Chapter 1

Data Structures for Pattern Matching

Olga Sergeeva

For many applications, efficient string processing is crucial. Searching for a substring or a subsequence in a string; searching for common substrings of a set of strings; finding out the number of direct repetitions these are just a few important examples of the problems often appearing in work with strings. In the applications, there is need in solving these problems as efficient (in asymptotic) as possible, and also to store the strings 'economically'. So, there arises a question of efficient strings representations, both being compact and providing good bases for algorithms.

This paper is concerned with two representations, in some sense revealing the structure of the initial string and thus meeting many demands: suffix trees and suffix arrays.

1.1 Suffixes

Definition 1.1. Let s be a string. s' is called a suffix of s, if $s = as'$ for some a. Note, that an empty string is a suffix of any string.

It turns out, that if the suffixes are well structured , the resulting construction can be very informative and can be a good base for developing efficient algorithms. Also, both structures we are to discuss can be built in time, linear in the length of the initial string (in this paper, always n).

Why is it possible to build representations, based on suffix relations, efficienly? The very simple (but also very general) idea is that the suffixes are closely related, being parts of each other. Maybe it s worth looking at how this idea refracted in combination with other ideas, and what it lead to in different algorithms, as they historically appeared. But we in this paper will discuss mostly those which made the most of it.

One of the string representations of interest is suffix tree.

1.2 Suffix trees

1.2.1 Definitions, examples, background

Definition 1.2. Suffix tree for a string s is a rooted tree with edges, marked with substrings of s, having the following properties:

1. Any concatenation of the marks along each path from the root to a leaf forms a suffix and every suffix appears once.
2. The marks on the edges, having a common root, begin with different symbols of the alphabet.

From the definition, you can see that there must be as many leaves in the suffix tree as there are non-empty suffixes in s, i.e. n.
What is good in such a representation? The following bright example can give some comprehension of the advantages of suffix trees.

Example 1.1 (Substring search). Suppose that we have a string s and a (shorter) string p, for which we are to answer if it occurs in s.
In fact, even this is not enough detailed description of the problem. The context is important: will we work with s and p once? Or we have a constant pattern, which we are to search in many strings? Or we have a fixed string s, and are to answer many queries about different p? Answers to such questions of course greatly influence the structures and the algorithms preferred.
Suppose that this is s which is fixed (some encyclopedia, for example). So it is s to be preprocessed how can we answer if p is a substring of s, having its suffix tree S?
If p is a substring, then it s a beginning of some suffix. In S, every suffix must appear as a concatenation of marks from the root to a leaf, so if p begins with a symbol, with which no branch from the root begins, we can definitely say that p is not a substring. Otherwise, we should search only in the subtree, corresponding to the branch beginning with this symbol (there is only one such branch from the root!) Reasoning the same for the consequent vertices, we will answer our question, making in each vertex a constant number of comparisons (the alphabet is constant), and there will be no more then the length of p (denote: $|p|$) steps. So overall time is $O(|p|)$ operations plus $O(|s|)$ to build the suffix tree.
Note that for our case the distribution of work is very successful: although for the first query we spend $O(|p| + |s|)$ time, for all the consequent queries time will be linear in the length of the pattern! So, although for the first query the time is the same with, for example, Knuth-Moris-Pratt algorithm, but in that algorithm p needs $|s|$ time preprocessing, not s. So in fact we have an algorithm which, after initial preprocessing, answers the queries in time, linear in the length of p. Mention, that Donald Knuth did not believe in existence of such an algorithm, until suffix trees were invented.

Suffix trees have many other applications, for instance, in search for the longest common substring of a pair (and, further a set of) strings; for repetitions; for longest common ancestor.
It is also like a 'bridge' to much more difficult and important problem of inexact matching when given strings may contain errors and in practice they will contain them!

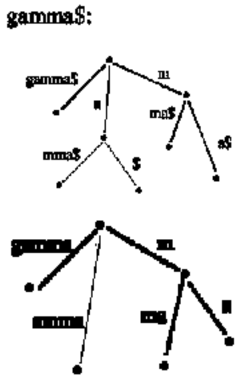

Figure 1.1: Suffix tree and implicit tree for 'gamma'.

Note. We gave a definition of suffix tree, but didn t check correctness of the definition: does suffix tree exist for an arbitrary s? Indeed, if we want to have a leaf for each suffix, then it is impossible to build a suffix tree for a string, in which one suffix s_1 is a prefix of another suffix s_2 we will inevitably spell out the shorter suffix, spelling out the longer one. This problem is easy to solve: if we add a symbol not from alphabet (say, $'\$'$) to the end of s, then no suffix will be a prefix of another suffix.

We will call a tree for a string without additional symbol an implicit tree for S. In implicit tree, there is exactly one path spelling out any suffix, but the ends of some suffixes are not marked anyhow.

1.2.2 Algorithms

The suffix tree can be built naively , adding suffixes to the tree one by one, beginning from the longest suffix (which is the string itself). To add a new suffix, we try to spell it out in the tree, and at the point it s no more possible, create a new branch, marked with the not spelled remainder. This approach demands $O(|s|^2)$ time.
The first linear-time algorithm, by Weiner, appeared in 1973, in his article Linear Pattern Matching Algorithms . Although it was linear in time, it was space-consuming. A less complex and less space-consuming algorithm was invented in 1976 (McCreight. A Space-Economial Suffix Tree Construction Algorithm). Eventually, in 1993, Ukkonen in his On-Line Construction of Suffix Trees introduced his simpler algorithm, having several nice features.

1.2.3 Ukkonen's algorithm

Description

We will start with a simple but not efficient algorithm, and then in several steps, suggested by common sense ('how not to do excessive work?'), will transform it to linear.
Ukkonen s algorithm is on-line: it is split up into $|s|$ phases, after each of them there is an implicit(!) tree for a prefix of s. We begin with the string, containing

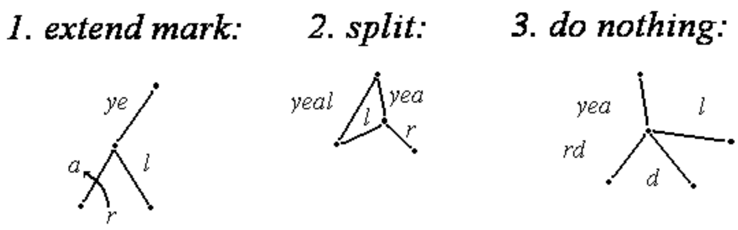

Figure 1.2: Extending 'yea' to 'year' in different trees.

only the first symbol of s, and each phase increase the length of the processed string by one. Phase $i+1$ is itself split into $i+1$ *extensions*, one extension for each from the $i+1$ suffixes of $s[1..i+1]$. In the extension j in the phase $i+1$ algorithm finds the end of the path, marked with $s[j..i]$. It then extends this substring, adding $s(i+1)$ to its end, if it doesn't exist in the tree. So, in the phase $i+1$ it one by one inserts the strings $s[1..i+1], s[2..i+1], .., s[i+1]$ into the tree, if they do not exist.
T_1 consists of one edge, marked with $s(1)$.
After the last phase, we will carry out one more phase , adding symbol \$. The resulting implicit tree for $s\$$ will be a real suffix tree, as we already pointed out. The formal definition of the algorithm is as follows.

Algorithm.
Build T_1.
for i from 1 to $m-1$ do begin {phase $i+1$}
  for j from 1 to $i+1$ begin {extension j}
    find in the current tree the end of the path with mark $S[j..i]$.
    If needed, extend the path with $s(i+1)$, providing existence
    of the string $s[j..i+1]$.
  end;
end;

Let's look closely on the possible cases of extention (see Fig. 2).

1. We are just to extend a mark on an edge
2. When we are to add an edge (and maybe to split an existing edge into two)
3. When nothing is needed to be done.

We will refer to these cases as the first rule, the second rule and the third rule for the algorithm.
What do we spend time for? For looking for the end of the current suffix in a tree after this, we spend constant to prolong it. So, it is essential how we search for the ends of the suffixes in T_i.
We can find the end of a suffix s in $O(|s|)$, walking from the root each time. In this case, we will build the T_{i+1} from T_i in $O(i^2)$, so the final tree will appear

after $O(n^3)$ operations, comparing to $O(n^2)$ in the naive algorithm! We ll reduce this to $O(n)$ using some observations and techniques. Each of them is (just!) a useful heuristics, which cannot qualitatively change estimation of time for the worst case, but applied together they result in essential speeding-up.

Suffix links

Definition 1.3. Suffix link is a pointer from an inner vertex v with the path mark $x\alpha$ to a vertex $s(v)$ with mark α, if it exists in the tree, where x is a symbol, and α is a string. Notice, that if α is empty, suffix link points to the root.

It is easy to see that there is a vertex $s(v)$ for every inner vertex of the tree. Moreover, the following statement is true: if a vertex v with path mark $x\alpha$ is added to the tree in the extension j of the phase $i+1$ (that is, the second rule is applied), then the vertex $s(v)$ either already exists in the tree of will be created in the next extension, in this phase. In both cases, we will find it incidentally, so adding a pointer will not require additional search.
Using suffix links can substantially reduce amount of search.
First, we will maintain a pointer to the end of the longest suffix $s[1..i]$, so for the first extension we need only to prolong a mark on the given edge - no search.
To proceed with the next one, we will not move from the root every time we search for the end of the next suffix. Instead, we will get from the available ending point up to the first inner vertex v (if it s not an inner vertex itself), walk along its suffix link $s(v)$ and search only in the subtree of $s(v)$. If our current suffix can be written in the form $x\alpha\beta$, where $x\alpha$ is the mark of the path to v, then $s(v)$ is marked with α, so trying to spell out the next suffix (which is $\alpha\beta$), we would surely have come to $s(v)$. Adding new suffix links after transformation of the tree in each extension (if needed), we will easily maintain the tree in this pleasant for processing form.

Note. This note will help us to estimate the algorithm complexity.
Denote by depth of a vertex the number of edges on the way from the root to that vertex. Then the moment we are moving along the suffix link, the depth of v, exceeds the depth of $s(v)$ by no more then 1. The reason is that the prefixes of the shorter string (α) will occur in the string more often, so there will be more bifurcations along the way corresponding to α. And in the case no bifurcations are added comparing to the way corresponding to $x\alpha$, $d(x\alpha) - d(\alpha) = 1$. So, getting along the suffix link, we will not get much nearer to the root. Later this will help us to estimate how many times we could descend, and then the overall processing time.

Not check, but search

Still, there are unnecessary steps. The end of the next suffix to prolong surely exists in the tree (in the subtree from $s(v)$, as we cleared up), so we are not to check its existence, but only to find its end. We do not have to compare every symbol on en edge, once having chosen it. If it is shorter then the not spelled out remainder of the suffix, we can skip over it to the next vertex; if it is longer split it at the needed point. Doing this way, we spend for descending each edge constant time. All we need for this style of work is to know the number of symbols on the edge (which is a detail of implementation) and to be able to extract a symbol in s by its number in constant time. Along with the notice on the vertex depths correlation, this yields in the following

Theorem 1.1. *In the improved algorithm, each phase takes $O(n)$ time.*

Proof. Summing up, what steps we perform during one extension? Get up to the nearest inner vertex no more than one edge; go along the suffix link; descend some vertices; apply one of the rules of suffix extension; maybe add a new suffix link. All these actions, except descending, take constant time.
We need to estimate how many times we descend during one phase. Let s pay our attention to the current vertex depth changes. Raising, we decrease it by no more than 1; the same with walking along the suffix link; and while descending, we increase current depth that s why overall depth increment over the phase in not more than $3 * n$. □

Corollary 1.1. *The current version of Ukkonen s algorithm terminates in $O(n^2)$.*

Question. We spent so much effort, and got nothing in comparison with the naive algorithm?

The last touches

Improving an algorithm, one can encounter the following problem: if the output is large, the algorithm cannot be very fast, for working time is no less then output size. This is the case: if we don't change the format of representation the tree, we will not get further optimization, because in general case the overall length of labels on the edges doesn't have to be linear.

Example 1.2. Suffix tree for the string 'abcd...xyz' consists of 26 branches, with marks having 26, 25, ... 1 letters on them. So, the overall length of marks is $26 * 27/2$. Although for arbitrary long strings there will not exist such an example, because the size of the alphabet is considered constant, this example gives notion on how much redundancy can the marks contain.

Example 1.3. Tree for $(a)^n(b)^n(a)^{n-1}b^{n-1}...a^2b^2ab$.

We will modify algorithm in a few touches, taking the following observations:

1^{st}*touch* If we replace the labels with indexes (the beginning and the end of the substring in s), we'll have two numbers, corresponding to each edge, and as the number of edges is less then $2*m-1$, linear space is spent. This also simplifies maintaining the length of the edge (the detail, which we need), making it more consequent. After mentioning that, we can immediately forget that we have not symbols, but numbers, because working with them is the same.

2^{nd}*touch* If $x\alpha\beta$ appeared in the tree, then definitely $\alpha\beta$ appeared also. Then when we are to apply the third rule, we can complete with the phase. So a phase is a consequency of extensions, which use the first (prolonging a mark) and the second (branching off) rules.

3^{rd}*touch* A leaf cannot become an inner vertex, because the three 'rules' algorithm uses do not transform leaves anyhow.

During the phase, we add the same symbol to edge marks. In terms of indices, we increment by one ending indices, setting them to i on the phase i.

We will split or do nothing only with the suffixes, not processed in the previous phase (those, to which we applied the do nothing rule).

If in the previous phase we ended in extension j, in the beginning of phase we already know that all we have to do with the first j suffixes is to increment by one their last end. That s why it s efficient to write on such edges the mark of infinity, in the phase i implying infinity is i, and to replace them only after the tree is built. This means, that we build the tree 'in one path', moving only 'forward'.

The last observation yealds the

Theorem 1.2. *Ukkonen s algorithm terminates in $O(n)$.*

Difficulties

We saw that suffix trees can make string processing much simpler. At the same time, this is a complicated structure, which is sometimes difficult to implement. The reasons are

1. No "locality" – bad for paging
2. Dependency on the length of the alphabet (Σ): the 'constant' for choosing the right branch gets bigger with grouth of $|\Sigma|$.
3. Number of "children" ranges for different vertices – no general ways of representation: the vertices near the root have many children (almost $|\Sigma|$), and for them arrays provide good representation; the vertices near leaves have almost no children, arrays would be too rarefied, and for these vertices linked lists are preferred; and for 'middle' vertices balanced trees and hashing are better (linked lists would increase time for search, in case $|\Sigma|$ is large).

Due to these reasons, in a number of applications a simpler, although a bit less convenient for algorithms, structure is preferred.

1.3 Suffix arrays

1.3.1 The very first algorithm

Definition 1.4. Suffix array for a string s is an array, containing the suffixes of s in lexicographic order.

The idea to alphabetically order the suffixes belongs to Udi Manber and Gene Myers (1993, "Suffix Arrays: a New Method For On-Line String Searches").
They proposed an algorithm of direct constructing the array (i.e. construction, not based on firstly built suffix tree) in $O(n*logn)$ time. This algorithm not only

builds the array, but on the way gathers some additional information (useful for algorithms). In their article, Manber and Myers also presented an algorithm of search, using this information, for a pattern P in $O(|P| + logn)$ time.
The clear advantage of suffix arrays in comparison with suffix trees is that they are much less complicated structure. So, they were preferred in several fields even though there were no known linear time algorithms for direct suffix array construction. Such algorithms (in amount of three, simultaniously and independently!) appeared in 2003, and we will touch upon one of them.

Note. Search time on suffix arrays ($O(|P| + logn)$) seems to be great loss in comparison with $O(|P|)$ for suffix trees. But in practice, these values ($O(|P|)$ and $O(logn)$) are usually comparable, and search time does not decrease dramatically.

As we mentioned, suffix array can be built (in linear time) from a corresponding suffix tree. This approach has obvious disadvantages. Here is an idea of an algorithm, based on different approach.

Algorithm. As with the trees, we build the array inductively, greatly using it's structure. Initially, we have an array with unordered suffixes. Beginning with sorting the suffixes by the first symbol (which is linear in the string's length - radix sort), every phase we twice the number the suffixes are sorted on.
After the phase H, the suffixes are organized into buckets, holding suffixes with the same H first symbols.

If A(i) is the suffix in the first bucket, A(i-H) should be first in its 2H-bucket. We can move it to the beginning of its 2H-bucket, and mark this fact. For every bucket, we need to know the number of suffixes in this bucket that have already beer moved and placed in 2H-order. The algorithm basically scans the suffixes as they appear in the H-order and for each A(i) it moves A(i-H) (if it exists) to the next available place in its bucket.
The number of phases is logarithmical, so the overall running time is $O(n*logn)$. The implementation is quite interesting, see [MM93].

1.3.2 Skew algorithm

In 2003, independently and in parallel, three different direct linear time suffix array construction algorithms were introduced (by Kim; by Ko and Aluru; and the one we are to consider - 'Skew' algorithm by Juha Karkkainen and Peter Sanders.)
Before getting to the idea of the *skew algorithm*, we need to refer to some background, and thus to give another glance at the history of suffix trees development. Together with the line of algorithms of Weigner, McCreight, Ukkonen, in which the suffix tree is built inductively, with use in some way of the tight relations between suffixes which, provided with insight into these relations, make it possible to organize induction very efficiently - together with this line, there existed another line, introduced by Martin Farach. Farach's suffix tree construction was based on quite different idea: construct separate trees for suffixes

starting at odd positions (recursively) and for the remaining suffixes (using the results of the first step). Merge the two suffix trees into one. Merging, being a difficult procedure, relies on structural properties of suffix trees that are not available in suffix arrays. (Worth mentioning that Farach's approach has an important advantage of not being dependent on the alphabet size.) Kim (the author of another linear algorithm) managed to perform similar merging with suffix arrays, but the procedure is still very complicated.
The skew algorithm has a similar structure:

1. Construct the suffix array of the suffixes starting at positions $imod3 \neq 0$. This is done by reduction to the suffix array construction of a string of two thirds the length, which is solved recursively.

2. Construct the suffix array of the remaining suffixes using the result of the first step

3. Merge the two suffix arrays into one.

The use of two thirds instead of half of the suffixes in the first step makes the last step indeed easy: a simple comparison-based merging is sufficient. For example, to compare suffixes strating at i and j with $imod3 = 0$ and $jmod3 = 1$, we first compare the initial characters, and if they are the same, we compare the suffixes starting at $i+1$ and $j+1$ whose relative order is already known from the first step (the situation here is quite similar to the situation in the algorithm in previous section, see picture).
The figure 3 gives an example of algorithm. In [KS03], together with useful ideas and theoretical observations, there is a (short and easy understandable) implementation in C++.

Theorem 1.3. *The skew algorithm can be implemented to run in time $O(n)$.*

Proof. The second step is easy, due to the idea, same with the one for step 3. Again: the suffixes S_i with $imod3 = 0$ are sorted by sorting the pairs $(s[i], S_{i+1})$, where s is the initial string. So, it's not difficult to see that the second and the third steps require linear time, and the execution time obeys the reccurence $T(n) = O(n) + T(2n/3)$, $T(n) = O(1)$ for $n < 3$. This reccurence has the solution $T(n) = O(n)$. $\square$

Note. In the previous section, we mentioned that generally suffix array should be built together with collecting some additional information (an array of longest common prefixes of suffixes that are adjacent in the suffix array), which makes it much more valuable for algorithms. Referring to [KS03] for details, we will just mention that this information can be gathered in the course of the *skew algorithm* as well.

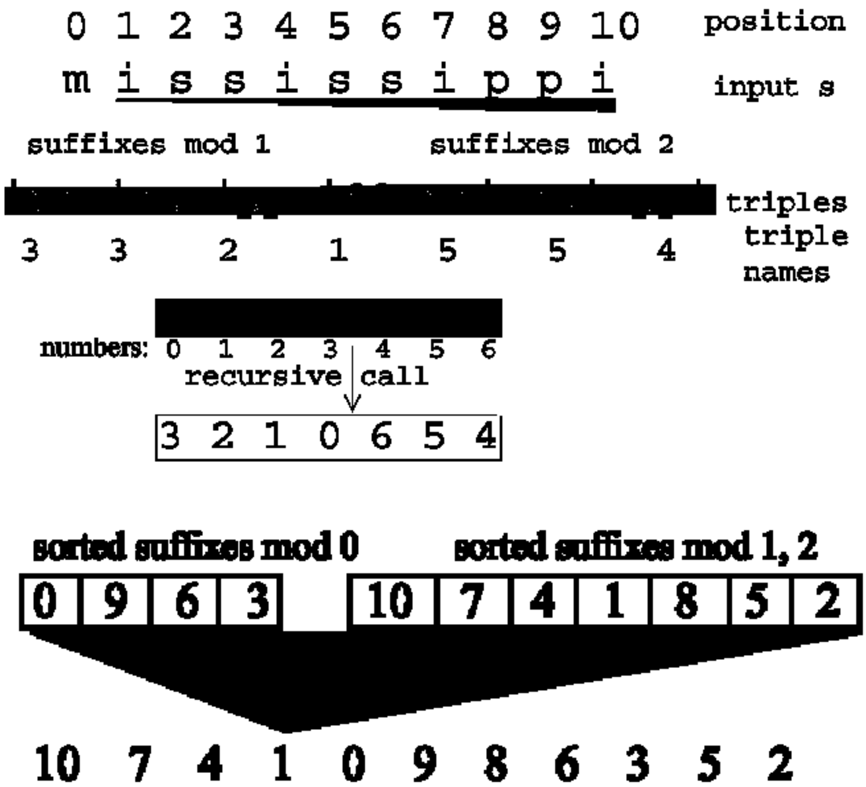

Figure 1.3: Skew algorithm, applied to string 'mississippi'.