Chapter 4

On-line construction of suffix trees

We present here the first basic data structure representing the set \( \mathcal{F}(text) \) of all factors (subwords) of a given word. Their importance derives from a multitude of applications. For simplicity we assume throughout this chapter that the alphabet \( A \) is of constant size (otherwise, the complexity of algorithms should be multiplied by \( \log |A| \)). Since \( \mathcal{F}(text) \) is a set, the most typical problem related to such data structures is the Membership Problem:

\[
\text{test if } x \in \mathcal{F}(text).
\]

The data structure \( D \) representing the set \( \mathcal{F}(text) \) is said to be good if:

1. \( D \) has linear size,
2. \( D \) can be constructed in linear time,
3. \( D \) enables to test the membership problem in \( O(|x|) \) time after preprocessing \( text \).

4.1 Tries and their compact versions

Our approach to represent the set of factors of a text is graph theoretical. Let \( G \) be an acyclic rooted directed graph in which the edges are labeled with symbols or with words: \( \text{label}(e) \) denotes the label of edge \( e \). The label of a path \( \pi \) (denoted by \( \text{label}(\pi) \)) is the composition of labels of its consecutive edges. The edge-labeled graph \( G \) represents the set:
Labels(G) = \{label(\pi) : \pi \text{ is a directed path in } G \text{ starting at the root} \}.

We also say that G represents the set of factors of text if Labels(G) = \mathcal{F}(text).

The first naive approach to represent \mathcal{F}(text) is to consider graphs that are trees in which edges are labeled by single symbols. These trees are called subword tries. Figure 4.1 shows the trie associated with the 6-th Fibonacci word Fib_6 = \text{abaababa}. In these trees, the links from a node to its children are labeled by letters. In the tree associated with text, a path down the tree spells a factor of text. All paths from the root to leafs spell suffixes of text. And all suffixes of text are labels of paths from the root. In general, these paths do not necessarily end in a leaf. The nodes correspond to subwords of the given text, each node can be identified with the word "spelled" by the path from the root to this node. In tries and suffix trees we distinguish nodes which correspond to suffixes of the given text, we call them essential nodes. Essential nodes are shaded in black in Figure 4.1.

Observation. The tries are not "good" representations of \mathcal{F}(text), because they can be too large. If text = a^n b^m a^n b^d, then Trie(text) has a quadratic number of nodes. We define a chain in a trie as a longest path consisting of non-essential nodes with outdegree one, except possibly the extremities of the chain. Two subtrees of a trie are isomorphic iff they have the same sets of paths leading from their roots to their essential nodes. We consider two kinds of succinct representations of the set \mathcal{F}(text). They both result from compacting the tree Trie(text). Two types of compaction can be applied, separately or simultaneously:

- compacting chains, which produces the suffix tree of the text,
- merging isomorphic subtrees (e.g., all leaves are to be identified), which leads to the directed acyclic word graph (DAWG) of the text, discussed in Chapter 6.

The suffix tree ST(text) is the compacted version of Trie(text) when using the first method of compaction, see Figure 4.1 and Figure 4.2. Each chain \pi (path consisting of nodes of out-degree one) is compacted into a single edge e with label(e) = [i, j], where text[i..j] = label(\pi) (observe that a compact representation of labels is also used). Note a certain nondeterminism here, because there can be several possibilities of choosing i and j representing the same factor text[i..j] of text. Any such choice is acceptable. In this context we identify the label [i, j] with the word text[i..j]. The tree ST(text) is called the suffix tree of the word text.

For a node v of ST(text), let val(v) be label(\pi), where \pi is the path from the root to v. Whenever it is unambiguous we identify nodes with their values, and paths with their labels. Note that the suffix tree obtained by compacting
Figure 4.1: The tree $Trie(text)$ and its compacted version, the suffix tree $ST(text)$, for the 6-th Fibonacci word: $abaababa$. The essential nodes are black. The numbers at these nodes indicate starting positions of the suffixes corresponding to the paths leading to these nodes. It is possible that the suffix tree contains some internal nodes with only one son since essential nodes are not deleted when compacting the trie.
CHAPTER 4. ON-LINE CONSTRUCTION OF SUFFIX TREES

Figure 4.2: The suffix tree for the word of Figure 4.1 with end-marker: 
\( ST(abaababa\#) \). There are no internal nodes of out-degree one.

Figure 4.3: A suffix link \( suf \) points to the node representing the factor with the first letter removed, if such node exists.
chains has the following property: the labels of edges starting at a given node are words having different first letters. Therefore, the branching operation performed to visit the tree is reduced to comparisons on the first letters of the labels of outgoing edges. If we assume that no suffix of text is a proper prefix of another suffix (this is satisfied with a right end-marker), the leaves of $\text{Trie(text)}$ are in one-to-one correspondence with the non-empty suffixes of text. The following fact about suffix trees is trivial, though a crucial one.

**Lemma 4.1** The size of the suffix tree $ST(\text{text})$ is linear ($O(||\text{text}||)$).

The crucial concept in efficient construction of tries and suffix trees is the table of suffix links: if $x$ is the string corresponding to the node $u$ then $\text{suf}[u]$ is a node which represents the string $x$ with the first letter cut off, see Figure 4.3. If there is no such node then $\text{suf}[u] = \text{nil}$. By convention, we define $\text{suf}[\text{root}] = \text{root}$.

### 4.2 Prelude to Ukkonen algorithm

We denote the prefix of length $i$ of the text by $p^i$. We add a constraint on the suffix trie construction: not only do we want to build $\text{Trie}(p)$, but we also want to build on-line intermediate trees (see Figure 4.5)

$$\text{Trie}(p^1), \text{Trie}(p^2), \ldots, \text{Trie}(p^{n-1}).$$

However, we do not keep all these intermediate suffix tries in memory, because overall it would take a quadratic time. Rather, we transform the current tree, and its successive values are exactly $\text{Trie}(p^1), \text{Trie}(p^2), \ldots, \text{Trie}(p^n)$. Doing so we also require that the construction takes linear time (on fixed alphabets).

Let us examine closely how the sequence of uncompacted trees is constructed in Figure 4.5. The nodes corresponding to suffixes of the current text $p^i$ are shaded. Let $v_k$ be the suffix of length $k$ of the current prefix $p^i$ of the text. Identify $v_k$ with its corresponding node in the tree. The nodes $v_k$ are the essential nodes. In fact, additions to the tree (to make it grow up) are "essentially" created at such essential nodes. Consider the sequence $v_i, v_{i-1}, \ldots, v_0$ of suffixes of $p^i$ in decreasing order of their length. Compare such sequences in trees for the prefixes of the word ababba, see Figure 4.5.

The basic property of the sequence of trees $\text{Trie}(p^i)$ is related to the way they grow. It is easily described with the sequence of essential nodes. Let $a_i = \text{text}[i]$ and let $v_i$ be the first node in the sequence $v_{i-1}, v_{i-2}, \ldots, v_0$ of essential nodes, such that child$(v_j, a_i)$ exists. Then, the tree $\text{Trie}(p^i)$ result from $\text{Trie}(p^{i-1})$ by adding a new outgoing edge labeled $a_i$, to each of the
Figure 4.4: One iteration in the construction of Trie(text). Thick edges are created at this step as well as new links for the corresponding new sons.
nodes \(v_{i-1}, v_{i-2}, \ldots, v_j-1\) simultaneously creating a new son (see Figure 4.4). If there is no such node \(v_j\), then a new outgoing edge labeled \(a_i\) is added to each of the nodes \(v_{i-1}, v_{i-2}, \ldots, v_0\).

The sequence of essential nodes can be generated by iteratively taking suffix links from its first element, the leaf for which the value is the prefix read so far. The sequence of essential nodes is given by:

\[(v_i, v_{i-1}, \ldots, v_0) = (v_i, \text{suf}[v_i], \text{suf}^2[v_i], \ldots, \text{suf}^i[v_i]).\]

Using this observation we obtain the algorithm \textit{on-line-trie}.

\textbf{Theorem 4.1} The algorithm \textit{on-line-trie} builds the tree \(\text{Trie}(p)\) of suffixes of text \textit{on-line} in time proportional to the size of \(\text{Trie}(p)\).

\textit{Proof.} The complexity results from the fact that the work performed in one iteration is proportional to the number of created edges. \(\square\)

\begin{algorithm}
\begin{algorithmic}
  \STATE \textbf{Algorithm} \textit{on-line-trie};
  \STATE create the two-node tree \(\text{Trie}(\text{text}[1])\) with suffix links;
  \FOR {\(i := 2\) \text{ to } \(n\)} \text{ do begin}
    \STATE \(a_i := \text{text}[i];\)
    \STATE \(v_{i-1} := \text{deepest leaf of } \text{Trie}(p^{i-1});\)
    \STATE \(k := \text{min}\{k : \text{son}(\text{suf}^k[v_{i-1}], a_i) \neq \text{nil}\};\)
    \STATE create \(a_i\)-sons for \(v_{i-1}, \text{suf}[v_{i-1}], \ldots, \text{suf}^k[v_{i-1}]\),
    \STATE \, and new suffix links for them (see Figure 4.4);
  \ENDFOR
\end{algorithmic}
\end{algorithm}

\section{Ukkonen algorithm}

Ukkonen algorithm can be viewed as a “compacted version” of the algorithm \textit{on-line-trie}. The basic point is that certain nodes of the trie do not exist explicitly in the suffix tree (after chain compaction). Indeed every node of the trie can be treated as an \textit{implicit node} in the suffix tree. If this node corresponds to a node of the suffix tree then we say that such an implicit node is \textit{real}. More formally: a pair \((v, \alpha)\) is an implicit node in \(T\) if \(v\) is a node of \(T\) and \(\alpha\) is a (proper) prefix of the label of an edge from \(v\) to a son of it. The implicit node \((v, \alpha)\) is said to be a “\textit{real}” node if \(\alpha\) is the empty word.

\textbf{Observation.} The sequence of suffix trees produced by Ukkonen algorithm will differ slightly from our definition. We keep in the tree only the deepest
internal essential node. Other essential nodes of out-degree one are not kept explicitly. However at the end of the algorithm we can add a special end-marker and this automatically will create all internal essential nodes. There is another important implementation detail. If the node \( v \) is a leaf, then there is no need to extend the edge coming from its father by a single symbol. If the label of the edge is a pair of positions \( (l, r) \), then, after updating it, it should be \( (l, r + 1) \). We can omit such updates by setting \( r \) to * for all leaves. This *-symbol is automatically understood as the last scanned position \( i \) of the pattern. Doing so reduces the amount of work involved at each iteration of the algorithm.

The \( i \)-th iteration executed in the previous algorithm can be adapted to work on a compact suffix tree \( ST(p^{i-1}) \). Each node of \( Trie(p^{i-1}) \) is treated as an implicit node. The new algorithm simulates the version on-line-trie.

If \( v_i, v_{i-1}, \ldots, v_0 \) is the sequence of essential nodes, and if we know that \( v_i, v_{i-1}, \ldots, v_k \) are leaves, then we can skip processing them because this is done automatically by the * trick. We thus start processing essential nodes from \( v_{k-1} \). In the algorithm we call it \( v \), the working node. This node is indicated in Figure 4.8. In other words the working node is the deepest internal essential node (corresponding to one suffix of the actual text). We do not maintain the set of essential nodes at lower levels. Another crucial concept is that of implicit suffix links, denoted by \( insuf \). If \( u \) is an implicit node then its suffix link can point to a non-real implicit node \( w \), see Figure 4.6.

The computation of such a link is done by following the suffix link of the real-node father of \( u \), then following down by the path having the same label as from father\( (u) \) to \( u \), see Figure 4.6. Creation of new edges is illustrated in Figure 4.9 and 4.10.

**Theorem 4.2** Ukkonen algorithm builds the compressed tree \( ST(text) \) in an on-line manner. It works in linear time (on a fixed alphabet).

**Proof.** The correctness follows from the correctness of the version working on an uncompacted tree. The new algorithm is just a simulation of it. To prove the \( O(|text|) \) time bound, it is sufficient to prove that the total work is proportional to the size of \( ST(text) \), which is linear. The work is proportional to the work performed by processing all working paths (paths of implicit suffix links needed to go from one working node to the working node of the next iteration). The cost of processing one working path is proportional to the decrease of distance between the working node and its father, plus some additional additive constant, see Figure 4.7. This distance is defined in terms of number of symbols from the implicit node \( (v, \alpha) \) to its real father. If the working node is real itself then this distance is zero. On the other hand, the length of \( \alpha \) is increased by at most one per iteration. Hence, the total increase
of $\alpha$ is linear and, consequently, the total number of length reductions of $\alpha$'s
is linear. $\square$

**Algorithm Ukkonen;**

create the two-node tree $Trie(\text{text}[1])$ with suffix links;

\{ $v$ is the *working node* \}

$v := \text{root}$;

for $i := 2$ to $m$ do begin

$a_i := \text{text}[i]$;

if $son(v,a_i) \neq \text{nil}$ then $v := son(v,a_i)$

else begin

$k := \min\{k : son(imsuf^k[v],a_i) \neq \text{nil}\}$;

create $a_i$-sons for $v$, $imsuf[v], \ldots, imsuf^{k-1}[v]$,

and new $imsuf$ links for new internal nodes;

$v := son(imsuf^{k-1}(v),a_i)$;

\{ $v$ is the deepest internal essential node of $Trie(p^{i-1})$ \}

end

**Remark.** If we want, at the end of the execution, to have a leaf corresponding
to each suffix, then one extra stage of Ukkonen algorithm to manage an end-
marker can do it, see Figure 4.10.

**Bibliographic notes**

The on-line algorithm that builds a compact suffix tree presented in the chapter has been discovered by Ukkonen [U 92]. The method is similar to the
construction of suffix DAWG's presented in Section 6.2.
Figure 4.5: The history of the computation of the algorithm on-line-trie. The sequence of suffix tries Trie($p^1$), Trie($p^2$), ..., Trie($p^6$) for the text $p = abaabb$. Essential nodes are shaded. Edges created at the current step are thick. At each step $i$ of the algorithm on-line-trie we follow a path of suffix links from the deepest node to the first node having an $a_i$-son.
Figure 4.6: Computation of the implicit suffix link for an implicit node $u$, see also Figure 4.8.

Figure 4.7: The working path in Ukkonen algorithm. Its cost can be charged to the decrease of the distance (measured in symbols) between the working node and its explicit father (the quantity $p - q$), plus an additive constant.
Figure 4.8: The tree during one stage of Ukkonen algorithm, the working node is shaded. The path of implicit suffix links is indicated by arrows. When letter $b$ is added to the text, five new edges labelled by $b$ are to be created.
4.3. UKKONEN ALGORITHM

Figure 4.9: After extending the text by the letter $b$, several new edges, indicated by thick lines, are created.

Figure 4.10: After extending the previous text by the end-marker $\#$ several new nodes (thick circles) are added to the tree. In this tree, each suffix corresponds to exactly one leaf.