Linear Pattern Matching of Repeated Substrings

Alejandro López-Ortiz

Department of Computer Science
University of Waterloo
Waterloo, Ont. N2L 3G1 Canada

1 Introduction

In 1970, Knuth, Morris, and Pratt proposed their famous linear-time pattern matching algorithm for two strings. Their algorithm was derived from a result of Cook that 2-way deterministic pushdown languages are recognizable on a RAM in linear time [Co71]. In 1973, Weiner [PeWe73] presented a very original algorithm that performs linear time recognition of repeated instances of a substring in a string. Weiner’s approach to this problem was as important as the solution to the problem itself. The relevance of his work was immediately appreciated. The result was announced in October, 1973 at what was then SWAT (now FOCS) and some selected ideas made their way to Section 9.5 of the first edition of Aho, Hopcroft and Ullman’s textbook [AHU74], published half a year later.

Unfortunately, Weiner’s paper may be difficult for modern readers. Familiar objects such as trees and other data structures are described using notation drawn from automata theory. Typographical errors and overloading of terms contribute to the difficulty. This paper attempts to explain Weiner’s result in a more accessible manner.

2 Basic Definitions and Notation

Let $X = x_1x_2 \ldots x_n$ be a string over an alphabet $\Sigma$ and $\$ a symbol not in the alphabet. Thus, the end of every string can be determined unambiguously by adding the symbol $\$ at the end.

**Definition 1** By $X_{i:j}$ we denote the substring $x_i x_{i+1} \ldots x_j$ of $X$. The string $X'$ denotes the substring $x_i x_{i+1} \ldots x_n \$$, i.e. the substring of $X$ starting in position $i$ and ending with $\$. 

Definition 2 A trie is a labeled tree $T$ such that for each interior vertex $v$ in $T$, the edges leaving $v$ have distinct labels in an alphabet $\Sigma$.

Definition 3 If $\sigma_j$ labels the edge from node $q$ to node $t$ in a trie, we say that $t$ is the $\sigma_j$-child of $q$.

Similarly, we define the concept of successor. A node $t$ is the $\omega$-successor of $q$ (for $\omega = \sigma_1\sigma_2\ldots\sigma_j$), if there is a descending path in the tree labeled $\sigma_1\sigma_2\ldots\sigma_j$ from $q$ to $t$. Formally:

Definition 4 It is said that $t$ is the $\omega$-successor of $q$ is $t$ if either $\omega = \lambda$ (the empty string) and $t = q$ or $u$ is the $\sigma_1$-child of $q$ and $t$ is the $\sigma_2\ldots\sigma_j$ successor of $u$, and $\omega = \sigma_1\sigma_2\ldots\sigma_j$.

The word associated with a node $q$ is the concatenation of the symbols down the path from the root of the tree to the node itself. We denote this word as $W(q)$. Clearly, if $r$ is the root of the tree, then for any node $q$, $q$ is the $W(q)$ successor of $r$.

Definition 5 A bi-tree is a pair of tries $P$ and $S$, sharing the same set of nodes (but not edges), such that the depth of a node is the same on both tries. Further, the edges in $P$ and $S$ are labeled in such a way that $W_P(q) = W_S(q)^R$ for all nodes $q$. (See figure 2)

Observation 1 The same node is the root of both $P$ and $S$. The root of $P$ and $S$ are the same node.

It is not immediately clear that bi-trees can be constructed, given the set of constrains. If the nodes need not be shared, then it would be simple to have two trees, in one of which strings are in reverse order. The following sections characterizes in which cases a bi-tree exists.

3 Basic Properties of Tries

The following lemma states that a bi-tree can be constructed from a position tree if and only if all suffixes of a node’s word are words themselves in the $P$ tree.

Lemma 1 Given a trie $P$, it is possible to construct an adjoining trie $S$ to obtain a bi-tree if and only if for every node $q$ with $W(q) = \sigma_1\ldots\sigma_n$ there exists a node $t$ such that $W(t) = \sigma_2\ldots\sigma_n$. 
Proof. First, we assume that a bi-tree can be constructed on $P$ and show that the suffix that drops the first letter in a word, i.e. $WP(q)_{2:k}$ for $k = |WP(q)|$ is a word in the tree.

From the definition of the bi-tree containing $P$, we know that for all nodes $q$, $WP(q)R$ is a word in the adjoining $S$ tree of the bi-tree. Moreover, since $WP(q)R$ is a path in the $S$ tree, then there is a node $t$ with $[WP(q)]_{1:k-1}^R$ as a word in $S$ (i.e. $t$ is the immediate ancestor of $q$ in $S$). But from the definition of bi-tree this implies that $q$ has as word in $P$ the reverse of the word in $S$, namely $WP(t) = WS(t)1:k-1$ as required.

Now assume that for every word $!$ in the $P$-tree $!_{2:k}$ is also in the $P$-tree. We construct an adjoining tree $S$ by induction on the length of words as follows:

Basis of Induction. Since $W(q) = W(q)R$ for all words of length one, we add $S$ edges from the root to its children, labeled with the same character as the respective edge in $P$.

Induction Step. Assume that all nodes with words of length at most $k-1$ are in the $S$ tree and consider a node $q$ with a word of length $k$. Let $t$ be the node in $P$ such that $W(t) = W(q)_{2:k}$. Then, by induction hypothesis $t$ is in $S$ and $WS(t) = [W(q)]_{2:k}^R$.

In $S$, we add the edge from $t$ to $q$ and label it $W(q)_{1:1}$. It follows then, from the recursive definition of $W$ descendants, that $WS(q) = WS(t)W(q)_{1:1} = [W(q)]_{2:k}^R W(q)_{1:1} = W(q)^R$ as required. \qed

4 Position Trees

Definition 6 We say that a substring $U$ identifies position $i$ in string $X$ if $X = YUZ$, $|Y| = i - 1$, and $X$ cannot be written as $Y'UZ'$ unless $Y' = Y$. That is, the only occurrence of $U$ within $X$ begins at position $i$.

Observation 2 A string $X$ terminated with the $\$ character has at least one identifier for each position $i$, namely $X^i = x_ix_{i+1}\ldots x_n\$.

Definition 7 The substring identifier for position $i$ in $X$, denoted $Di$, is the shortest string that identifies position $i$ in $X\$.

Definition 8 A position tree for a string $X\$ = $x_1\ldots x_{n+1}$, is a trie with the leaves labeled $1, 2, \ldots n + 1$ such that the word associated with leaf $i$, is the substring identifier of position $i$, i.e., $W(leaf_i) = Di$. (See figure 1)
Lemma 2 Let \( D_i \) be the substring identifier for position \( i \) of \( X \). Then \( |D_i| \leq |D_{i+1}| + 1 \).

Proof. Let \( j = |D_{i+1}| \). If \( D_i \) has length greater than \( j + 1 \) it implies that the string \( X_{i:i+j} \) does not uniquely identify position \( i \), i.e. there is another position \( k \) such that \( X_{k:k+j} = X_{i:i+j} \). This contradicts the fact that \( D_i = X_{i:i+j} \) is a unique string in \( X \).

Corollary 1 No substring identifier is a proper prefix of another.

Observation 3 All proper prefixes of a substring identifier \( D_i \) are prefixes of a substring identifier \( D_j \), for \( i \neq j \). (Otherwise, the prefix would be a shorter string identifying the position).

Observation 4 All leaf nodes of a position tree have siblings. (This follows from observation 3).

Theorem 1 For every string \( X \) over \( \Sigma \), there exists a bi-tree \( B \) such that the \( P \) component of \( B \) is the position tree of \( X \).

Proof. We will show that the position tree \( P \) of the substring satisfies the conditions of Lemma 1, from which the theorem follows.

Let \( q \) be a node in \( P \). From the definition of \( P \), \( W(q) \) is a prefix of a substring identifier \( D_i \) for some \( i \). Lemma 2 implies that \( D_{i+1} \) is at least as long as \( D_i \). It follows then that there exists a node \( t \) in the path of \( D_{i+1} \) in \( P \) such that \( x_i W(t) = W(q) \). 

\[ \]
5 Construction of a Position tree.

We construct a position tree in an iterative process, parsing the input string \( X \) from right to left. Because of this we need to introduce some notation for shortened strings.

**Definition 9** Let \( X^i \) be as in definition 1. Let \( P^i \) denote the position tree of \( X^i \). \( D^i_j \) denotes the substring identifier of position \( j \) in \( X^i \) for \( i \leq j \), where position numbers are as in \( X \).

**Observation 5** Let \( l \) be the position in \( X^{i+1} \) such that \( D^i_{l+1} \) and \( D^i_i \) have the longest common prefix. Then \( D^i_j \equiv D^{i+1}_j \) for all \( i < j \leq n+1 \) and \( j \neq l \).

In other words, the position trees of strings \( X^i \) and \( X^{i+1} \) differ only in two substrings at the most, namely \( D^i_i \) which was not in \( P^{i+1} \) to start with, and \( D^i_l \subseteq D^{i+1}_l \) which might need to be extended, so as to distinguish it from \( D^i_i \).

From this a straightforward algorithm for the construction of position trees can be deduced: Build \( P^n, P^{n-1}, \ldots, P_1 = P \) in that order by means of inserting the new prefix of \( X^i \) in the trie for \( i = n \ldots 1 \).

Trivially such an algorithm would take at least linear time on the size of the tree. Unfortunately, a position tree may be quadratic on the size of the string (viz. strings of the form \( 0^m1^m0^m1^m\$ \)). Further, even on trees which are not of quadratic size, this simple algorithm may require \( \Omega(n^2) \) to build the tree (viz. strings of the form \( 0^{100}Y \) where \( Y \in \Sigma^m \)).

On the other hand, any position tree has exactly \( n \) leaves since each leaf corresponds to a substring identifier, and there are only \( n \) positions to be identified. With this in mind, for an arbitrary position tree, is possible to compress each path with no branching into a single edge obtaining a compressed tree of \( \Theta(n) \) size. Even on such a compressed tree, the algorithm described above takes \( \Omega(n \log n) \) time in the worst case.

6 Construction of a Bi-tree

Now consider the construction of a bi-tree. A bi-tree can be constructed by adding, in a straightforward manner \( S \)-edges during the construction of the position tree described above. Such method does not takes advantage of information encoded by the \( S \)-edges.

The following observations and enhancements allows us to propose an algorithm that will be the basis of a linear time repeated pattern matching algorithm.
Observation 6 Suppose that the symbol $x_i$ occurs within $X^{i+1}$. Let $Y$ be the longest prefix of $D_i^{i+1}$ such that $x_iY$ occurs elsewhere in $X^{i+1}$, say position $j$. Then the substring identifier of position $i$ and of position $j$ is a string starting with $x_i$ followed by $Y$ and terminated with the extra character following $Y$ in each position, i.e. $D_i^i = x_i Y x_{i+j+1} = X_i; Y | +i+1$ and $D_j^j = x_j Y x_{j+i+1} = X_j; Y | +j+1$.

For the construction of a bi-tree associated to a string $Y$, nodes in the bi-tree are labeled with an array of integers with an entry for each letter of the alphabet. The $\sigma_j$-entry of the array contains the index $i$ if the word $\sigma_j W(q)$ is a substring of $Y$ and $W(q)$ is a prefix of the substring identifier of position $i$ (in those cases where there are two or more strings with the conditions above, the largest index is selected). What makes this algorithm superior over the standard construction of position trees is that edges in the $S$ tree allow us to move between different branches of $P$ in one step without requiring to backtrack all the way up to a common ancestor in $P$ of the two branches and then going downwards from it.

Algorithm for the construction of the bi-tree of $X^i$ given the bi-tree of $X^{i+1}$ and a pointer to leaf $q_{i+1}$ identifying position $i + 1$. Each node is a record containing pointers to its parents and children in $S$ and $P$, an indicator of its depth in tree, and a label vector as described above.

1. Each node is a record containing pointers to its parents and children in $S$ and $P$, an indicator of its depth in tree, and a label vector as described above.

2. Let $q_{i+1}$ be the node in $B_{i+1}$ such that $W(q_{i+1})$ is the substring identifier of position $i + 1$ in $X^{i+1}$.

3. Let $t \leftarrow q_{i+1}$.

4. Repeat until $t$ has an $x_i$-child in $S$ or $t$ is the root:
   
   (a) First update the labeling
       If the $x_i$ label of $t$ is uninitialized, then set its value to $i + 1$.
       Else if $b$ is uninitialized, let $b$ be the value of the $x_i$-th label of $t$.
       Now $b$ holds the starting position of the longest substring $Y$ in $X^{i+1}$
       such that $Y$ is a prefix of $X^{i+1}$; $x_{b-1} = x_i$; and $b \neq i + 1$.
   
   (b) Then move upwards in the tree.
       Let $t \leftarrow parent_P(t)$.

5. If $t$ has no $x_i$-child in $S$ this implies that $x_i$ is not in $X^{i+1}$, and thus $t$ is the root $r$. In this case add the $x_i$-child to $r$ in $P$ which will also be the $x_i$-child of $r$ in $S$. Set the labels accordingly. Return $q_i \equiv child_P(r, x_i)$. 

6.
6. Let \( q \) be the \( x_i \)-child in \( S \) of \( t \) and let \( d \) be the depth of \( t \).

Descend from \( t \) down the tree \( P \), while creating a parallel path downwards from \( q \) in \( P \) until \( x_{i+1+d} \neq x_{b+d} \) as follows:

(a) If the \( x_{i+1+d} \)-child of \( t \) is not labeled \( b \) in its \( x_i \) position then we have found the first mismatch between \( X_{i+1} \) and \( X^b \) and thus positions \( i \) and \( b-1 \) are about to be substring identified. Insert the \( x_{i+1+d} \) and \( x_{b+d} \)-children of \( q \). Let \( q', q'' \) be such children. Further, \( t \) must also have \( x_{i+1+d} \) and \( x_{b+d} \)-children, \( t' \) and \( t'' \). Make \( q' \) (\( q'' \)) the \( x_i \)-child of \( t' \) (\( t'' \)) in \( S \). RETURN \( q_i \equiv q' \).

(b) Else note that \( q \) has no \( x_{b+d} \)-child, otherwise \( q \) would be the \( x_i \)-child in \( S \) of a proper descendant of \( t \) down the \( t \rightarrow q_{i+1} \) path, but by construction, \( t \) was the deepest such node in the path. Insert the \( x_{b+d} \)-child for \( q \); set it’s label accordingly, and let this new node be the \( x_i \)-child of \( t \) in tree \( S \). Let \( q \leftarrow \text{child}_P(q, x_{i+1+d}) \) and \( t \leftarrow \text{child}_P(t, x_{i+d}) \). Continue the descent.

Notice that this algorithm runs in linear time on the number of nodes of the bi-tree, since for each edge traversed upwards, an edge is eventually added while traversing downwards. In this sense this algorithm is an improvement over the position tree construction, where even on a linear size tree the algorithm could take quadratic time.

7 Compact bi-trees

Because of the observation that there are only \( n \) leaves in a bi-tree, and the fact that now we know how to construct a bi-tree in time linear in its size, it is natural to ask whether a compacted version of a bi-tree could be constructed in linear time.

**Definition 10** A compact position tree is a position tree where all non-branching paths of length greater than two are compressed into two edges. The label of the first edge is the same as the label of the first edge in the original path and the second edge, which represents all the other compressed edges, is labeled \( * \).

**Observation 7** A node of the position tree is in the corresponding compact position tree if and only if either it has outdegree bigger than or equal to 2 or its parent does.
Figure 3: Case 5(a) (New edges in boldface)

Figure 4: Case 5(d)

**Definition 11** A compact bi-tree $C$ is a bi-tree constructed over the compact position tree of string $X$. All the edges of the $S$ tree in the uncompacted bi-tree of $X$ whose end nodes have not been compacted out form the compacted $S$ tree in $C$. Edges in $S$ whose parent is in the compacted tree but their child is not are marked * in the compact bi-tree and are left dangling.

**Lemma 3** If a node $q$ is in the compact bi-tree then its $S$-parent $t$ is in the compact bi-tree as well.

To prove this lemma we need the following observation:

**Observation 8** A node $q$ in an uncompacted bi-tree has outdegree less than or equal to that of its $S$ parent $t$. (This follows from the same argument that proves lemma 2).

**Proof** (Lemma 3). If $q$ is in the compacted bi-tree, because of observation 7, either it or its parent has outdegree $\geq 2$. In the first case, by observation 8, $t$ has outdegree $\geq 2$ and thus it appears in the compacted bi-tree. For the second case, the $P$ parent $u$ of $q$ has outdegree $\geq 2$; which implies that the $S$ parent of $u$ is in the compact bi-tree and has outdegree $\geq 2$. Now, the $S$ parent of $u$ is in the bi-tree with outdegree $\geq 2$ and is the $P$ parent of $t$ thus, by observation 7, $t$ is in the compact bi-tree as well. □

Algorithm for the construction of the compact bi-tree of $X^i$ given the bi-tree of $X^{i+1}$ and a pointer to leaf $q_{i+1}$ identifying position $i + 1$:
1. Let $t \leftarrow q_{i+1}$.

2. While the $x_i$ label of $t$ is uninitialized set its value to $i + 1$ and if $t$ is not the root then let $t \leftarrow \text{parent}_P(t)$. 

3. If $t$ is the root insert its $x_i$-child in $P$. This node should also be set to be the $x_i$-child of the root in $S$. Let $q_i \leftarrow \text{child}_P(t, x_i)$. RETURN $q_i$.

4. Let $l$ be the label of $x_i$ in $t$ and $d$ its depth. Let $q \leftarrow \text{child}_P(t, x_{i+d})$. (Notice that $x_{i+1+d}$ is necessarily different from $x_{i+d}$). At this point, the algorithm has identified the longest string in $X^{i+1}$ which shares a prefix with $X^i$. This step corresponds to step 4.a in the construction of a non-compacted bi-tree.

5. There are six possible cases (see figures 3 & 4):

   (a) Both $q$ and $t$ have an $x_i$-child in $S$.
   Let $u \leftarrow \text{child}_S(t, x_i)$.

   (b) The node $q$ has an $x_i$-child but $t$ has an $x_i$-edge in $S$ marked * and dangling.
   Let $v$ be the $x_i$-child of $q$ in $S$. In this case the edge in $P$ between $v$ and its parent in $P$ is labeled *. Insert a new node $u$ dividing this edge in two. The upper part remains marked * if the difference in depth is bigger than one, else is marked $x_{i+1}-1$. The lower part is marked $x_{i+d}$. Make $u$ the $x_i$-child of $t$ in $S$.

   (c) The node $q$ has a dangling $x_i$-edge in $S$ but $t$ has an $x_i$-child in $S$.
   Let $u \leftarrow \text{child}_S(t, x_i)$. Thus, there is only one edge out of $u$ in $P$ and is marked *. Divide this edge in two. Let $v$ be the new node in between. Label the upper half of the edge $x_{i+d}$ and the lower part * (or $x_{i+d+1}$, if the height difference is less than two). Make $v$ the $x_i$-child of $q$ in $S$.

   (d) Both $q$ and $t$ have dangling $x_i$-edges in $S$.
   Let $w \leftarrow t$. While $w$ has a dangling $x_i$ edge do $w \leftarrow \text{parent}_P(w)$.
   Now, as in 5.c, $w$ has an $x_i$-child in $S$, say $u'$, and its child in $P$ has a dangling $x_i$-edge in $S$. Thus, there is only one edge out of $u'$ in $P$ and must be marked *. Divide this edge in two. Upper part is labeled *, lower part is labeled $x_{i+d}$, the node in the middle is $u$.

   (e) Node $q$ has no $x_i$-edge in $S$ but $t$ has an $x_i$-child in $S$.
   Let $u \leftarrow \text{child}_S(t, x_i)$. Then, $u$ is a leaf. Insert the $x_{i+d}$-child of $u$ in $P$.

   (f) Both $q$ and $t$ have no $x_i$-edge in $S$.
   Let $w \leftarrow t$. While $w$ has no $x_i$-child mark the $x_i$-edge of $w$ with *
and leave it dangling; let \( w \leftarrow \text{parent}_P(w) \).

Let \( u' \) be the \( x_i \)-child of \( w \) in \( S \). Insert a child \( u \) of \( u' \) labeled \( * \).

Insert the \( x_{i+d} \)-child of \( u \) in \( P \).

6. Insert the \( x_{i+1+d} \)-child of \( u \) in \( P \). Let \( q_i \) be that child. Make \( q_i \) the \( x_i \)-child in \( S \) of the \( x_{i+1+d} \)-child of \( t \) in \( P \). Return \( q_i \).

Let \( h_i \) be the height of the node \( q_i \) in the compact tree corresponding to \( X^i \). Each iteration of this algorithm takes time proportional to \( (h_{i+1} - h_i + 6) \). This can be seen from the fact that the insertion algorithm consists of a sequence of constant time operations being performed while traversing the tree, starting from node \( q_{i+1} \) and going up to a node \( t \) or \( w \) and then proceeding downwards to node \( q_i \). The node \( q_i \) is at most three edges down from \( t \) or \( w \), thus giving the desired bound.

Total time for the construction of a compact bi-tree is then proportional to

\[
\sum_{i=n}^{1} (h_{i+1} - h_i + 6) = 6n - h_1 = O(n)
\]

which proves that the compact bi-tree can be constructed in linear time. It is important to note that the constant number of operations is proportional to the alphabet size, thus the algorithm takes time \( O(nk) \) where \( k = |\Sigma| \).

In 1976, McCreight proposed a more space efficient algorithm, which is now often used instead of Weiner’s construction. The ideas behind both algorithms are the same, although the approaches differ somewhat [McC76].

References


[PeWe73] P. Weiner, Linear Pattern Matching Algorithms. 14th Annual Symposium on Switching and Automata Theory, IEEE.