4.5. More Tricks

There are two observations about the way extension rules interact in successive extensions and phases. These two observations lead to two more implementation tricks.

Observation 1: Rule 3 is show stopper

In a phase i, there are i extensions (1 to i) to be done. When rule 3 applies in any extension j of phase i+1 (i.e. path labeled S[j..i] continues with character S[i+1]), then it will also apply in all further extensions of same phase (i.e. extensions j+1 to i+1 in phase i+1). That's because if path labeled S[j..i] continues with character S[i+1], then path labeled S[j+1..i], S[j+2..i], S[j+3..i],…, S[i..i] will also continue with character S[i+1]. Consider Figure 3, Figure 4 and Figure 5, where Rule 3 is applied. In Figure 3, "xab" is added in tree and in Figure 4 (Phase 4), we add next character "x". In this, 3 extensions are done (which adds 3 suffixes). Last suffix "x" is already present in tree. In Figure 5, we add character "a" in tree (Phase 5). First 3 suffixes are added in tree and last two suffixes "xa" and "a" are already present in tree. This shows that if suffix S[j..i] present in tree, then ALL the remaining suffixes S[j+1..i], S[j+2..i], S[j+3..i],…, S[i..i] will also be there in tree and no work needed to add those remaining suffixes. So no more work needed to be done in any phase as soon as rule 3 applies in any extension in that phase. If a new internal node v gets created in extension j and rule 3 applies in next extension j+1, then we need to add suffix link from node v to current node (if we are on internal node) or root node.

Trick 2

Stop the processing of any phase as soon as rule 3 applies. All further extensions are already present in tree implicitly. Trick 2 is clearly a good heuristic to reduce work, but it's not clear if it leads to a better worst-case time bound. For that we need one more observation and trick.

Observation 2: Once a leaf, always a leaf

Once a leaf is created and labeled j (for suffix starting at position j in string S), then this leaf will always be a leaf in successive phases and extensions. Once a leaf is labeled as j, extension rule 1 will always apply to extension j in all successive phases.

Consider Figure 1 to Figure 6:

- In Figure 2 (Phase 2), Rule 1 is applied on leaf labeled 1. After this, in all successive phases, rule 1 is always applied on this leaf.
- In Figure 3 (Phase 3), Rule 1 is applied on leaf labeled 2. After this, in all successive phases, rule 1 is always applied on this leaf.
- In Figure 4 (Phase 4), Rule 1 is applied on leaf labeled 3. After this, in all successive phases, rule 1 is always applied on this leaf.

In any phase i, there is an initial sequence of consecutive extensions where rule 1 or rule 2 are applied and then as soon as rule 3 is applied, phase i ends. Also rule 2 creates a new leaf always (and internal node sometimes). If J_i represents the last extension in phase i when rule 1 or 2 was applied (i.e after i^{th} phase, there will be J_i leaves labelled 1, 2, 3, …, J_i) , then $J_i <= J_{i+1}$. J_i will be equal to J_{i+1} when there are no new leaf created in phase i+1 (i.e rule 3 is applied in J_{i+1}extension).

- In Figure 3 (Phase 3), Rule 1 is applied in 1st two extensions and Rule 2 is applied in 3rd extension, so here $J_3 = 3$
- In Figure 4 (Phase 4), no new leaf created (Rule 1 is applied in 1st 3 extensions and then rule 3 is applied in 4th extension which ends the phase). Here $J_4 = 3 = J_3$
- In Figure 5 (Phase 5), no new leaf created (Rule 1 is applied in 1st 3 extensions and then rule 3 is applied in 4th extension which ends the phase). Here $J_5 = 3 = J_4$ J_i will be less than J_{i+1} when few new leaves are created in phase i+1.
- In Figure 6 (Phase 6), new leaf created (Rule 1 is applied in 1st 3 extensions and then rule 2 is applied in last 3 extension which ends the phase). Here $J_6 = 6 > J_5$

So we can see that in phase i+1, only rule 1 will apply in extensions 1 to J_i (which really doesn't need much work, can be done in constant time and that's the trick 3), extension J_{i+1} onwards, rule 2 may apply to zero or more extensions and then finally rule 3, which ends the phase. Now edge labels are represented using two indices (start, end), for any leaf edge, end will always be equal to phase number i.e. for phase i, end = i for leaf edges, for phase i+1, end = i+1 for leaf edges.

Trick 3

In any phase i, leaf edges may look like (p, i), (q, i), (r, i), …. where p, q, r are starting position of different edges and i is end position of all. Then in phase i+1, these leaf edges will look like (p, i+1), (q, i+1), (r, i+1),…., so on. This way, in each phase, end position has to be incremented in all leaf edges. For this, we need to traverse through all leaf edges and increment end position for them. To do same thing in constant time, maintain a global index e and e will be equal to phase number. So now leaf edges will look like (p, e), (q, e), (r, e).. In any phase, just increment e and extension on all leaf edges will be done.

So using suffix links and tricks 1, 2 and 3, a suffix tree can be built in linear time. Tree T_m could be implicit tree if a suffix is prefix of another. So we can add a $ terminal

symbol first and then run algorithm to get a true suffix tree (A true suffix tree contains all suffixes explicitly). To label each leaf with corresponding suffix starting position (all leaves are labeled as global index e), a linear time traversal can be done on tree. At this point, we have gone through most of the things we needed to know to create suffix tree using Ukkonen's algorithm.

4.6. Creating the true Suffix Tree

The final implicit suffix tree can be converted to a true suffix tree in 0(m) time. First, add a string terminal symbol $ to the end of S and let Ukkonen's algorithm continue with this character. The effect is that no suffix is now a prefix of any other suffix, so the execution of Ukkonen's algorithm results in an implicit suffix tree in which each suffix ends at a leaf and so is explicitly represented. The only other change needed is to replace each index e on every leaf edge with the number in. This is achieved by a 0(m)-time traversal of the tree, visiting each leaf edge. When these modifications have been- made, the resulting tree is a true suffix tree. In summary, Ukkonen's algorithm builds a true suffix tree for S, along with all its suffix links in 0(m) time.

4.7. Improvements over Suffix Tree

4.7.1. Suffix Array

The advantages of suffix arrays over suffix tree include improved space requirements, simpler linear time construction algorithms and improved cache locality. A suffix array is a sorted array of all suffixes of a given string. The definition is similar to Suffix Tree, which is compressed trie of all suffixes of the given text. Any suffix tree based algorithm can be replaced with an algorithm that uses a suffix array enhanced with additional information and solves the same problem in the same time complexity. A suffix array can be constructed from Suffix tree by doing a DFS traversal of the suffix tree. In fact Suffix array and suffix tree both can be constructed from each other in linear time.

Example:
Let the given string be "banana".

0	Banana		5	a
1	anana	Sort the Suffixes	3	ana
2	nana	---------- >	1	anana
3	ana	alphabetically	0	banana
4	na		4	na
5	a		2	nana

So the suffix array for "banana" is {5, 3, 1, 0, 4, 2}

The time complexity of naive method to build suffix array is O (n^2Logn) if we consider a O(nLogn) algorithm used for sorting. The sorting step itself takes O (n^2Logn) time as every comparison is a comparison of two strings and the comparison takes O (n) time. There are also many efficient algorithms available to build suffix array.

Search a pattern using the built Suffix Array:

To search a pattern in a text, we preprocess the text and build a suffix array of the text. Since we have a sorted array of all suffixes, Binary Search can be used to search. There are more efficient algorithms to search pattern once the suffix array is built.

4.7.2. Generalized Suffix Tree

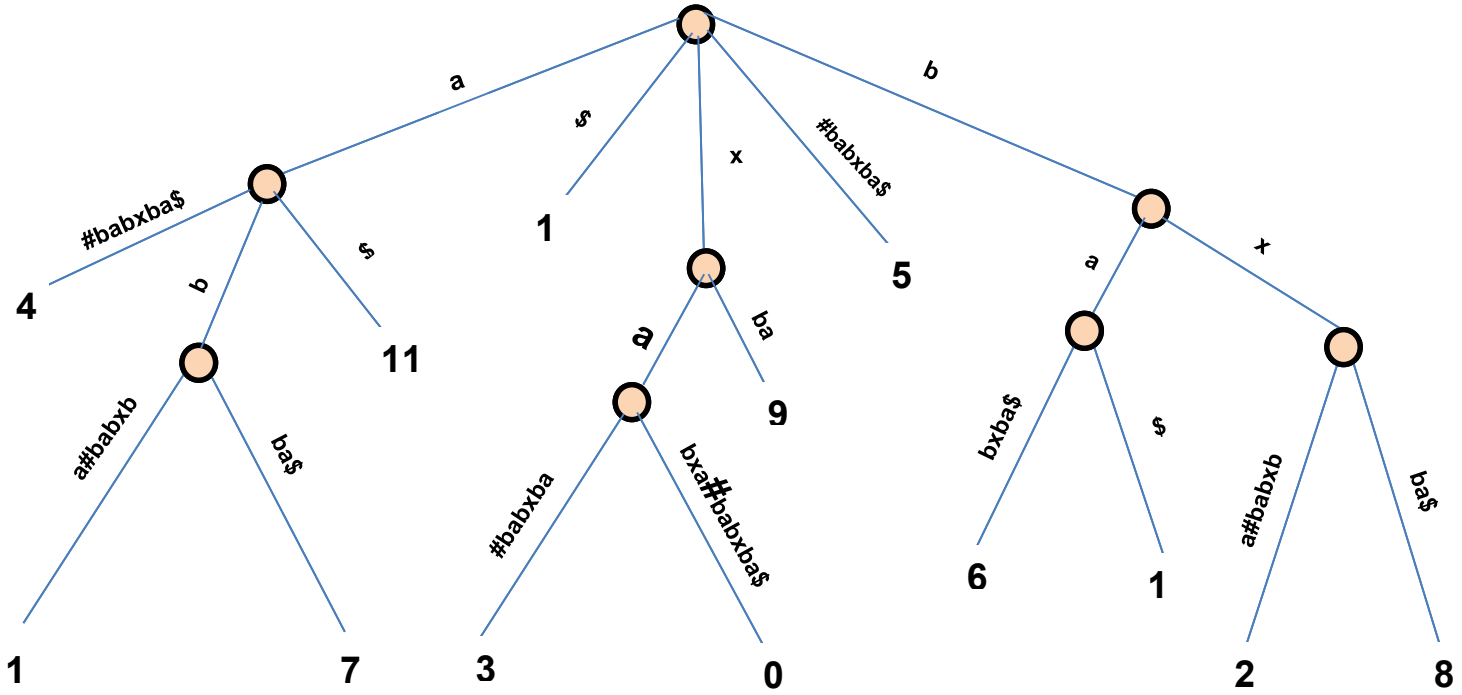
So far, we created suffix tree for one string and then we queried that tree for substring check, searching all patterns, longest repeated substring and built suffix array. There are lots of other problems where multiple strings are involved.

Example: pattern searching in a text file or dictionary, spell checker, phone book, Autocomplete, Longest common substring problem, Longest palindromic substring and More. For such operations, all the involved strings need to be indexed for faster search and retrieval. One way to do this is using suffix trie or suffix tree. A suffix tree made of a set of strings is known as Generalized Suffix Tree. We will discuss a simple way to build Generalized Suffix Tree here for two strings only.

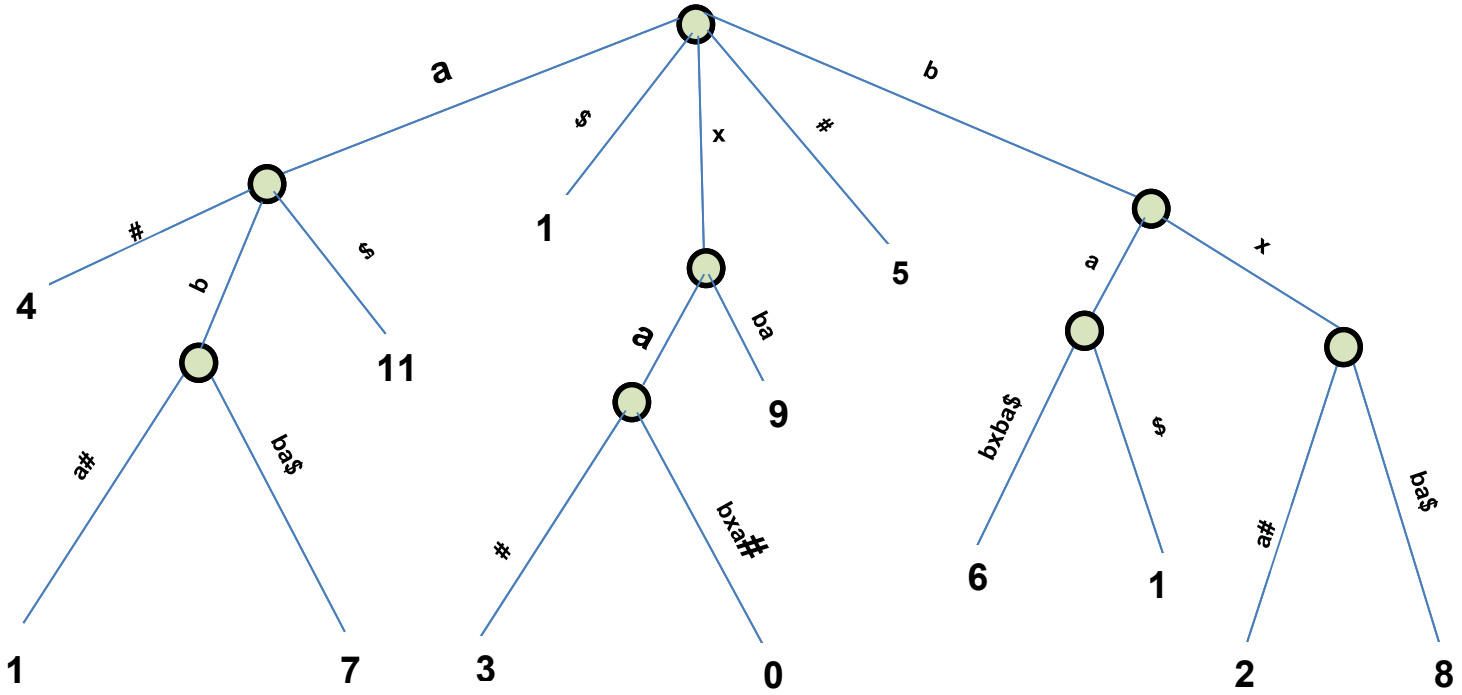
Let's consider two strings X and Y for which we want to build generalized suffix tree. For this we will make a new string X#Y$ where # and $ both are terminal symbols (must be unique). Then we will build suffix tree for X#Y$ which will be the generalized suffix tree for X and Y. Same logic will apply for more than two strings (i.e. concatenate all strings using unique terminal symbols and then build suffix tree for concatenated string).

Let's say,
X = xabxa, and Y = babxba, then
X#Y$ = xabxa#babxba$

The resulting generalized suffix tree for the above string:



We can use this tree to solve some of the problems, but we can refine it a bit by removing unwanted substrings on a path label. A path label should have substring from only one input string, so if there are path labels having substrings from multiple input strings, we can keep only the initial portion corresponding to one string and remove all the later portion. For example, for path labels #babxba$, a#babxba$ and bxa#babxba$, we can remove babxba$ (belongs to second input string) and then new path labels will be #, a# and bxa# respectively. With this change, above diagram will look like below:



Here we are removing unwanted characters on path labels. If a path label has "#" character in it, then we are trimming all characters after "#" in that path label. If two strings are of size M and N, this implementation will take O(M+N) time and space. If input strings are not concatenated already, then it will take 2(M+N) space in total, M+N space to store the generalized suffix tree and another M+N space to store concatenated string.

4.8. Applications of Suffix Tree

Suffix Tree can be used either as exact string matching algorithms or approximate string matching algorithms.

4.8.1. Exact matching

As it was already mentioned, suffix trees are useful when solving the exact matching problem. With a suffix tree, the solution of this problem is very easy (i.e. in linear time in the size of the string). Let n and m denote the length of the string S and of the pattern P, respectively.

First of all, observe that the pattern occurs in the string if it is the prefix of a suffix of the string. Therefore, taking the suffix tree of the string, we should search the first character of the pattern among the edges from the root. If this character does not appear, then the pattern does not occur in the string. If the first k characters of the pattern are found, then the edge labeled with the character k + 1 of the pattern should be chosen. If there is no such edge, the pattern does not occur in the string. If all characters were found in the tree in the appropriate order, an occurrence of the pattern is found and the algorithm stops.

The position of an occurrence is the first character of the pattern in the string. Suppose that the algorithm above found an occurrence of the pattern and stopped at node v. Now, one would like to have all the positions of the occurrences of the pattern. Determine the leaves, which are reachable from v on a directed path. Let the indices of these leaves be contained in the set J. If $j \in J$, then P is a prefix of the suffix S[j,n], so the pattern occurs at the position j. If the pattern is the prefix of the suffix S[j,n], then j will be obviously in J. Now, we found all the occurrences of the pattern, thus we solved the exact matching problem.

One can easily generalize the exact matching for more strings. First, the generalized suffix tree has to be built for all the strings, and then the algorithm above can be repeated. Now, the leaves have two labels. The pattern occurs in string i at position j if and only if the algorithm stops at an internal node from which the leave labeled by (i,j) is reachable on a directed path.

Complexity of the algorithm:

Now, we will compare suffix trees and other linear-time algorithms (Knuth-Morris- Pratt, Boyer-Moore) for exact matching. The linear-time algorithms take O(n) + O(m) time for each matching problem. If we search k patterns in the k same string, the running time is kO(n) + $\sum_{i=1}^{k} \boldsymbol{O}(\boldsymbol{m_i})$, where m_i is the length of the i=1 pattern i. Constructing the suffix tree takes O(n) time with Ukkonen's algorithm. Finding an instance of P in S takes O(m) time. If we search k patterns in the same string,

The running time is O(n) + $\sum_{i=1}^{k} \boldsymbol{O}(\boldsymbol{m_i})$

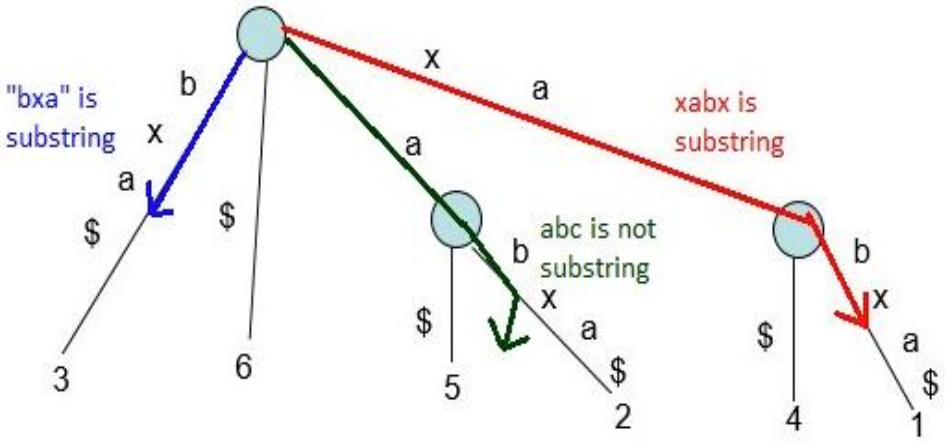
For single searches, linear-time algorithms perform better, but if more patterns are to be found, the algorithm using suffix trees is faster.

4.8.2. Inexact matching

In molecular biology, the k-mismatch problem is a central topic. The nature of the genetic code allows some mismatches in a DNA sequence. The properties of two protein molecules can be similar even if there are several differences between their amino acid sequences. In several experiments RNA sequences have to be planned such that there are at least k mismatches between them. Formally, the k-mismatch problem can be defined as it follows: given a pattern, a string and a number k, find those all substrings of the string matching the pattern with not more than k mismatches. For finding all k-mismatches of a pattern, the fastest way is to generate every string, which mismatches the pattern, at most k positions, and find all the occurrences of them using the exact matching algorithm. If k and the size of the alphabet is small enough, this gives a fast algorithm. For multiple strings, one can use the generalized suffix tree instead of the simple suffix tree.

4.8.3. Suffix Tree Application 1 – Substring Check

Given a text string and a pattern string, check if pattern exists in text or not. Once we have a suffix tree built for given text, we need to traverse the tree from root to leaf against the characters in pattern. If we do not fall off the tree (i.e. there is a path from root to leaf or somewhere in middle) while traversal, then pattern exists in text as a substring.



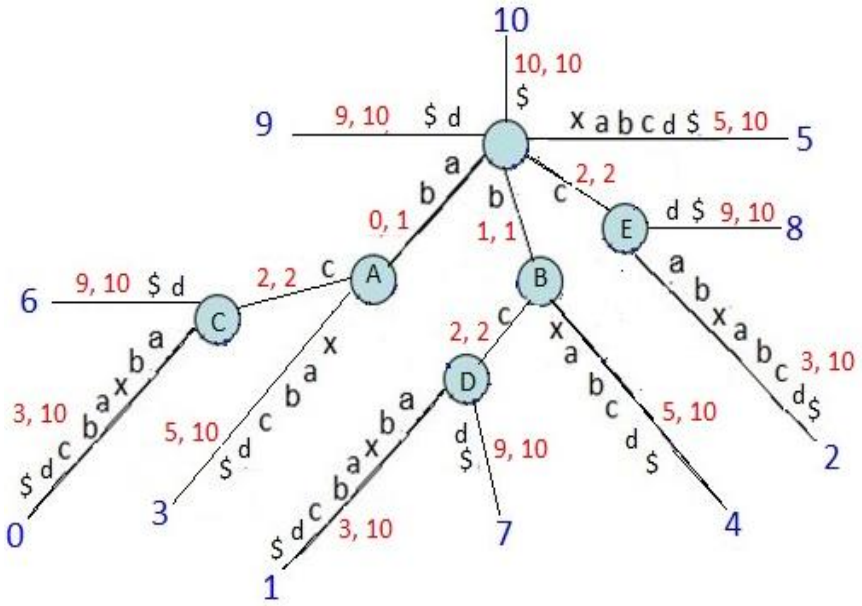
Suffix Tree for String S = "xabxa$"

Ukkonen's Suffix Tree Construction takes O(N) time and space to build suffix tree for a string of length N and after that, traversal for substring check takes O(M) for a pattern of length M.

4.8.4. Suffix Tree Application 2 – Searching All Patterns

Given a text string and a pattern string, find all occurrences of the pattern in string. In the Substring Check algorithm, we saw how to check whether a given pattern is substring of a text or not. If a pattern is substring of a text, then we will find all the positions on pattern in the text.

Let’s look at following figure:



Suffix Tree for String S = abcabxabcd$

This is suffix tree for String “abcabxabcd$”, showing suffix indices and edge label indices (start, end). The substring value on edges are shown only for explanatory purpose. We never store path label string in the tree. Suffix Index of a path tells the index of a substring (starting from root) on that path. Consider a path “bcd$” in above tree with suffix index 7. It tells that substrings b, bc, bcd, bcd$ are at index 7 in string. Similarly path “bxabcd$” with suffix index 4 tells that substrings b, bx, bxa, bxab, bxabc, bxabcd, bxabcd$ are at index 4.Similarly path “bcabxabcd$” with suffix index 1 tells that substrings b, bc, bca, bcab, bcabx, bcabxa, bcabxab, bcabxabc, bcabxabcd, bcabxabcd$ are at index 1.

If we see all the above three paths together, we can see that:

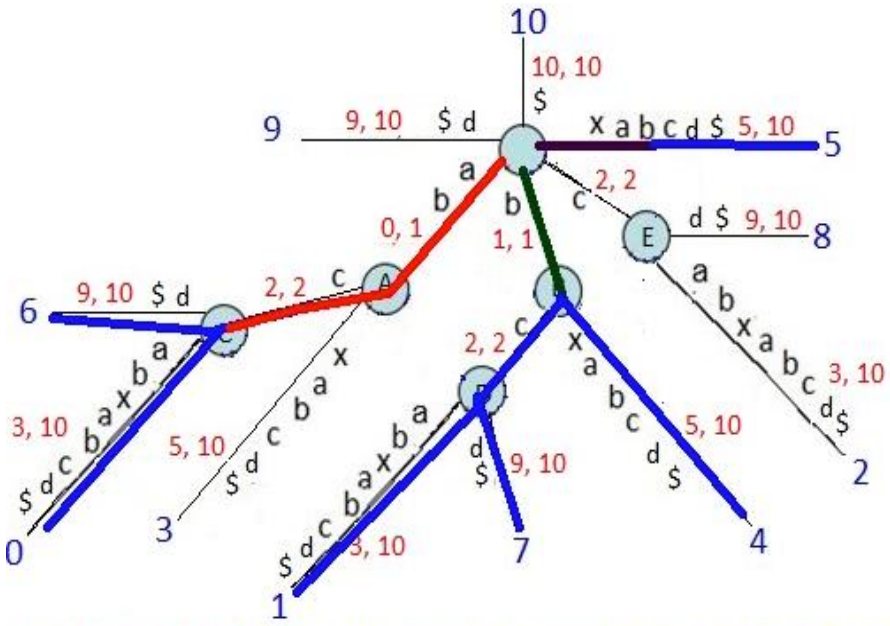
- Substring “b” is at indices 1, 4 and 7
- Substring “bc” is at indices 1 and 7

With above explanation, we should be able to see following:

- Substring “ab” is at indices 0, 3 and 6
- Substring “abc” is at indices 0 and 6
- Substring “c” is at indices 2 and 8
- Substring “xab” is at index 5
- Substring “d” is at index 9
- Substring “cd” is at index 8

Steps to find all the occurrences of a pattern in a string:

- Check if the given pattern really exists in string or not (As we did in Substring Check). For this, traverse the suffix tree against the pattern.
- If you find pattern in suffix tree (don't fall off the tree), then traverse the subtree below that point and find all suffix indices on leaf nodes. All those suffix indices will be pattern indices in string.



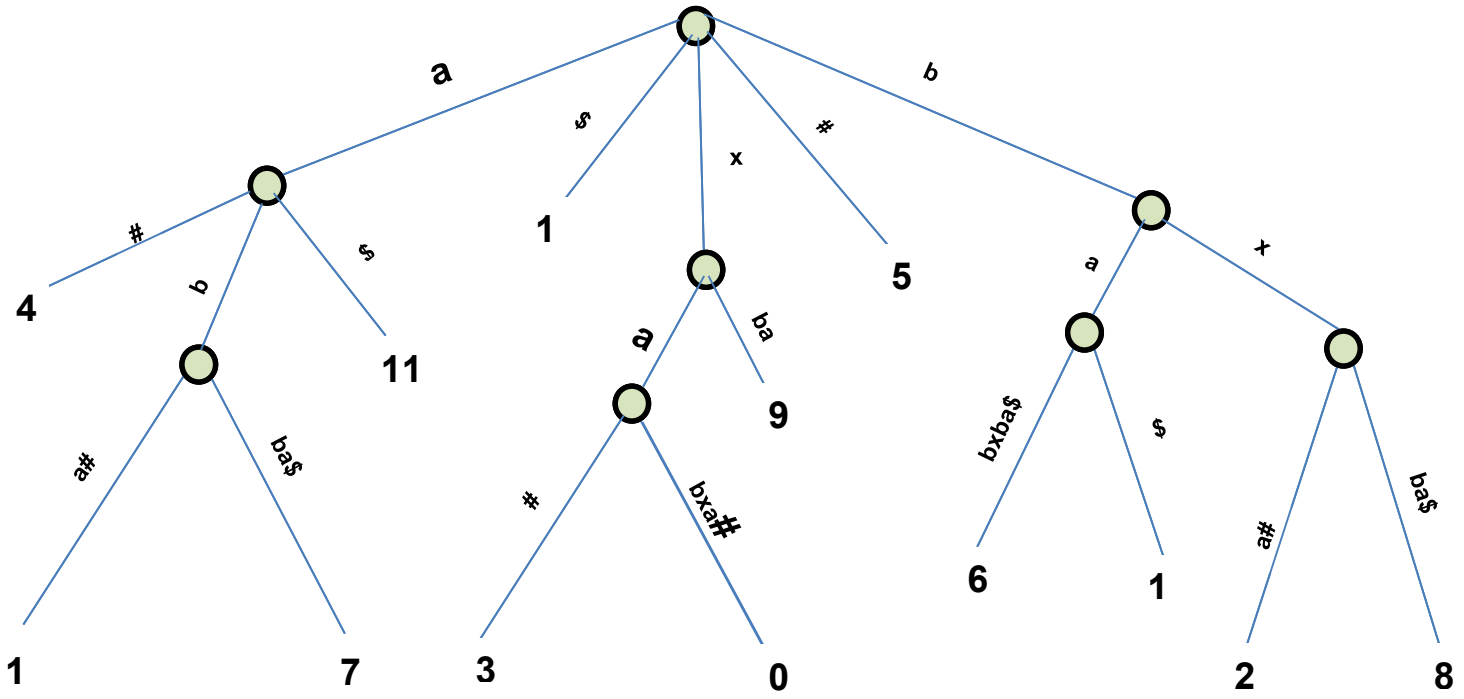
1. Substring abc is found, subtree traversal shows that it is at indices 0 and 6.
2. Substring b is found, subtree traversal shows that it is at indices 1, 4 and 7.
3. Substring xab is found, subtree traversal shows that it is at index 5.

Ukkonen's Suffix Tree Construction takes O(N) time and space to build suffix tree for a string of length N and after that, traversal for substring check takes O(M) for a pattern of length M and then if there are Z occurrences of the pattern, it will take O(Z) to find indices of all those Z occurrences. Overall pattern complexity is linear: O(M + Z).

4.8.5. Suffix Tree Application 3 – Longest Common Substring

Given two strings X and Y, find the Longest Common Substring of X and Y. We will discuss a linear time approach to find LCS using suffix tree. Here we will build generalized suffix tree for two strings X and Y. Let's take same example (X = xabxa, and Y = babxba) we saw in Generalized Suffix Tree 1.

We built following suffix tree for X and Y there:

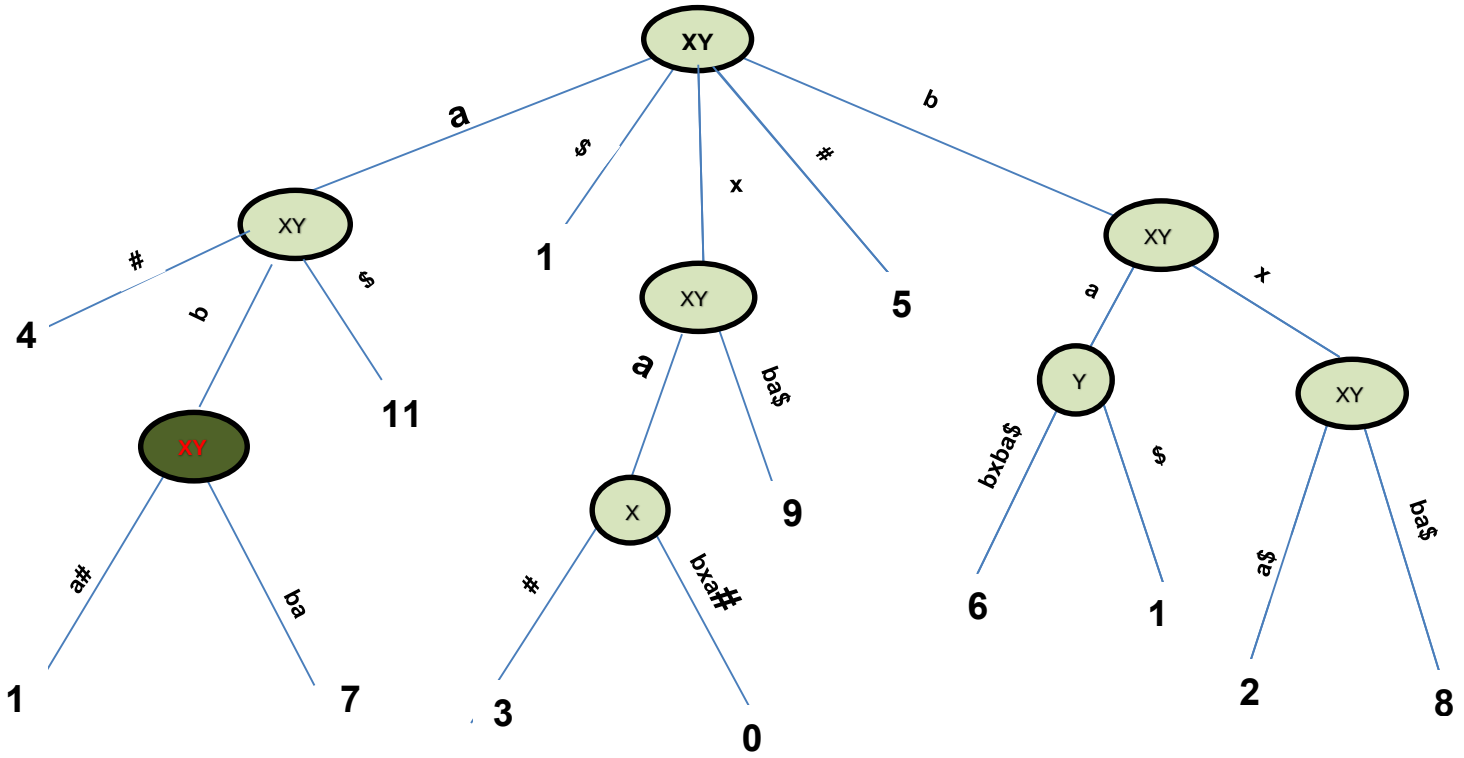


This is generalized suffix tree for xabxa#babxba$. In the above figure, leaves with suffix indices in [0, 4] are suffixes of string xabxa and leaves with suffix indices in [6, 11] are suffixes of string babxa. It is, because in concatenated string xabxa#babxba$, index of string xabxa is 0 and its length is 5, so indices of its suffixes would be 0, 1, 2, 3 and 4. Similarly index of string babxba is 6 and its length is 6, so indices of its suffixes would be 6, 7, 8, 9, 10 and 11.

With this, we can see that in the generalized suffix tree figure above, there are some internal nodes having leaves below it from,

- both strings X and Y (i.e. there is at least one leaf with suffix index in [0,4] and one leaf with suffix index in [6, 11]
- string X only (i.e. all leaf nodes have suffix indices in [0,4])
- string Y only (i.e. all leaf nodes have suffix indices in [6,11])

Following figure shows the internal nodes marked as “XY”, “X” or “Y” depending on which string the leaves belong to, that they have below themselves.



What these “XY”, “X” or “Y” marking mean?

The path label from root to an internal node gives a substring of X or Y or both. For node marked as XY, substring from root to that node belongs to both strings X and Y. For node marked as X, substring from root to that node belongs to string X only. For node marked as Y, substring from root to that node belongs to string Y only. By now, it should be clear that how to get common substring of X and Y at least. If we traverse the path from root to nodes marked as XY, we will get common substring of X and Y.

The path label from root to the deepest node marked as XY will give the LCS of X and Y. The deepest node is highlighted in above figure and path label "abx" from root to that node is the LCS of X and Y. If two strings are of size M and N, then Generalized Suffix Tree construction takes O(M+N) and LCS finding is a DFS on tree which is again O(M+N). So overall complexity is linear in time and space.

4.8.6. Suffix Tree Application 4 – Longest Palindromic Substring

Given a string, find the longest substring, which is palindrome. If given string is S, then approach is following:

- Reverse the string S (say reversed string is R)
- Get Longest Common Substring of S and R given that LCS in S and R must be from same position in S

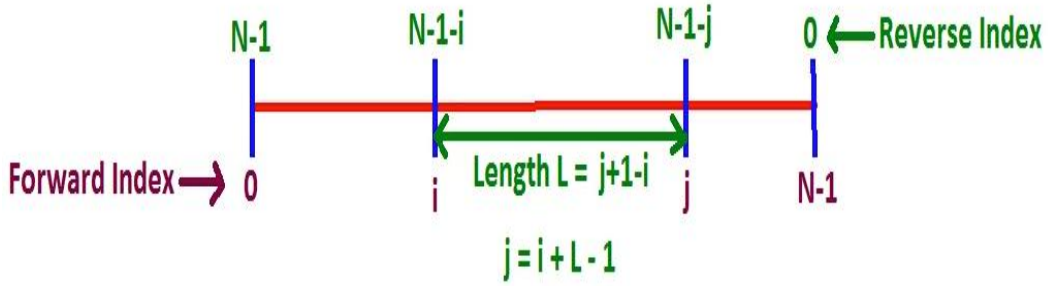
Let's look at following examples:

- For S = xababayz and R = zyababax, LCS and LPS both are ababa (SAME)
- For S = abacdfgdcaba and R = abacdgfdcaba, LCS is abacd and LPS is *aba*(DIFFERENT)
- For S = pqrqpabcdfgdcba and R = abcdgfdcbapqrqp, LCS and LPS both are pqrqp(SAME)
- For S = pqqpabcdfghfdcba and R = abcdfhgfdcbapqqp, LCS is abcdf and LPS is pqqp(DIFFERENT)

We can see that LCS and LPS are not same always. When S has a reversed copy of a non-palindromic substring in it, which is of same or longer length than LPS in S, then LCS and LPS will be different. In the second example above *(S* = abacdfgdcaba*)*, for substring abacd, there exists a reverse copy dcaba *in S*, which is of longer length than *LPS* aba and so *LPS* and *LCS* are different here.

If we look at second example again, substring *aba* in R comes from exactly same position in S as substring *aba* in S which is ZERO (0^{th} index) and so this is LPS.

The Position Constraint*:*



We will refer string S index as forward index (S_i) and string R index as reverse index (R_i). Based on above figure, a character with index i (forward index) in a string S of length N, will be at index N-1-i (reverse index) in it's reversed string R. If we take a substring of length L in string S with starting index i and ending index j (j = i+L-1), then in its reversed string R, the reversed substring of the same will start at index N-1-j and will end at index N-1-i. If there is a common substring of length L at indices S_i (forward index) and R_i (reverse index) in S and R, then these will come from same position in S if $R_i = (N - 1) - (S_i + L - 1)$ where N is string length. So to find LPS of string S, we find longest common string of S and R where both substrings satisfy above constraint, i.e. if substring in S is at index S_i, then same substring should be in R at index $(N - 1) - (S_i + L - 1)$. If this is not the case, then this substring is not LPS candidate.

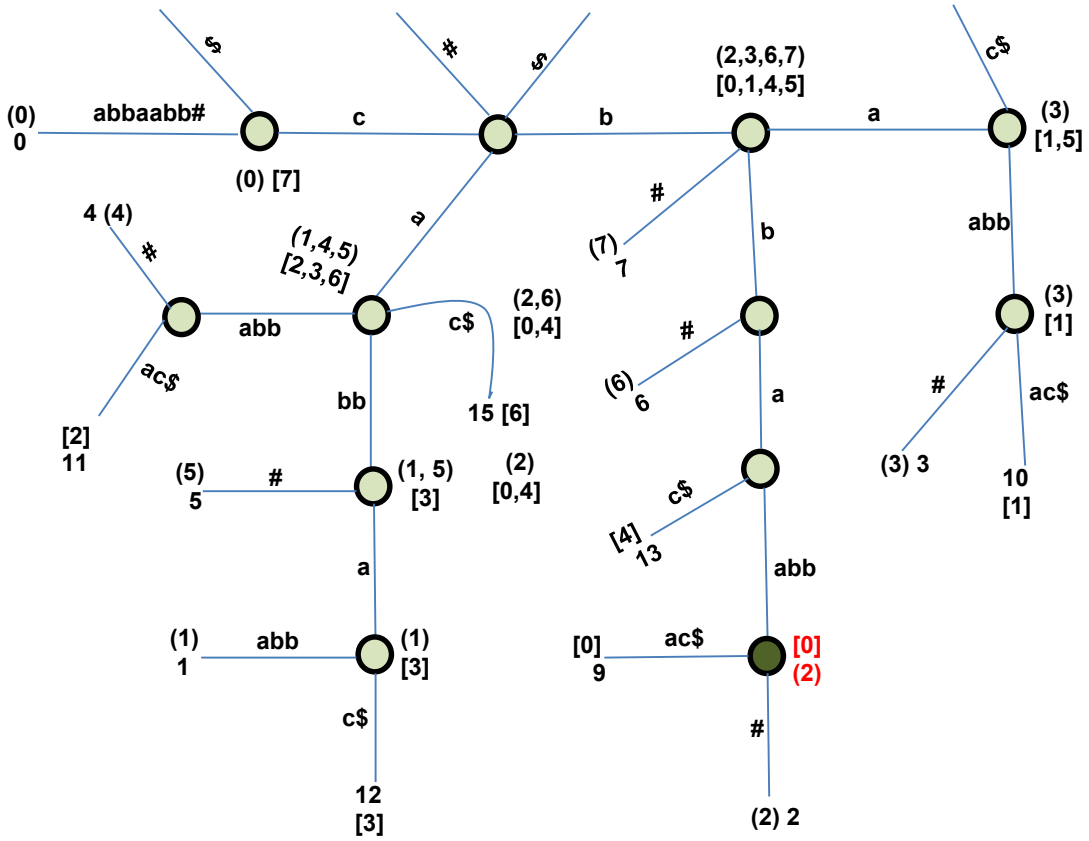
Now we will discuss suffix tree approach, which is nothing but an extension to Suffix Tree LCS approach where we will add the position constraint.

While finding LCS of two strings X and Y, we just take deepest node marked as XY (i.e. the node which has suffixes from both strings as it's children). While finding LPS of string S, we will again find LCS of S and R with a condition that the common substring should satisfy the position constraint (the common substring should come from same position in S). To verify position constraint, we need to know all forward and reverse indices on each internal node (i.e. the suffix indices of all leaf children below the internal nodes).

In Generalized Suffix Tree of *S#R$*, a substring on the path from root to an internal node is a common substring if the internal node has suffixes from both strings S and R. The index of the common substring in S and R can be found by looking at suffix index at respective leaf node. If string *S#* is of length N then:

- If suffix index of a leaf is less than N, then that suffix belongs to S and same suffix index will become forward index of all ancestor nodes
- If suffix index of a leaf is greater than N, then that suffix belongs to R and reverse index for all ancestor nodes will be N – suffix index.

Let's take string S = cabbaabb. The figure below is Generalized Suffix Tree for *cabbaabb#*bbaabbac$ where we have shown forward and reverse indices of all children suffixes on all internal nodes (except root). Forward indices are in Parentheses () and reverse indices are in square bracket [].



In above figure, all leaf nodes will have one forward or reverse index depending on which string (S or R) they belong to. Then children's forward or reverse indices propagate to the parent. Look at the figure to understand what would be the forward or reverse index on a leaf with a given suffix index. At the bottom of figure, it is shown that leaves with suffix indices from 0 to 8 will get same values (0 to 8) as their forward index in S and leaves with suffix indices 9 to 17 will get reverse index in R from 0 to 8.

For example, the highlighted internal node has two children with suffix indices 2 and 9. Leaf with suffix index 2 is from position 2 in S and so its forward index is 2 and shown in (). Leaf with suffix index 9 is from position 0 in R and so its reverse index is 0 and shown in []. These indices propagate to parent and the parent has one leaf with suffix index 14 for which reverse index is 4. So on this parent node forward index is (2) and reverse index is [0, 4]. And in same way, we should be able to understand the how forward and reverse indices are calculated on all nodes.

In above figure, all internal nodes have suffixes from both strings S and R, i.e. all of them represent a common substring on the path from root to themselves. Now we need to find deepest node satisfying position constraint. For this, we need to check if there is a forward index S_i on a node, then there must be a reverse index R_i with value $(N - 2) - (S_i + L - 1)$ where N is length of string *S#* and L is node depth (or substring length).

If yes, then consider this node as a LPS candidate, else ignore it. In above figure, deepest node is highlighted which represents LPS as bbaabb.

We have not shown forward and reverse indices on root node in figure. Because root node itself doesn't represent any common substring. How to implement this approach to find LPS? Here are the things that we need:

- We need to know forward and reverse indices on each node.
- For a given forward index S_i on an internal node, we need know if reverse index $\mathbf{R_i = (N - 2) - (S_i + L - 1)}$ also present on same node.
- Keep track of deepest internal node satisfying above condition.

While DFS on suffix tree, we can store forward and reverse indices on each node in some way (storage will help to avoid repeated traversals on tree when we need to know forward and reverse indices on a node). Later on, we can do another DFS to look for nodes satisfying position constraint. For position constraint check, we need to search in list of indices. The data structure can be processed in linear time, and each of the linear number of queries can be answered in constant time, so the total time is linear.

4.8.7. Suffix Tree Application 5 – DNA contamination

When processing DNA in a laboratory, foreign DNA sequences often contaminate the sequence of interest, like the DNA of a vector or of the host organism. Even a very small amount of contamination can be inserted into the DNA or can be copied by the polymerase chain reaction (PCR), which is used to amplify the DNA sequences. DNA contamination is a very serious problem to be solved, as it can falsify the whole experiment. Usually, many potential contaminating DNA sequences are known, like cloning vectors, PCR primers, whole genome of the host etc.

This problem can be modeled in the following way: we are given a string S (the DNA string of interest) and a string set C (the known possible contaminators). The goal is to and all substrings of all $T \in C$ which occur in S and longer than a certain length l. These are the substrings, which are likely to be contaminators of the DNA string of interest. With suffix trees, the solution is to build the generalized suffix tree of S and C, and mark all internal nodes v, which is part of a path representing a suffix of S and of another one representing a suffix of an element of C. Among the marked nodes, take all with path length at least l. These will be the parts, which are likely to be contaminated.

4.8.8. Suffix Tree Application 6 – Genome-scale projects

Suffix trees have been applied in several genome projects. Three of these projects are the mapping of the Arabidopsis thaliana, the Saccharomyces cerevisiae (yeast) and the Borrelia burgdorferi genomes.

In the Arabidopsis project suffix trees were used in three ways. First, they searched the contaminations by known vector DNA sequences. Here, a generalized suffix tree was created for the vector sequences. As a second step, all sequenced fragments were checked to find duplications. The fragment sequences were also kept in a generalized

suffix tree. Third, suffix trees were also used to find biologically known and important sequences in the found sequences. Patterns were represented as regular expressions. The problem was formulated as an inexact matching problem with a small number of errors. They used suffix trees to give an answer to the question.

In the yeast and the Borrelia projects the suffix tree was the main data structure, and was used to solve the fragment assembly problem, which is the following: we are given a large number of fragments which are partially overlapping, and we have to find the full sequence.

5. Summary, future work and conclusion

In this paper we gave a short review of few string matching algorithms and their complexities. This paper mainly focuses on various construction and applications of suffix tree. Large sequences can be analyzed and compared faster by algorithms using suffix trees. We have described the main algorithmic techniques used for its construction. We also suggested some improvisation techniques of suffix tree such as suffix array and generalized suffix tree and presented some of its applications in different areas. The substring check, searching all patterns, longest common substring, and longest palindromic substring problems can be solved efficiently using suffix tree. Suffix trees were also used in DNA contamination and genome scale projects.

Every suffix tree algorithm can be systematically replaced with an algorithm that uses a suffix array enhanced with additional information (such as the LCP array) and solves the same problem in the same time complexity. So, the future work may include implementation of suffix array for solving the above problems.

In conclusion, when the text is not known or both the pattern and the text are presented at the same time, algorithm like Boyer-Moore method is better than suffix trees. Compared to other algorithms, suffix trees are suitable when the text is fixed and known, and the pattern varies (query mode for the pattern). This is a very common and important situation appearing mainly in Bioinformatics.

6. References

- http://www.cs.princeton.edu/courses/archive/fall04/cos226/lectures/string.4up.pdf
- http://en.wikipedia.org/wiki/Rabin%E2%80%93Karp_algorithm
- http://users.csc.calpoly.edu/~dekhtyar/448-Spring2013/lectures/lec03.448.pdf
- http://homepage.usask.ca/~ctl271/857/suffix_tree.shtml
- http://web.stanford.edu/class/cs97si/suffix-array.pdf
- http://web.stanford.edu/~mjkay/gusfield.pdf
- http://web.stanford.edu/~mjkay/Maass.pdf
- http://www.di.unipi.it/~grossi/IND/survey.pdf
- http://www.cs.helsinki.fi/u/tpkarkka/opetus/13s/spa/lecture10.pdf
- http://ab.inf.uni-tuebingen.de/teaching/ss07/albi2/script/suffixtrees_14May2007.pdf
- http://www.sciencedirect.com/science/article/pii/S1570866703000625
- http://www.bgjackson.net/bcb597/supplement/StringApplications.pdf
- Introduction to Algorithms by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein.