# A Simple Tree Pattern-Matching Algorithm

*Hsiao-Tzu Lu, Wuu Yang*
Department of Computer and Information Science,
National Chiao Tung University, Hsinchu, Taiwan, ROC.
E-mail: wuuyang@cis.nctu.edu.tw

*Abstract*— **Tree pattern matching occurs as a crucial step in a number of programming tasks. We propose a new algorithm to solve the tree pattern-matching problem. The algorithm may be viewed as an extension of the Knuth-Morris-Pratt string-matching algorithm to the tree pattern-matching problem. In the new algorithm, the times of each node in the subject tree needs to be traversed is bounded by their level. Therefore, the time complexity of the simple tree pattern-matching is bounded by $O(n \times \log n)$, where $n$ is the number of nodes in the subject tree. The worst case occurs when the frequency of the same content of the pattern's root in the subject tree is high. But we need an extra preprocessing time of the pattern. The time complexity of the pattern preprocessing is bounded by $O(m \times \log m)$, where $m$ is the number of nodes in the pattern. The worst case occurs similarly when the frequency of the same content of the root in the pattern is high. By using indirection, the space complexity will be down to $O(n + m)$.**

## I. Introduction

Tree pattern matching is an interesting special problem which occurs as a crucial step in a number of programming task. It occurs frequently in the context of tree replacement systems and has applications in different areas of computer science, including automatic implementation of abstract data types, code optimization, automatic proof systems, syntax-directed compilation and evaluations for programming languages such as LISP [3] [4]. And the tree replacement approach is very convenient for producing interpreters for implementing experimental languages.

Tree pattern matching is an extension to the problem of pattern matching in strings. We extend the Knuth-Morris-Pratt string-matching algorithm to tree patterns. Similarly, we need to preprocess the pattern first in order to speed up the matching method and reduce the number of times that nodes need to be compared. And in our method, the sibling nodes in the tree are ordered (i.e. the subject tree and the pattern are ordered trees). We take Fig. 1 for example. We cannot find out a full match of the pattern of Fig. 1(a) in the subject tree of Fig. 1(b), though the pattern in Fig. 1(a) and the subject tree in Fig. 1(b) are similar.

## II. Existing Approaches to Tree Pattern Matching

Hoffmann and O'Donnell [3] proposed several algorithms to solve the tree pattern-matching problem. Their algorithms may be classified into two categories: one is a bottom-up approach, and the other is a top-down approach. While the bottom-up method generalizes string matching, the top-down method reduces tree matching to a string-matching problem.

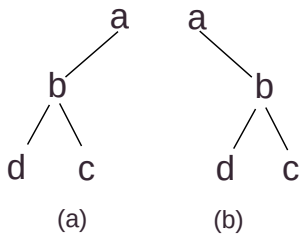


Fig. 1. (a)The pattern. (b)The subject tree.

The key idea of the bottom-up matching algorithms in [3] is to find, at each point in the subject tree, all patterns and all parts of patterns which match at this point. When the pattern forest F is simple (a simple pattern forest is a set of trees that contains no independent subtrees), Hoffmann and O'Donnell can construct the subsumption graph for it and set the tables via the subsumption graph. We can see an example of the subsumption graph in Fig. 2.

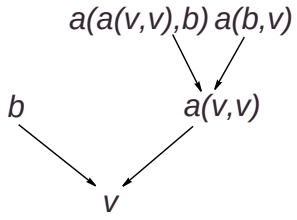


Fig. 2. An example of the immediate subsumption graph.

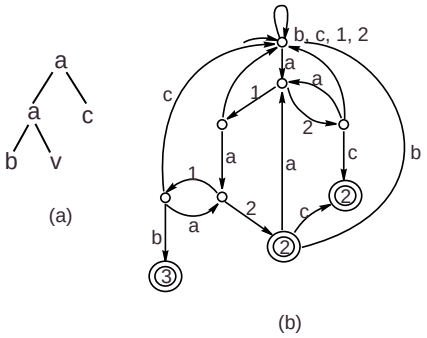


Fig. 3. (a)A pattern. (b)The associated matching automaton.

The top-down matching algorithm uses a matching automaton, and we can see an example in Fig. 3. In Fig. 3, the automaton is associated with the pattern of the example in Fig. 3(a). Accepting states are circled twice and are

labeled with the length of the accepted path string [3]. The top-down matching algorithm is slower than the bottom-up matching algorithms, but has shorter preprocessing time.

Rational patterns are used to specify recognizable tree languages. Simon [4] proposed an efficient tree pattern matching algorithm and applied it on nets. The algorithm solves the tree pattern matching in $O(|p| \cdot |t|)$ steps where there is some match of rational pattern $p$ in $t$.

## III. Our Approach

### A. Extension of the KMP String-Matching Algorithm

The basic concept of the KMP string-matching algorithm are discussed in [5] and [6]. The auxiliary table *next* in the KMP string-matching algorithm may be shown pictorially. Take TABLE I for example, Fig. 4 is also a representation and the arcs represent the *next* pointers.

One of the basic concept in the KMP string-matching algorithm is precomputation of the *shifts*, as in tree pattern-matching problem, we can preprocess the pattern and precompute the *shifts* which is called *back* in our algorithm. The difference is that we shift the pattern not only to move the pattern horizontally right or left but also move it vertically up or down. We show the complete algorithm in the following sections.

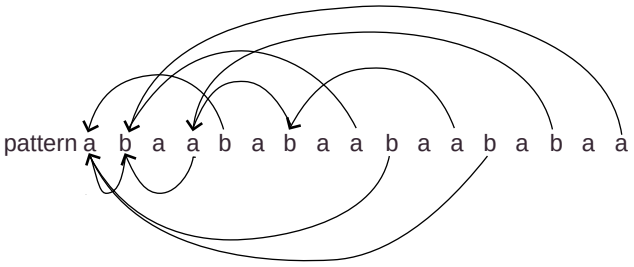


Fig. 4. The other presentation of TABLE I.

### B. A Simple Tree Pattern-Matching Algorithm

#### B.1 Presentation

The *level* of a node in the tree is defined as follows. The root is at level 1. If a node is at level $l$, then its children are at level $l+1$ [2]. Fig. 5 shows the levels of all nodes in that tree. The *height* of a tree is defined to be the maximum level of any node in the tree [2].

We use a numbering scheme to represent a binary tree in memory. Suppose we number the nodes in a complete binary tree starting with the root on level 1, continuing with the nodes on level 2, and so on. Nodes on any level are numbered from left to right. The numbering representation can clearly be used for all binary trees, though in most cases there will be a lot of unutilized space [2]. We can see the number of each node of the tree in Fig. 5. The number labeled in the circle is the index of a node in our numbering presentation.

Since the nodes are numbered from 1 to $n$, we can use a one-dimensional array to store the nodes. Using Lemma 2 we can easily determine where the parent, the left child, and the right child of any node are in the binary tree [2].

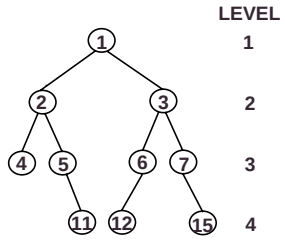


Fig. 5. A sample tree.

**Lemma 1** [The maximum number of nodes in a binary tree] [1] [2]
*(1)* The maximum number of nodes on level $i$ of a binary tree is $2^{i-1}$, $i \geq 1$.
*(2)* The maximum number of nodes in a binary tree of height $k$ is $2^k - 1$, $k \geq 1$.

**Lemma 2** If a complete binary tree with $n$ nodes is represented sequentially, then for any node with index $i$, $1 \leq i \leq n$, we have [2]
*(1)* $parent(i)$ is at $\lfloor i/2 \rfloor$ if $i \neq 1$. If $i = 1$, $i$ is at the root and has no parent.
*(2)* $LeftChild(i)$ is at $2i$ if $2i \leq n$. If $2i > n$, then $i$ has no left child.
*(3)* $RightChild(i)$ is at $2i+1$ if $2i+1 \leq n$. If $2i+1 > n$, then $i$ has no right child.

We use the numbering representation in order to show our idea more clearly, and to determine the relationship between nodes more easily. We will show the complete algorithms in section B.2 for pattern preprocessing and in section B.3 for searching the pattern in the subject tree.

#### B.2 Pattern Preprocessing

In our algorithm, we need to record information about the pattern. In particular, we need to record the *content*, *nextnode* and *back* of each node. The *content* field is the content of this node. The content can be a character, a digit, a symbol, etc. The *nextnode* field is the next node's index in the left-to-right breadth-first traversal of the pattern. The *back* field records the index of the node that should be compared subsequently when a mismatch occurs. Consequently, the pattern is stored in an array of the structure $\{content, nextnode, back\}$. The steps of pattern preprocessing are shown in Fig. 6. For convenience, we divide the algorithm into three parts: part I is for initialization, part II is the preprocessing, and part III is the Descend procedure used in part II.

As we initialize the data structure for the pattern, we first initialize the *content* and *nextnode* fields of the pattern. We record the content of each node in the *content* field. The *nextnode* field of a node as mentioned before is the next node's index in the left-to-right breadth-first traversal of the pattern. But the *nextnode* field of the last node in the left-to-right breadth-first traversal is recorded as 0, because there is no other node follows it. The *back* field of a node is initialized as 0 when the *content* field of the

TABLE I

THE SAMPLE TABLE OF THE KMP STRING-MATCHING ALGORITHM.

| j | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| pattern[j] | a | b | a | a | b | a | b | a | a | b | a | a | b | a | b | a | a |
| next[j] | 0 | 1 | 0 | 2 | 1 | 0 | 4 | 0 | 2 | 1 | 0 | 7 | 1 | 0 | 4 | 0 | 2 |

node is not equal to the *content* field of pattern's root, and as 1 otherwise. The initialization steps were shown in Part I of Fig. 6. We will take the example in Fig. 7 as the pattern and show the result of initialization in TABLE II. In TABLE II, some nodes' fields are empty. In fact, those nodes don't exist in the tree. Those nodes waste our space, and we provide a solution to save the space. We will discuss the solution later.

```
Part I
    index:=1;
    base:=1;
    initialize the content and nextnode fields of pattern;
    for ( i := 1; i ≠ 0; i := P[i].nextnode )
    {
        // initialize back field of pattern nodes //
        if P[i].content = P[1].content then P[i].back := 0
        else P[i].back := 1;
    }
Part II
    for ( i := 2; i ≠ 0; i := P[i].nextnode )
    {
        if P[i].content = P[1].content then
        {
            base := i;
            Descend(i,1);
            // find the most repeated region //
        }
    }
Part III
    Descend(i,index)
    // compare for descendants, if the corresponding //
    // nodes' contents are the same, then continue   //
    // comparing to find the most repeated region.   //
    {
        index := P[index].nextnode;
        temp := ⌊log2 index⌋;
        k := 2^temp × (base − 1) + index;
        // compute the corresponding node //
        if P[k].content ≠null then
            if P[k].content ≠ P[index].content
            then if P[k].back < index
                    then P[k].back := index
                    // find the most repeated region //
            else Descend(k,index);
            // continue comparing next node //
    }
```

Fig. 6. Algorithm of pattern preprocessing.

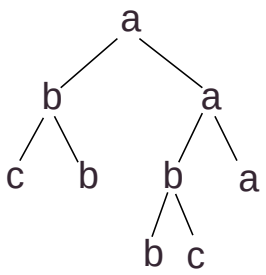


Fig. 7. A sample tree.

TABLE II

THE INITIALIZATION RESULT OF THE PATTERN STRUCTURE IN FIG. 7

| index | content | nextnode | back |
|---|---|---|---|
| **1** | a | 2 | 0 |
| **2** | b | 3 | 1 |
| **3** | a | 4 | 0 |
| **4** | c | 5 | 1 |
| **5** | b | 6 | 1 |
| **6** | b | 7 | 1 |
| **7** | a | 12 | 0 |
| **8** | | | |
| **9** | | | |
| **10** | | | |
| **11** | | | |
| **12** | b | 13 | 1 |
| **13** | c | 0 | 1 |

**D**efinition 1. Let $t$ be a tree, $v$ and $u$ be two nodes of $t$ where $v$ is an ancestor of $u$. We show the tree $t$ pictorially as Fig. 8.
*(1)* The subtree of $t$ with root $v$ is called the $\boldsymbol{v}$-subtree of $t$.
*(2)* The subtree of the $v$-subtree where $u$ is the last node in the left-to-right breadth-first traversal is called the $\boldsymbol{v}$-$\boldsymbol{u}$ part of $t$. The $v$-$u$ part of $t$ is essentially the $v$-subtree of $t$ with all nodes whose indices are greater than $u$'s index removed.

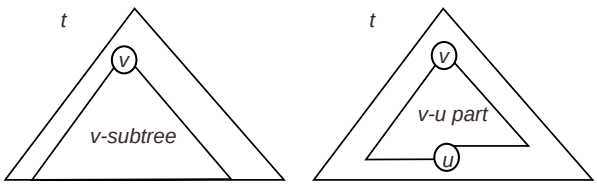


Fig. 8. The $v$-subtree and the $v$-$u$ part.

**D**efinition 2. Let $t$ and $s$ be two trees, $u$ be the node of $t$,

$t[i]$ be the node of $t$ whose index is $i$, and $s[k]$ be the node of $s$ whose index is $k$.
*(1)* When we want to search $t$ in $s$, first we need to find out a node $u$ of $s$ that its content is the same as the root in $t$ and we can possibly find out $t$ in $u$-subtree of $s$. We call the node $u$ the **h**eadnode.
*(2)* When the node $u$ of $s$ is the headnode, the node $u$ of $s$ and the root of $t$ which is $t[1]$ are **c**orresponding nodes. The corresponding node of $t[i]$ is $s[k]$, if and only if $s[k]$ is at the same relative position in $u$-subtree with $t[i]$ in $t$.
For example, if $t[i]$ and $s[k]$ are corresponding nodes, then $t[i]$'s left (or right) child and $s[k]$'s left (or right, respectively) child are corresponding nodes. In Fig. 9, root of tree $t$ corresponds to node $c$ of tree $s$. Therefore, node $c$ of tree $s$ is the corresponding headnode of tree $t$. Node 7 of tree $t$ is at the same relative position with node $n$ of $c$-subtree, so the corresponding node of node 7 is node $n$. The dotted lines in Fig. 9 denote the pairs of corresponding nodes.

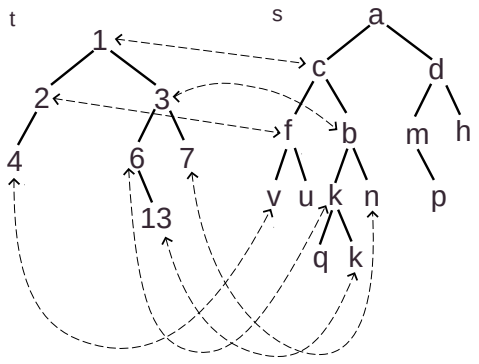


Fig. 9. Corresponding nodes.

After initialization, we need to find out the most appropriate *back* value for each node. We traverse the pattern in the left-to-right, breadth-first order. Only the descendants of the nodes whose contents equal the content of pattern's root need to be compared. The steps of computing *back* are shown in the Descend procedure of Fig. 6.

In the Descend procedure, we need to compute the indices of the corresponding nodes. We will prove the correctness of the index computation in Lemma 3. Besides, we must prove that the value of the *back* field we compute is correct and is the largest, and the proofs are shown in Lemma 4 and Lemma 5.

**Lemma 3.** Let $P$ be tree, $P[base]$ be the node of $P$ whose index is $base$ , $P[index]$ be the node of $P$ whose index is $index$, and $P[k]$ be the node of $P$ whose index is $k$. When the headnode is $P[base]$, $P[index]$'s corresponding node is $P[k]$, where

$$k = 2^{\lfloor \log_2 index \rfloor} \times (base - 1) + index$$

*Proof :*
We want to find corresponding nodes between two subtrees : $P[1]$-subtree and $P[base]$-subtree
$P[1]$'s corresponding node is $P[base]$;
$P[base].content = P[1].content$;
$P[index]$ is a node of $P[1]$-subtree, its corresponding node in $P[base]$-subtree is $P[k]$;
The level of $P[index]$ in the $P[1]$-subtree is $\lfloor \log_2 index \rfloor + 1$;
$\therefore$ $P[index]$'s corresponding node $P[k]$ is also on the same level of $P[base]$-subtree,
and $P[index]$,$P[k]$ are at the same relative position;
$\because$ $index - 2^{\lfloor \log_2 index \rfloor} = k - base \times 2^{\lfloor \log_2 index \rfloor}$
$\therefore$ $k = 2^{\lfloor \log_2 index \rfloor} \times (base - 1) + index$ $\square$
The numbering scheme we use has some characteristics. One is that the leftmost node on level $i$ is numbered as $2^{i-1}$ [1] [2].

**Lemma 4.** Let $P$ be a tree pattern. When $P[k].back = i$, we can find a subtree of $P[base]$-$P[k]$ part of pattern that is isomorphic to $\{P[1]$-$P[i]$ part$-P[i]\}$ where $P[base]$ is the corresponding headnode of $P[1]$-subtree and $base = \frac{k-i}{2^{\lfloor \log_2 i \rfloor}} + 1$.
*Proof :*
We number the nodes in $P[1]$-$P[i]$ part from $node_1$ to $node_{j+1}$ i.e. there are $(j+1)$ nodes in $P[1]$-$P[i]$ part and their corresponding nodes of subtree in $P[base]$-$P[k]$ part from $k^{(1)}$ to $k^{(j+1)}$;
$\therefore$ $P[k^{(1)}]$ represents $P[base], \cdots, P[k^{(j+1)}]$ represents $P[k]$;
Lemma 1 assures that we can find correct corresponding nodes.
And $k^{(i)} = 2^{\lfloor \log_2 node_i \rfloor} \times (base - 1) + node_i \quad \forall i \leq (j+1)$
According to our algorithm, when $P[k].back = i$ :
$P[node_1].content = P[k^{(1)}].content$
$P[node_2].content = P[k^{(2)}].content$
$\vdots$
$P[node_j].content = P[k^{(j)}].content$
But $P[node_{j+1}].content \neq P[k^{j+1}].content$
$\therefore$ $\{P[k^{(1)}], P[k^{(2)}], \ldots, P[k^{(j)}]\}$ is the same structure of tree as $\{P[node_1], P[node_2], \ldots, P[node_j]\}$,
and each corresponding node pair's contents are the same;
$\therefore$ $\{P[k^{(1)}], P[k^{(2)}], \ldots, P[k^{(j)}]\}$ is isomorphic with $\{P[node_1], P[node_2], \ldots, P[node_j]\}$;
$\because$ $\{P[k^{(1)}], P[k^{(2)}], \ldots, P[k^{(j)}]\}$ is a subtree of $P[base]$-$P[k]$ part of the pattern $P$;
$\therefore$ We can find a subtree of $P[base]$-$P[k]$ part isomorphic with $\{P[1]$- $P[i]$ $part$ $-P[i]\}$. $\square$

**Lemma 5.** The value of the *back* field is the largest, i.e., we cannot find any other *back* value that is greater and satisfies Lemma 4.
*Proof :*
Suppose $P[k].back = i$;
Assume that we can find $i'$ which $i' > i$ and $i'$ can satisfy Lemma 4, i.e., we can find a new corresponding root $P[base']$;
Let $L_{i'}$ represents the level of node $P[i']$, $L_i$ represents the level of node $P[i]$.
$\because$ $i' > i$
$\therefore$ $L_{i'} = (2^{\lfloor \log_2 i' \rfloor} + 1) \geq (2^{\lfloor \log_2 i \rfloor} + 1) = L_i$
$\because$ $base = \frac{k-i}{L_i} + 1$
$base' = \frac{k-i'}{L_{i'}} + 1$

$\therefore$ $base' < base$
According our algorithm, $P[base']$ should be compared earlier than $P[base]$
and $P[k].back = i'$;
When $P[base]$ be the headnode,
$\because$ $i < i'$
$\therefore$ $P[k].back$ unchanged.
$\therefore$ We cannot find another $i'$ greater than $i$ and satisfy LEMMA 4. □

TABLE III
THE RESULT OF PREPROCESSING PATTERN IN FIG. 7

| **i**ndex | **c**ontent | **n**extnode | **b**ack |
|---|---|---|---|
| **1** | a | 2 | 0 |
| **2** | b | 3 | 1 |
| **3** | a | 4 | 0 |
| **4** | c | 5 | 1 |
| **5** | b | 6 | 1 |
| **6** | b | 7 | 1 |
| **7** | a | 12 | 0 |
| **8** | | | |
| **9** | | | |
| **10** | | | |
| **11** | | | |
| **12** | b | 13 | 4 |
| **13** | c | 0 | 1 |

We show the result of pattern preprocessing in TABLE III for the example in Fig. 7. Note that the *back* field of the node whose index is 12 is 4, because $\{P[1], P[2], P[3]\}$ is isomorphic to $\{P[3], P[6], P[7]\}$, and the *nextnode* field of $P[3]$ is 4. The corresponding node of $P[4]$ is $P[12]$ with headnode $P[3]$, but their contents are not equal and $P[12].back = 1 < index = 4$. Therefore, $P[12].back := 4$. When a mismatch occurs at $P[12]$, we can continue comparing $P[P[12].back]$ (i.e. $P[4]$) subsequently. TABLE III may be shown pictorially as in Fig. 10. The thin edges represent the *back* pointers. Some nodes in Fig. 10 don't have *back* pointers, because their *back* value is 0 and their *back* pointers is NULL.

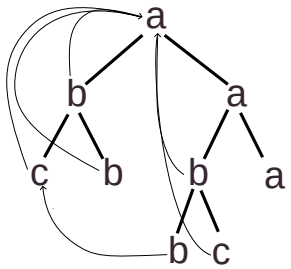


Fig. 10. The pattern after preprocessing.

The times that each node in the pattern needs to be visited is bounded by their level. The time complexity

TABLE IV
THE INDIRECTION STRUCTURE OF THE TREE IN FIG. 7

| **i**ndex | **n**umber |
|---|---|
| 1 | 1 |
| 2 | 2 |
| 3 | 3 |
| 4 | 4 |
| 5 | 5 |
| 6 | 6 |
| 7 | 7 |
| 8 | 12 |
| 9 | 13 |

of the preprocessing step is bounded by $O(m \times \log_2 m)$, where $m$ is the number of nodes in the pattern. The worst case may occur when there are many nodes in the pattern that have the same content as the root. For example, if the ancestors of node $P[k]$ in the pattern have the same content as the root, then node $P[k]$ may need to be visited at most $\lfloor \log_2 k \rfloor$ times.

The space complexity is dependent with the structure of the pattern. When the pattern is skew and has $m$ nodes, the space complexity is $O(2^m)$; when the pattern is a complete binary tree and has $m$ nodes, the space complexity is $O(m)$. Therefore, the space complexity is from $O(2^m)$ to $O(m)$.

However, we can use **i**ndirection to save the space. We use an extra array of structure $\{index, number\}$ for the tree. The *index* is the number of the node's sequence in the left-to-right breadth-first traversal, and the *number* is the index of the node in the numbering scheme we adopt. We can show it more clearly with the example in Fig. 7 in TABLE IV. Therefore the space needed for a binary tree will be down to $O(m)$ no matter what kind of structure the tree is.

Adopting the *indirection* into our algorithm, we need an extra searching algorithm. No matter what kind of the tree structure the pattern is, we know that $number \geq index$. We can use binary search, and the worst case in binary search takes $O(\log m)$ time complexity. Therefore, the time complexity of our algorithm in Fig. 6 with *indirection* will be $O(m \times \log^2 m)$ where $m$ is the number of nodes in the pattern.

### B.3 Searching a Pattern in a Subject Tree

For the subject tree, we need to record the *content*, *nextnode*, $flag$ and *match* of each node. The *nextnode* field is the next node's index in the left-to-right breadth-first traversal of the subject tree. The $flag$ field is either 0 or 1, and we initialize it as 1 in the beginning. When the node is visited as a headnode, we set its $flag$ field as 0 to avoid the repeated comparison. The *match* field is either 0 or 1, and we initialize it as 0 in the beginning. When we find a full match in the subject tree, we set the *match* field of the headnode in the full match as 1. Consequently,

the data structure of the subject tree is an array of the structure $\{content, nextnode, flag, match\}$.
The algorithm of searching the pattern in the subject tree is shown in Fig. 11.

```
Part IV
head := 0; // the corresponding headnode //
index := 1;
initialize the content and nextnode fields of nodes of the subject tree;
for ( i := 1; i ≠ 0; i := S[i].nextnode )
{  // initial the flag and match fields of subject nodes //
   S[i].match := 0;
   S[i].flag := 1;    // flag=0 means to stop searching //
}

Part V
for ( i := 1; i ≠ 0; i := S[i].nextnode )
{
   if S[i].flag ≠ 0 then
   // skip nodes that have been searched //
   {
      Com_Des(i,1,index,head);
      // compare descendants //
      if ( index ≠ 0 and P[index].nextnode = 0 )
      // Match //
      then S[head].match := 1;
      else index := 1;
      // back to the root of the pattern //
   }
}

Part VI
Com_Des(i,j,index,head)
{
   if S[i].content = P[j].content then
   {
      head:=Base(i,j);
      // compute the corresponding head node index //
      index := j;
      do
      {
         index := P[index].nextnode;
         // next node of the pattern//
         if index ≠ 0 then // check if the last node //
         {
            k := 2^⌊log2 index⌋ × (head − 1) + index;
            // find corresponding node //
            if S[k] =null then index := 0
            else Compare(S[k],P[index],index,head);
         }
      }
      until (index = 0 or P[index].nextnode = 0);
   }
}

Part VII
Compare(m,n,index,head)  // compare nodes //
{
   if m.content ≠ n.content then
   do
   {
      index := n.back;  // check back //
      if index ≠ 0 then head:=Base(k,index);
      n := P[index];
   }
   until (m.content = n.content or index = 0);
}

Part VIII
int Base(x,y)  // find the corresponding head node //
{
   while y ≠ 1 do
   {
      x := ⌊x/2⌋;  // find parent(x) //
      y := ⌊y/2⌋;  // find parent(y) //
   }
   k := x;
   S[k].flag := 0; // set the headnode's flag as 0 //
   return(k);
}
```

Fig. 11. Algorithm for searching the pattern in the subject tree.

TABLE V
THE INITIALIZATION RESULT OF THE SUBJECT TREE STRUCTURE IN FIG. 7

| index | content | nextnode | flag | match |
|---|---|---|---|---|
| **1** | a | 2 | 1 | 0 |
| **2** | b | 3 | 1 | 0 |
| **3** | a | 4 | 1 | 0 |
| **4** | c | 5 | 1 | 0 |
| **5** | b | 6 | 1 | 0 |
| **6** | b | 7 | 1 | 0 |
| **7** | a | 12 | 1 | 0 |
| **8** | | | | |
| **9** | | | | |
| **10** | | | | |
| **11** | | | | |
| **12** | b | 13 | 1 | 0 |
| **13** | c | 0 | 1 | 0 |

Part IV of Fig. 11 shows the initialization Part V is the main code for searching patterns in the subject tree. Part VI, Part VII and Part VIII are related procedures. In Part IV of Fig. 11, we first initialize the *content* and *nextnode* fields of the subject tree. The initialization is similar to Part I of Fig. 6. The *flag* field is initialized as 1. The *match* field is initialized as 0. We show the result of the initialization of the subject tree of Fig. 7 in TABLE V. Similarly as TABLE II, some nodes' fields are empty in TABLE V. Those nodes don't exist in the tree and waste space.

In Part V of Fig. 11, we traverse the nodes in the left-to-right breadth-first order. Only nodes whose *content*

fields equal to the *content* field of pattern's root may lead a match and has the need to compare their descendants. When a mismatch occurs, we know which node of the pattern should be compared to the node in the subject tree or just skip it. The index of a node in the pattern that should be compared subsequently comes from the *back* field of the pattern. When the value of the *back* field is 0, we skip the node in the subject tree because there is no chance to find the pattern with this node.

Procedure Com_Des of Part VI in Fig. 11 computes the index of a node's corresponding node and compares if their contents are equal. Procedure Compare used in procedure Com_Des handles the situation when a mismatch occurs at the node. We can immediately know the next node of the pattern should be compared subsequently. Procedure Base used in procedure Com_Des and Compare is for the computation of the headnode's index in the subject tree.

When we find a corresponding headnode in the subject tree, we continue to compare the descendants to determine if there is any chance to find the pattern in the subject tree. The steps are shown in procedure Com_Des. When a mismatch occurs, we handle it with procedure Compare and we must compute the index of the new corresponding headnode by procedure Base. Similar to the pattern preprocessing part, we must compute the index of the corresponding node. And the computation is similarly with LEMMA 3, we apply it to two different trees.

In Part V of Fig. 11, we visit each node in the subject tree as the headnode in the left-to-right breadth-first order. Therefore, we won't miss any chance to lead a match. The times of each node in the subject tree needs to be visited depends on the frequency of its ancestors' contents equal to the content of the pattern's root. Therefore, the times that each node in the subject tree needs to be visited is bounded by their level. The time complexity of the simple tree pattern-matching is bounded by $O(n \times \log_2 n)$, where $n$ is the number of nodes in the subject tree. The worst case may occur when there are many nodes in the subject tree that have the same content as the root of the pattern.

The space complexity is dependent with the structure of the subject tree. When the subject tree is skew and has $n$ nodes, the space complexity is $O(2^m)$; when the subject tree is a complete binary tree and has $n$ nodes, the space complexity is $O(n)$. Therefore, the space complexity is from $O(2^n)$ to $O(n)$.

However, we can similarly use **i**ndirection to save the space. We use an extra array of structure $\{index, number\}$ for the tree. The *index* is the number of the node's sequence in the left-to-right breadth-first traversal, and the *number* is the index of the node in the numbering scheme we adopt. Therefore the space needed for a binary tree will be down to $O(n)$.

Adopting the *indirection* into our algorithm, we need an extra searching algorithm. We can use binary search, and the worst case in binary search takes $O(\log n)$ time complexity. Therefore, the time complexity of our algorithm in Fig. 11 with *indirection* will be $O(n \times \log^2 n)$ where $n$ is the number of nodes in the subject tree.

## IV. Conclusions and Future Work

### A. Conclusions

In our simple tree pattern-matching algorithm, each node in the subject tree needs to be traversed is bounded by their level. Therefore, the time complexity of the simple tree pattern-matching is bounded by $O(n \times \log n)$, where $n$ is the number of nodes in the subject tree. The worst case may occur when there are many nodes in the subject tree that have the same contents as the root of the pattern.

And we need an extra step to preprocess the pattern. The times of each node needs to be visited when we preprocess the pattern depends on the frequency of its ancestors' contents equal to the content of the pattern's root. Therefore, the times that each node in the pattern need to be visited is bounded by their level. The time complexity of the preprocessing step is bounded by $O(m \times \log m)$, where $m$ is the number of nodes in the pattern. The worst case may occur when there are many nodes in the pattern that have the same content as the root.

The space complexity depends on the structures of the pattern and the subject tree. The worst case occurs when the subject tree and the pattern both are skewed, the space complexity is $O(2^n + 2^m)$, where $n$ is the number of nodes in the subject tree and $m$ is the number of nodes in the pattern. The best case is $O(n+m)$ when both pattern and the subject tree are complete binary trees. However, we can use *indirection* to save the space. Therefore the space complexity will be down to $O(n + m)$.

We can reduce the pattern to be stored in an array of structure $\{number, content, back\}$ in the left-to-right breadth-first order and the subject tree to be stored in an array of struture $\{number, content, flag, match\}$ in the left-to-right breadth-first order. The *number* field is the index of the node in our numbering representation. We don't need to memorize the nextnode of each node, because the nextnode of $P[i]$ is $P[i+1]$, and the nextnode of $S[k]$ is $S[k+1]$, $\forall\ 0 < i < m$, $\forall\ 0 < k < n$. But we do need an extra searching algorithm when we are visiting the nodes.

The simple tree pattern-matching algorithm shown in Fig. 6 and Fig. 11 can be applied only to binary trees. If we want to apply this algorithm to general trees, we must find out the largest degree of the trees and extend our numbering scheme accordingly. With this method, we only need to change the computation of the indices of the parent or children nodes in our algorithm.

TABLE VI summarizes the time complexity for the pattern preprocessing and matching techniques in [3] and our approach. The complexities in TABLE VI are expressed in terms of [3]

***patno*** the number of different patterns involved.
***patsize*** the size of the pattern forest (when patno=1, patsize is the number of the pattern nodes).
***subsize*** the size of the subject tree.
***ht*** the height of a specific tree which is constructed as part of preprocessing.
***sym*** the number of symbols in the alphabet $\Sigma$.
***rank*** the highest rank of any symbol in $\Sigma$.

TABLE VI

THE TIME COMPLEXITY FOR THE PREPROCESSING AND MATCHING TECHNIQUES.

| Method | Restrictions | Preprocessing time | Matching time |
|---|---|---|---|
| Naïve algorithm | None | None | $O(subsize \times patsize)$ |
| Bottom up with Algorithm A and B [3] | Simple pattern forest | $O(patsize^2 \times rank + ht \times sym \times patsize^{rank})$ | $O(subsize + match)$ |
| Bottom up with Algorithm C [3] | Simple binary forest | $O(patsize \times ht^2)$ | $O(subsize \times ht^2 + match)$ |
| Top down with Algorithm D [3] | Pattern are full trees | $O(patsize)$ | $O(subsize \times patno)$ |
| | None | $O(patsize)$ | $O(subsize \times suf)$ |
| Our approach | For binary trees and $patno = 1$ | $O(patsize \times ht)$ | $O(subsize \times \log subsize)$ |
| Our approach for general trees | $patno = 1$ | $O(patsize \times ht)$ | $O(subsize \times \log subsize)$ |
| Our approach with indirection | For binary trees and $patno = 1$ | $O(patsize \times ht^2)$ | $O(subsize \times \log^2 subsize)$ |

***suf*** the maximum suffix number of the path string of the tree pattern.

***match*** the number of matches which are found.

When we restrict the pattern and the subject tree be binary trees and $patno = 1$, our algorithm has shorter preprocessing time than bottom up algorithms listed in the TABLE VI. But we cann't evaluate which one has shorter matching time, excepting when $ht = patsize$, our algorithm may have shorter matching time than the method of Bottom up with Algorithm C. The method of top down with Algorithm D has better performance than our algorithm, no matter in preprocessing time or matching time. But when the $suf$ is large enough (i.e. $suf$ is equal to the *height* of the subject tree), our algorithm can have the same performance as the method of top down with Algorithm D.

### *B. Future Work*

The pattern preprocessing method we use in our approach is not efficient and complete. In the part of pattern preprocessing in the Knuth-Morris-Pratt string-matching algorithm, the $next[j]$ comes from $f[j]$ and $f[j]$ is the largest $i$ less than $j$ such that $pattern[1] \cdots pattern[i-1] = pattern[j-i+1] \cdots pattern[j-1]$. Each node in the string needs to be visited once. So as our algorithm, if each node in the subject tree needs to be visited at most twice, the complexity of matching time can be down to $O(n)$, where $n$ is the number of nodes in the subject tree (i.e. the size of the subject tree). But the tree structure is more complicated than string, we should keep on trying to find out an efficient and correct method to handle the pattern preprocessing. And we should find out the proper data structure for the trees.

However, our simple tree pattern-matching algorithm restricts the pattern number be 1, that means our approach cannot apply on pattern forest. When the pattern number is largest than 1, the time complexities of pattern preprocessing and matching techniques will be multiple even though the pattern forest has no independent subtrees and there are close relationships between the patterns. In [3], Hoffmann and O'Donnell proposed the subsumption graph and other concepts to handle the simple pattern forest. Our future work should contains this part of pattern forest handling.

## REFERENCES


[1] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* MIT Press, 1996.

[2] E. Horowitz, and S. Sahni. *Fundamentals of Data Structure in Pascal.* Computer Science Press, 1976.

[3] C. M. Hoffmann, and M. J. O'Donnell. *Pattern Matching in Trees.* JACM 29, Vol. 29, No. 1, pp. 68–95, January 1982.

[4] H. U. Simon. *Pattern Matching in Trees and Nets.* ACTA Informatica, pp.227–248, 1983.

[5] C. M. Hoffmann, and M. J. O'Donnel. *Fast Pattern Matching in Strings.* SIAM J. COMPUT., Vol. 6, No. 2, pp. 323–350, June 1997.

[6] E. M. Reingold, K. J. Urban and D. Gries. *K-M-P string matching revisited.* Information Processing Letters, Vol. 64, pp. 217–223, 1997.

[7] D.S. Hirschberg. *A Linear Space Algorithm for Computing Maximal Common Subsequences.* Communications of the ACM, Vol. 18, No. 6, pp. 341–343, June 1975.

[8] J. W. Hunt and T. G. Szymanski. *A Fast Algorithm for Computing Longest Common Subsequences.* Communications of the ACM, Vol. 20, No. 5, pp. 350–353, May 1997.

[9] D. S. Hirschberg. *Algorithms for the Longest Common Subsequence Problem.* JACM, Vol. 24, No. 4, pp. 664–675, October 1977.

[10] K.Z. Zhang. *The Editing Distance Between Trees: Algorithms and Applications.* Technical Report, New York University, July 1989.

[11] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, MA, 1974.