IMPLEMENTATION OF SUFFIX TREES USING TWO LINEAR TIME ALGORITHMS AND THEIR APPLICATION TO STRING MATCHING

By

TONGYU LI

Bachelor of Science

Hebei University

Baoding, China

1992

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
December 1999

IMPLEMENTATION OF SUFFIX TREES USING

TWO LINEAR TIME ALGORITHMS AND THEIR

APPLICATION TO STRING MATCHING

Thesis Approved:

J Chandler

Thesis Advisor

[illegible]

[illegible]

Wayne B. Powell

Dean of the Graduate College

PREFACE

A suffix tree is a trie-like data structure representing all suffixes of a string. Such trees have a central role in many algorithms on strings, such as finding the longest repeated substring, finding all squares in a string, approximate string matching, data compression, and DNA sequence assembly. It is quite commonly felt, however, that the linear-time suffix tree algorithms presented in the literature are rather difficult to grasp. I implemented the suffix trees using two linear-time algorithms and tested the application to string matching in the thesis.

ACKNOWLEDGMENT

I wish to express my sincere appreciation to my thesis advisor, Dr. John P. Chandler for his consistent and continuous advice, guidance, kindness, and valuable instruction through my graduate work. I also wish to express my gratitude to Dr. Mansur H. Samadzadeh for his guidance, encouragement, and friendship. My appreciation extends to Dr. H. K. Dai for his guidance and kindness.

I would like to give my special appreciation to my husband, Kit Lau, for his precious suggestions, encouragement, love, and understanding throughout this whole process.

I would like to take this opportunity to thank my parents, Dechong Li and Xianyun Liu, who have provided consistent support and endless love in my whole life. Also, I whould like to thank my sister, Lingyan Li, for her encouragement and care.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER I

INTRODUCTION

String matching is an integral part of many problems that occur naturally such as text editing, data retrieval, symbol manipulation, lexical analysis, and computational biology. Formally, string matching is defined as follows: given a short string of length m, called the pattern, and a long string of length n, called the text string, locate an occurrence of the pattern in the text string (either the first occurrence or all occurrences), reporting "not present" if the pattern does not occur in the text. Usually, the text strings to be searched are very large documents, such as books, dictionaries, encyclopedias, and databases of DNA sequences.

There are several linear-time string matching algorithms, such as the Knuth-Morris-Pratt algorithm [14], the Boyer-Moore algorithm [4], and the Karp-Rabin algorithm [11], that are designed to solve the specific string matching problem.

A suffix tree is a trie-like data structure representing all suffixes of a string. Such trees have a central role in many algorithms on strings [3] [7], such as finding the longest repeated substring [20], finding all squares in a string [2], approximate string matching [8] [5], data compression [16], and DNA sequence assembly [5]. It is commonly felt, however, that the linear-time suffix tree algorithms presented in the literature are rather difficult to grasp.

The suffix tree which can be constructed in linear time in the length of the text string, and yet which enables substring searches to be completed in time linear in the length of the substring, was first discovered by Weiner [20].

In constrast with the method by Weiner that proceeds right to left and adds the suffixes to the tree in increasing order of their length, starting from the shortest suffix, and with the method by McCreight [15] that adds the suffixes to the tree in decreasing order of their length from left to right, an on-line construction algorithm was discovered by Ukkonen [19], which proceeds the string symbol by symbol from left to right, and always has the suffix tree for the scanned part of the string ready (this is the definition of "on-line" in this context).

In this thesis, I implemented the suffix tree using two linear time algorithms – McCreight's algorithm and Ukkonen's algorithm, which process the string from left to right, tested its application for string matching, and compared the time and space being used by the two algorithms.

CHAPTER II

LITERATURE REVIEW

2.1 Basic Definitions

2.1.1 Suffix Trie and Suffix Tree

A trie is a type of digital search tree [13], and thus represents a set of pattern strings, or keys, over a finite alphabet [17]. For a set of strings over a finite alphabet C, each edge of the trie for the set represents a symbol from C, and sibling edges must represent distinct symbols. The maximum degree of any node in the trie is thus equal to |C|. The suffix trie, which is also sometimes referred to as a position tree or a non-compact suffix tree, is a trie whose set of keywords comprises the suffixes of a single string [17]. Furthermore, it requires that each suffix is represented by a distinct terminal node (leaf) of the trie. Figure 1 shows a suffix trie for the string ABCABC$.

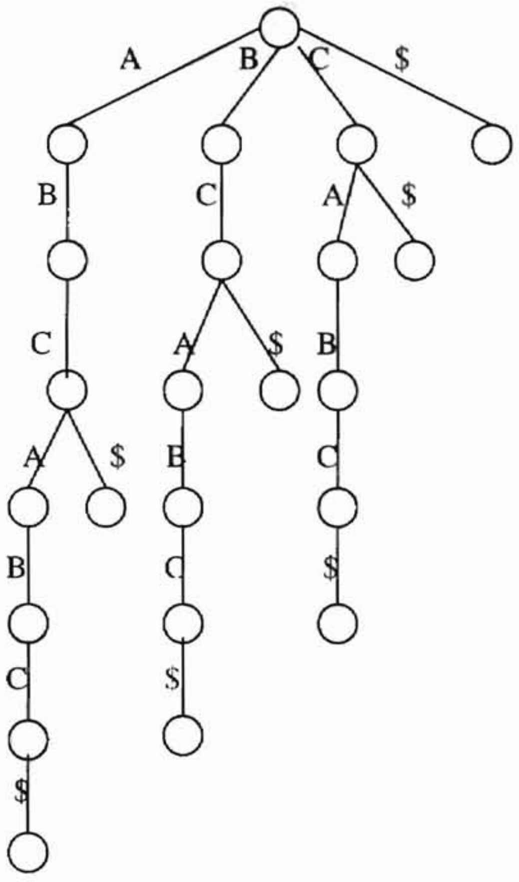

Figure 1: Suffix trie for string ABCABC$ [17]

The suffix tree, which is sometimes referred to more specifically as the compact suffix tree, reduces the number of edges by collapsing paths containing unary nodes, i.e. those nodes having only one child node. Figure 2 shows the result of converting the suffix trie of Figure 1 in this manner.

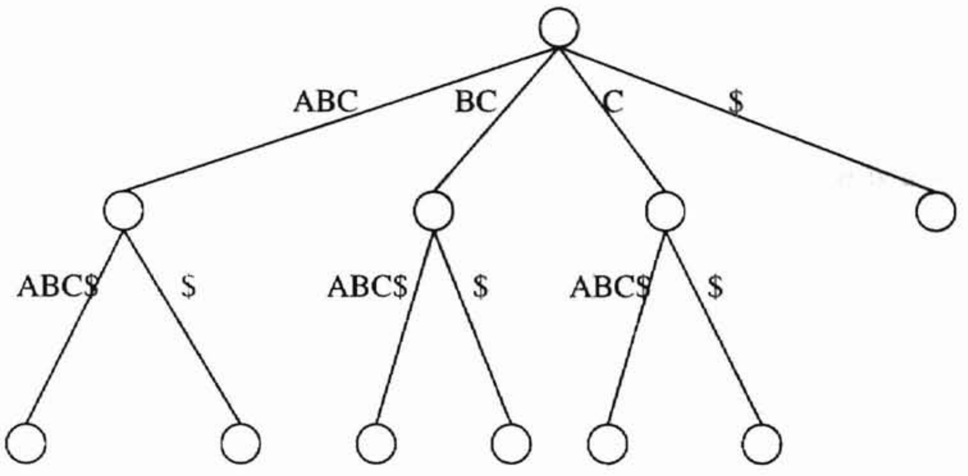

Figure 2: Suffix tree for string ABCABC$ [17]

Theorem 1 A suffix tree over a string S of length n uses Θ(n) space [13].

Proof: From the definition of the suffix tree, we know that each internal node of the suffix tree has at least two children. And a suffix tree over a string of length n has exactly n leaves. So, the total number of nodes in the suffix tree is at most 2n – 1, at least n, and so the suffix tree takes Θ(n) space.

2.1.2 Explicit States and Implicit States

A tree is a graph consisting of vertices, also called nodes, and states. All branching states (nodes), from which there are at least two transitions, and all leaves of suffix trie, are called *explicit states (explicit nodes)*. By definition, *root* is included in the branching states. In Ukkonen's algorithm [19], state $\perp$ is also an *explicit state*. The other states (the states other than *root* and $\perp$ from which there is exactly one transition) are called *implicit states (implicit nodes)*.

2.1.3 Suffix Link

For a node corresponding to a non-empty factor *aw* (*a* is a symbol and *w* is a string), we define the *suffix link* from *aw* to *w*, by *suf*[*aw*] = *w* (see Figure 3). It is expensive to walk down the root to find the path in each step of the extension of a suffix tree [6]. Both McCreight's algorithm and Ukkonen's algorithm take advantage of suffix links to save some unnecessary steps in this walk. We will discuss them in detail later.

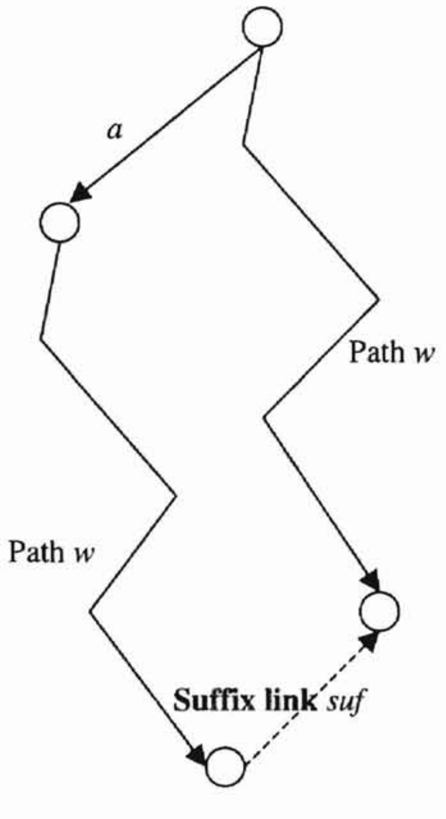

Figure 3: A suffix link *suf* [6]

2.2 The Two Algorithms That Construct a Suffix Tree in Linear Time

2.2.1 Differences in Suffix Tree Construction by the Two Algorithms

The algorithm of McCreight inserts the suffixes of string T into an initially empty tree. Starting with the longest suffix – the whole text string, the method is not on-line, and the intermediate trees are **not** suffix trees. The left column of Figure 4 shows the intermediate trees when constructing $T(c) = cst(adadc)$ using McCreight's algorithm.

The algorithm of Ukkonen has the important property of being on-line. That means processing the text string symbol by symbol from left to right, and always has the suffix tree for the scanned part of the text string ready. The algorithm is based on the simple observation that the suffixes of a string $T^i = t_1 \ldots t_i$ can be obtained from the suffixes of string $T^{i-1} = t_1 \ldots t_{i-1}$ by concatenating symbol t_i at the end of each suffix of T^{i-1} and by adding the empty suffix. The suffixes of the whole string $T = T^n = t_1\, t_2 \ldots t_n$ can be obtained by first expanding the suffixes of T^0 into the suffixes of T^1 and so on, until the suffixes of T are obtained from the suffixes of T^{n-1} [19]. The intermediate trees when constructing $cst(adadc)$ using Ukkonen's algorithm are shown in the right column of Figure 4.

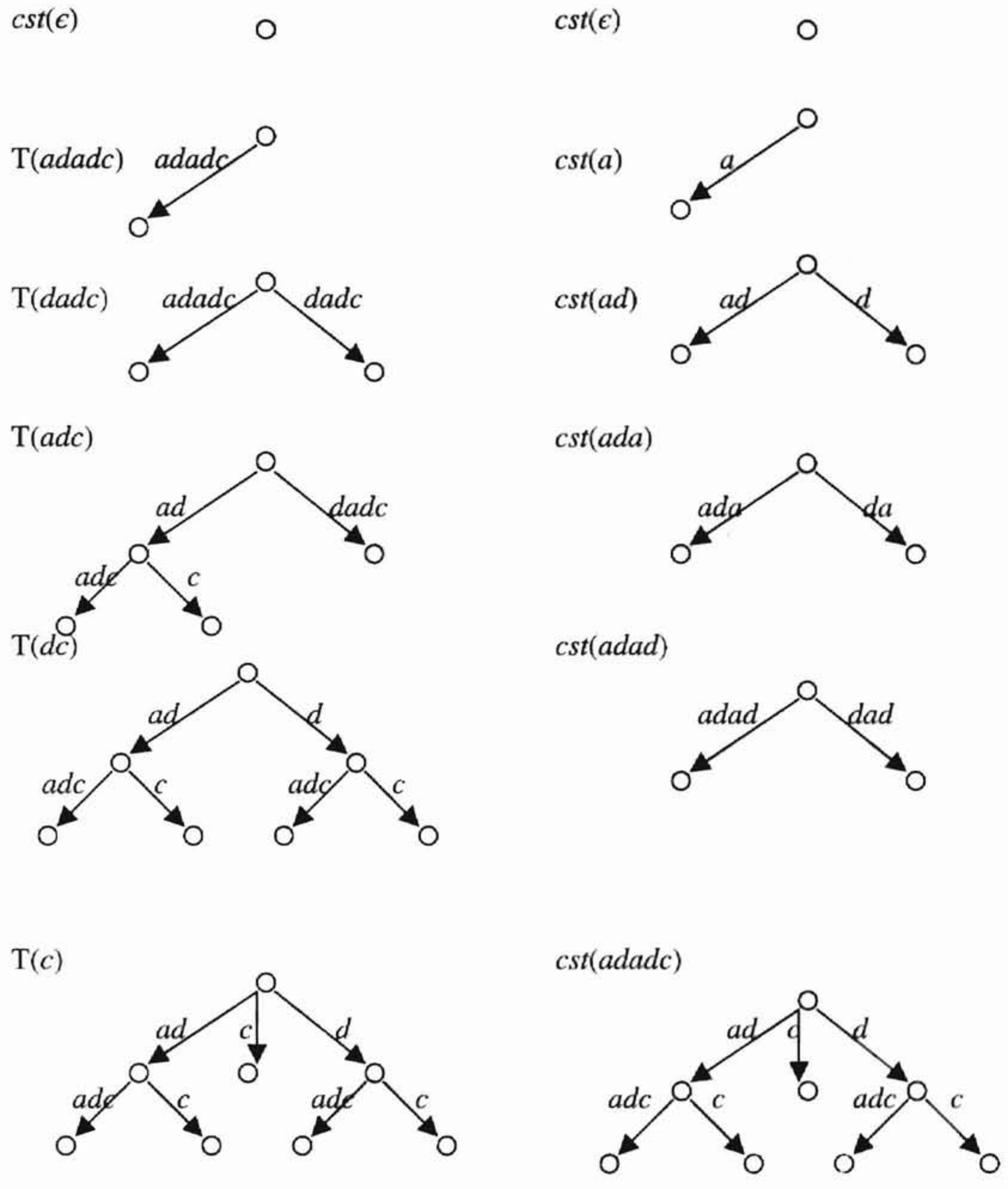

Figure 4: Sequence of trees constructed by McCreight and Ukkonen [10]

2.2.2 McCreight's Algorithm

General Scheme for Trie Construction

McCreight's algorithm is an incremental algorithm for constructing a suffix tree. The suffixes of the text string are inserted into the initial empty tree one-by-one, starting from the longest suffix. The intermediate trees contain the subsets of the suffixes. The insertion continues until all suffixes are included in the tree. For example, suppose p is the next suffix to be inserted in the current tree T. McCreight's algorithm defines the head of p, $head(p, T)$, as the longest prefix of p occurring in T (as the label of a path from the root). We can find $head(p, T)$ with its corresponding node in the tree. After finding the node, only the remaining part of p, say π, needs to be appended to $head(p, T)$. Now a new intermediate tree is constructed, and a new branch labeled by p is included (see Figure 5). Below is the general scheme of McCreight's algorithm [6].

```
Algorithm general-scheme;
begin
        compute initial tree $T$ for the first suffix;
        $leaf$ := leaf of T;
        for $i$ := 2 to $n$ do begin
                {insert next suffix}
                localize next $head$ as $head$(current suffix, $T$);
                let $\pi$ be the string corresponding to path from $head$ to $leaf$;
                create new path starting at $head$ corresponding to $\pi$;
                $leaf$ := lastly created leaf;
        end;
end.
```

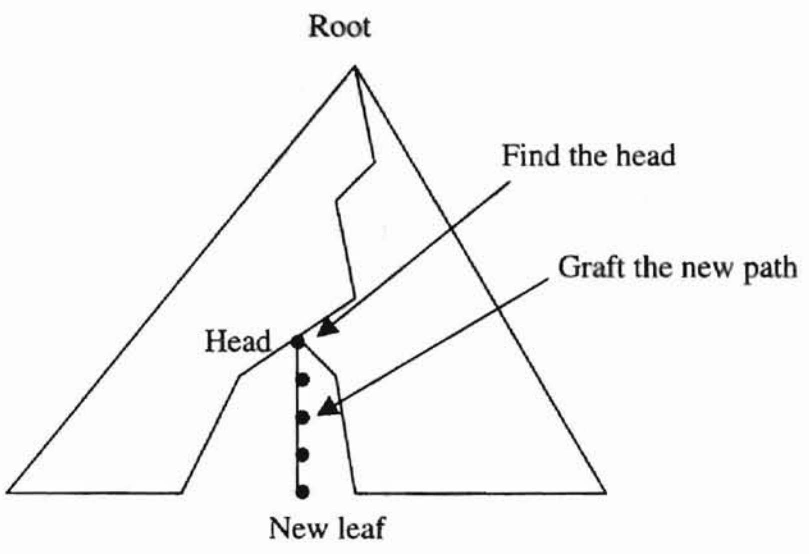

Figure 5: Insertion of the next suffix [6]

Up_Link_Down is the main technique used by McCreight's algorithm. It is the most significant improvement on a straightforward construction by using the suffix links. The idea is to find the node (the head) of the next suffix to be inserted from the most recently created leaf of the tree. The data structure that takes advantage of suffix links gives shortcuts in searching for the heads. The procedure Up_Link_Down works as follows: it starts from the last leaf, going up, until a shortcut through a suffix link is met; it then goes through the suffix link and goes down the tree to find the new head (see Figure 6). The following is the procedure Up_Link_Down [6].

```
**function** Up_Link_Down(*link*, *q*) node;
{finds new head, from leaf *q*}
**begin**
{UP, going up from leaf *q*}
        $v$ := first node $v$ on path from $q$ to root s.t. $link[v] \neq$ nil;
        **if** no such node **then return** nil;
        let $\pi = a_j\, a_{j+1\ldots}\, a_n$ be the string, label of path from $v$ to $q$;
{LINK, going through suffix link}
        $head := link[v]$;
{DOWN, going down to new head, making new links}
        **while** $son(head, a_j)$ exists **do begin**
                $v := son(v, a_j)$; $head := son(head, a_j)$;
                $link[v] := head$; $j := j+1$;
        **end;**
        **return** $(v, head)$;
**end.**
```

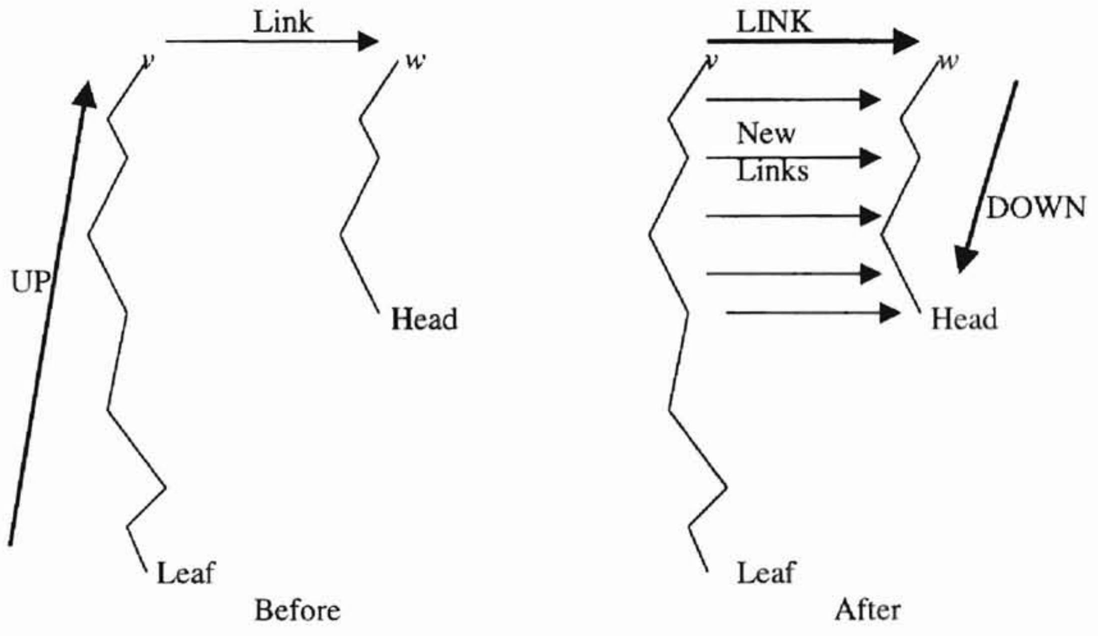

Figure 6: Strategy for finding the next head [6]

The procedure Graft [6] constructs a path of new nodes from the current head to a newly created leaf. It also updates the suffix links from the nodes on the path containing the previous leaf pointing to the corresponding nodes on the newly created path.

```
procedure Graft($link$, $v$, $head$, $a_j a_{j+1 \dots} a_n$);
begin  $w := head$;
        for $k := j$ to $n$ do begin
                $v := son(v, a_k)$; $w :=$ createson($w$, $a_k$); $link[v] := w$;
        end;
{$w$ is the last leaf}
end.
```

The algorithm Left_to_Right [6] builds the suffix tree of text string. Table *suf* will keep track of all the suffix links created during the construction of $Trie(p_1, p_2 \dots, p_n)$. In $Trie(p_1, p_2 \dots, p_n)$ it is possible for $suf[v]$ to be undefined, for some node v. This situation does not occur for (compressed) suffix trees.

```
        Algorithm Left_to_Right($a_1 a_2 \dots a_n$, n>0);
        begin
                $T := Trie(p_1)$ with suffix link (from son of root to root);
                for $i := 2$ to $n$ do begin
                {insert next suffix $p_i = a_1 a_2 \dots a_n$ into $T$}
                        {FIND new head}
                        ($v$, $head$) := Up_Link_Down($suf$, $leaf_{i-1}$); {$head = head_i$}
                        if $head$ = nil then begin
                        {root situation}
                                let $v$ be the son of root on the branch to $leaf_{i-1}$;
                                $suf[v]$ := root; $head$ := root;
                        end;
                        {going down from $head_i$ to $leaf_i$ creating a new path}
                        let $a_j \dots a_n$ be the label of the path from $v$ to $leaf_{i-1}$;
                        Graft($suf$, $v$, $head$, $a_j a_{j+1 \dots} a_n$);
        end;
end.
```

Theorem 2 The algorithm Left_to_Right constructs the T=*Trie*(*text*) in time O(|T|) [6].

Proof: The time is proportional to the number of suffix links (*suf*), which is obviously proportional to the number of internal nodes, and then to the total size of the tree.

McCreight's Algorithm

After the path π corresponding to a new suffix p is inserted into the tree, a new leaf is created. The father of the leaf, the nearest non-root explicit node, is denoted to be the current head. The head may not exist when the path contains only two explicit nodes – root and a leaf. In this case, we have to split the edge at the first implicit node from the root.

Following the function Up_Link_Down, the searching for heads may stop at some implicit nodes. We need break the edge at that point. Let (w, α) be an implicit node of the tree T (w is an explicit node of T, α is a string corresponding to the implicit node). The operation *break*(w, α) on the tree T is called only if our searching for the new head stopped at the implicit node (w, α). The effect of the operation *break*(w, α) is to break the corresponding edge, a new explicit node is inserted at the breaking point, and the edge is split into two edges: one is the existing edge, another will be grafted to a new path. The value of *break*(w, α) is the node created at the breaking point.

Let v be an explicit node of the tree T, and let p be a substring of the input string *text* represented by a pair of indexes l, r, then p = *text*[l … r]. The basic function used in

McCreight's algorithm is the function *find*. The value returned by *find*(v, p) is the last implicit node along the path starting in node v and labeled by p. The important aspect of McCreight's algorithm is the use of two different implementations of the function *find*. Function *fastfind* deals with the situation when we know in advance that the searching path labeled by p is fully contained in some path starting at v. The paths between the two suffix links should have an exactly same sequence. Node v is connected by a suffix link, and p actually is a path from the node that connects to v. We can save some steps by using *fastfind* to jump from one explicit node to another. Checking the first symbol of each edge to determine where to go is the only thing we need to do. Another implementation of find is the function *slowfind* [6] that follows the path symbol by symbol. The application of *fastfind* [6] is a main feature of McCreight's algorithm, and plays a central role in the performance.

```
function fastfind(v: node; p: string) node;
{p is fully contained in some path starting at v}
begin
    from node v, follow path labeled by p in the tree using labels of edges as
    shortcuts; only first symbols on each edge are checked;
    let (w, α) be the last implicit node;
    if α is empty then return w
    else return break (w, α);
end.

function slowfind(v: node; p: string) node;
begin
    from node v, follow the path labeled by the longest possible prefix of p
    letter by letter;
    let (w, α) be the last implicit node;
    if α is empty then return w
    else return break (w, α);
end.
```

McCreight's algorithm builds a sequence of trees T_i in the order $i = 1, 2, \ldots, n$. The tree T_i contains all the suffixes with length equal to or greater than n-i+1. Then T_n is the suffix tree for the whole text, but the intermediate trees are not strictly suffix trees. When we build tree T_k from T_{k-1}, suffix links play a crucial role in reducing the complexity.

McCreight's algorithm [6] is a transformation of the algorithm Left_to_Right; most of the nodes become implicit nodes here.

```
Algorithm scheme of McCreight's algorithm;
{left-to-right suffix tree construction}
begin
        compute the two-node tree $T$ with one edge labeled $p_1 = text$;
        for i := 2 to n do begin
                {insert next suffix $p_i = text[i..n]$}
                localize $head_i$ as $head(p_i, T)$,
                starting the search from $suf[\text{father}(head_{i-1})]$,
                using fastfind whenever possible;
                $T := insert(p_i, T)$;
         end;
end.
```

Property 1 *head*$_i$ is a descendant of the node $suf[head_{i-1}]$, $suf[v]$ is a descendant of $suf[\text{father}(v)]$ for any v [6].

Localizing heads is the first step in construction of the intermediate trees for McCreight's algorithm. The relation between *head*$_i$ and *head*$_{i-1}$ (Property 1) permits the search for the next head to start from some internal node, instead of from the root. This saves some work and the amortized complexity is linear. The behavior of McCreight's algorithm [6] is illustrated in Figure 7 and Figure 8.

```
Algorithm McCreight;
begin
        $T$ := two-node tree with one edge labeled by $p_1 = text$;
        for i := 2 to n do begin
                {insert next suffix $p_i = text[i..n]$}
                let $\beta$ be the label of the edge (father[$head_{i-1}$], $head_i$);
                let $\gamma$ be the label of the edge ($head_{i-1}$, $leaf_{i-1}$);
                $u$ := $suf$[father[$head_{i-1}$]];
                $v$ := $fastfind(u, \beta)$;
                if $v$ has only one son then
                        {$v$ is a newly inserted node} $head_i := v$
                else $head_i := slowfind(v, \gamma)$;
                $suf[head_{i-1}] := v$;
                create a new leaf $leaf_i$; make $leaf_i$ a son of $head_i$;
                label the edge ($head_i$, $leaf_i$) accordingly;
        end;
end.
```

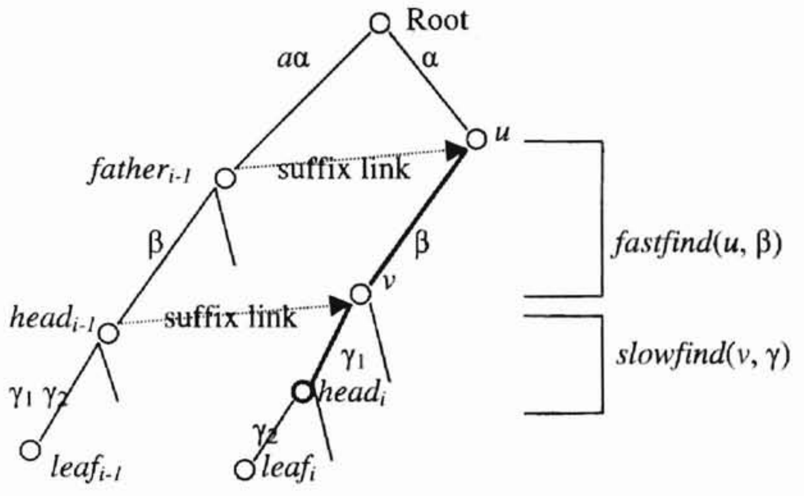

Figure 7: McCreight's algorithm: the case when v is an existing node [6]

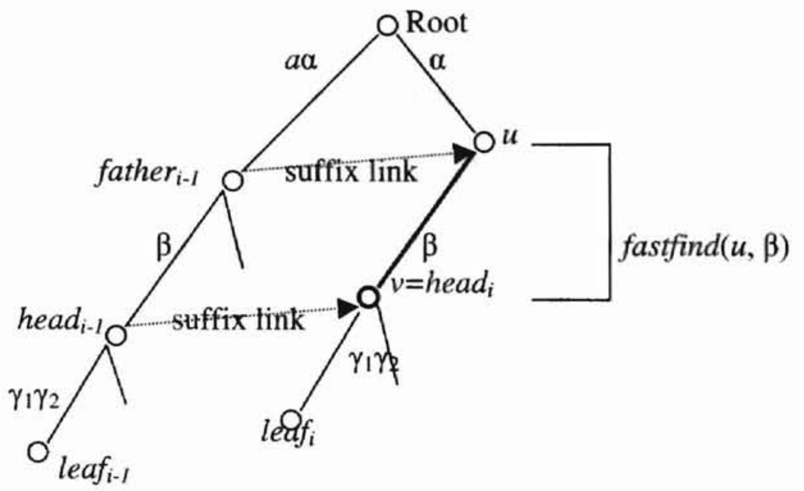

Figure 8: McCreight's algorithm: the case when $v = head_i$ is a newly created node [6]

Theorem 3 McCreight's algorithm has O(nhlog|A|) time complexity, where *A* is the underlying alphabet of the text of length *n* [6].

Proof: Let's assume that the alphabet is of a constant size. The time complexity of *fastfind* and *slowfind* can be considered separately. First, from the function *fastfind* above, it is not difficult to find that the time spent by *fastfind* at stage *i* should be proportional to the difference $|head_i| - |head_{i-1}|$, plus some constant. The total time should be bounded by $\Sigma(|head_i| - |head_{i-1}|) + O(n)$. It is linear time. Similarly, the time spent by *slowfind* at stage *i* is proportional to the difference $|father_i| - |father_{i-1}|$, plus some constant. Therefore, the total time of *slowfind* is bounded by $\Sigma(|father_i| - |father_{i-1}|) + O(n)$. It is also in linear time. For the situation that the alphabet size is not constant, we can find the stage from a single symbol down the tree through binary search in time O(log|A|). So, the total time complexity for McCreight's algorithm is O(nhlog|A|).

2.2.3 Ukkonen's Algorithm

We mentioned early that Ukkonen's algorithm for constructing suffix tree was on-line. It processes the text string symbol by symbol from left to right, and always has the suffix tree for the scanned part of the string ready. Now, we discuss the algorithm in detail.

Construction of Suffix Trie (*STrie*(*T*))

The transition function is defined as $g(\bar{x}, a) = \bar{y}$ for all $\bar{x}, \bar{y}$ in Q such that $y = xa$, where $a \in \Sigma$. Q is the set of the states in the suffix trie (*STrie*(*T*)). $\bar{x}$ denotes the state that corresponds to a substring x [19].

The suffix function f is defined for each state $\bar{x} \in Q$ as follows. Let $\bar{x} \neq root$. Then $x = ay$ for some $a \in \Sigma$, and we set $f(\bar{x}) = \bar{y}$ [19]. And, $f(root) = \perp$. We call $f(r)$ the suffix link of state r in Ukkonen's algorithm.

By Ukkonen's algorithm, the intermediate suffix trie T^i can be obtained from T^{i-1} by concatenating symbol t_i at the end of each suffix of T^{i-1} and by adding the empty suffix. Algorithm 1 shows the procedure for building $STrie(T^i)$ from $STrie(T^{i-1})$.

Here *top* denotes the state $\overline{t_1 \ldots t_{i-1}}$.

Algorithm 1

```
begin
    $r := top$;
    while $g(r, t_i)$ is undefined do
        create new state $r'$ and new transition $g(r, t_i) = r'$;
        if $r \neq top$ then create new suffix link $f(oldr') = r'$;
        $oldr' := r'$;
        $r := f(r)$;
    create new suffix link $f(oldr') = g(r, t_i)$;
    $top := g(top, t_i)$;
end.
```

On-Line Construction of Suffix Tree

The suffix tree of T is denoted as $STree(T) = (Q' \sqcup \{\perp\}, root, g', f')$ [19]. Ukkonen's algorithm refers to an explicit or implicit state r of a suffix tree by a *reference pair* (s, w), where s is some explicit state that is an ancestor of r, and w is the string spelled out by the transitions from s to r in the corresponding suffix trie. A *reference pair* is *canonical* if s is the closest ancestor of r (and, hence, w is the shortest possible string) [19]. For an explicit r the *canonical reference pair* obviously is (r, ϵ) [19].

Let's see how Algorithm 1 works. Let $s_1 = \overline{t_1 \ldots t_{i-1}}, s_2, s_3, \ldots, s_i = root, s_{i+1} = \perp$ be the states of $STrie(T^{i-1})$. j is the smallest index such that s_j is not a leaf, j' is the smallest index such that $s_{j'}$ has a t_i–transition. We know that s_1 is a leaf and root $\perp$ is a nonleaf that has a t_i–transition. $j \leq j'$ is always true.

Theorem 4 Algorithm 1 adds to $STrie(T^{i-1})$ a t_i–transition for each of the states s_h, $1 \leq h < j'$, the new transition expands an old branch of the trie that ends at leaf s_h, and, for $j \leq h < j'$, the new transition initiates a new branch from s_h. Algorithm 1 does not create any other transitions [19].

Ukkonen's algorithm calls state s_j the *active point* and $s_{j'}$ the *endpoint* of $STrie(T^{i-1})$. The states may be explicit or implicit. The active points of the last three trees in Figure 9 below are (*root*, *c*), (*root*, *ca*), (*root*, ϵ).

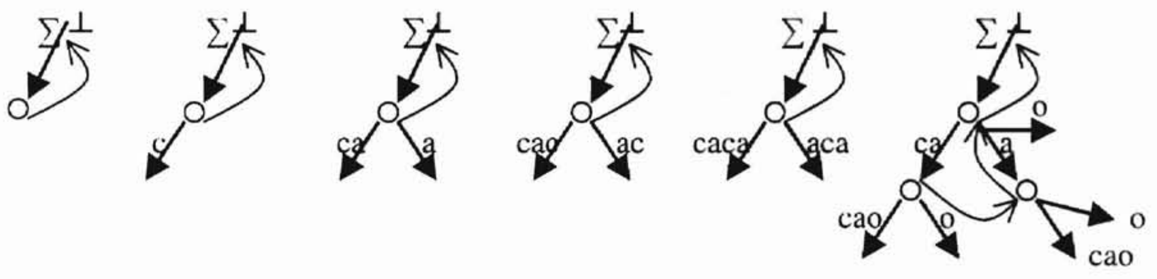

Figure 9: Construction of *STree*(cacao) [19]

From Theorem 4 we know that Algorithm 1 inserts two different groups of t_i –transitions into $STrie(T^{i-1})$: Group one are the states before the active point s_j get a transition. Theses states are leaves, so each such transition has to expand an existing branch of the trie. Another group are the states between the active point s_j and the endpoint $s_{j'}$, the endpoint is excluded, get a new transition. These states are not leaves, so each new transition has to initiate a new branch.

Any transition of $STrie(T^{i-1})$ leading to a leaf is called an *open transition* [19]. The form of the transition is $g'(s, (k, i\text{-}1)) = r$, where the right pointer points to the last position $i\text{-}1$ of tree T^{i-1}. So the actual value of the right point does not have to be present in the formula. We can rewrite an open transition as $g'(s, (k, \infty)) = r$ where ∞ means the transition is "open to grow." The explicit updating of the right pointer when t_i is inserted into this branch is not needed. So the first group of transitions is implemented without any explicit changes to $STree(T^{i-1})$.

Now let's focus on how to add the second group of transitions to $STree(T^{i-1})$. Let $h = j$ (s_j is an active point) and let (s, w) be the canonical reference pair for s_h. Since s_h is in $STrie(T^{i-1})$, w is a suffix of T^{i-1}. So $(s, w) = (s, (k, i-1))$ for some $k \leq i$. Some new branches from states s_h, $j \leq h < j'$ should be created. We need to take advantage of reference pairs and suffix links to save the steps searching states s_h since s_h may not be explicit states.

Next we need to create a new branch starting from the state represented by $(s, (k, i-1))$. At first, we should test whether or not $(s, (k, i-1))$ already refers to the endpoint $s_{j'}$. If it does, we do nothing. Otherwise a new branch will be created. The state s_h referred to $(s, (k, i-1))$ should be explicit. If it is not, we need to split the edge to generate a new explicit state s_h. Now a t_i–transition from s_h is created. It is an open transition $g'(s_h, (i, \infty)) = s_{h'}$ where $s_{h'}$ is a new leaf. At the end, the suffix link $f'(s_h)$ should be added if s_h was not an explicit state before.

The reference pair for s_h was $(s, (k, i-1))$, the canonical reference pair for s_{h+1} is *canonize*$(f'(s), (k, i-1))$ where *canonize* makes the reference pair canonical by updating the state and the left pointer. The above operations are then repeated for s_{h+1}, and so on until the endpoint $s_{j'}$ is found.

The procedure *update* [19], given below, together with procedure *canonize* [19] and *test-and-split* [19] transforms $STree(T^{i-1})$ into $STree(T^{i})$ by inserting the t_i–transitions in the second group. The procedure *test-and–split* [19] tests whether or not a given reference pair refers to the endpoint. If it does not, then the procedure creates and returns an explicit state for the reference pair, provided that the pair does not already represent an explicit state.

```
procedure update(s, (k, i)):
    (s, (k, i – 1)) is the canonical reference pair for the active point;
    oldr := root; (end-point, r) := test-and-split(s, (k, i – 1), t_i);
    while not(end-point) do
        create new transition g'(s, (i, ∞)) = r' where g'(s, (k, ∞)) = r' is a new state;
        if oldr ≠ root then create new suffix link f'(oldr) = r;
        oldr := r;
        (s, k) := canonize(f'(s), (k, i – 1));
      (end-point, r) := test-and-split(s, (k, i –1 ), t_i);
    if oldr ≠ root then create new suffix link f'(oldr) = s;
    return (s, k).

procedure test-and-split(s, (k, p), t):
    if k ≤ p then
      let g'(s, (k', p')) = s' be the t_k–transition from s;
      if t = t_{k' + p – k + 1} then return(true, s)
      else
            replace the t_k–transition above by transitions g'(s, (k', k' + p – k)) = r and
            g'(r, (k' + p – k + 1, p')) = s' where r is a new state;
            return(false, r)
    else
      if there is no t-transition from s then return(false, s)
      else return(true, s).
```

```
procedure canonize(s, (k, p)):
    if $p < k$ then return (s, k)
    else
        find the $t_k$–transition $g'(s, (k', p')) = s'$ from s;
        while $p' - k' \le p - k$ do
            $k := k + p' - k' + 1$;
            $s := s'$;
            if $k \le p$ then find the $t_k$–transition $g'(s, (k', p')) = s'$ from s;
        return (s, k).
```

The overall algorithm [19] for constructing *STree*(*T*) is finally as follows.

```
Algorithm 2. Construction of STree(T) for string $T = t_1t_2 \ldots \#$ in alphabet $\Sigma = \{t_{-1}, \ldots, t_{-m}\}$; # is the end marker not appearing else where in T.
create states root and $\perp$;
for $j := 1, \ldots, m$ do create transition $g'(\perp, (-j, -j)) = root$;
create suffix link $f'(root) = \perp$;
$s := root$; $k := 1$; $i := 0$;
while $t_{i+1} \ne \#$ do
    $i := i + 1$;
    $(s, k) := update(s, (k, i))$;
    $(s, k) := canonize(s, (k, i))$.
```

CHAPTER III

COMPARISION OF THE TWO ALGORITHMS

Suffix trees provide efficient solutions to a "myriad" of string processing problems. The suffix tree for a string t really turns t inside out, immediately exposing properties like longest or most frequent subwords. The fundamental question whether w occurs in t can be answered in O(|w|) steps – independent of the length of t – once the suffix tree for t is constructed. Thus it is of great importance that the suffix tree for t can be constructed and represented in linear time and space.

In spite of the basic role of suffix trees for string processing, elementary books on algorithms and data structures barely mention suffix trees and never give efficient algorithms for their construction.

There are two classical suffix tree construction algorithms: Weiner's algorithm and McCreight's algorithm. Weiner's algorithm [20] was the first linear-time algorithm. A few years later, a more space-efficient algorithm was developed by McCreight [15]. The two algorithms follow the same scheme for construction: the tree is computed for a subset of the suffixes and this procedure continues until all suffixes are included in the tree. Weiner's algorithm scans the text from right to left, while McCreight's algorithm scans the text from left to right. Though both algorithms use linear time, McCreight's algorithm was the first algorithm truly using linear space.

Ukkonen's algorithm [19] is a conceptually similar linear-time algorithm for constructing a suffix tree. Like McCreight's algorithm, Ukkonen's algorithm scans the text from left to right and has the same space improvement as Weiner's algorithm. However, "on-line construction" is the distinct difference from McCreight's algorithm.

I implemented the two efficient algorithms: McCreight's algorithm and Ukkonen's algorithm in C++, and, using real data, a book, as the input English text data and also using randomly generated DNA data.

3.1 Time Complexity

Using the English text data and DNA data, I ran the programs for text string sizes n: 200, 500, 1000, 2000, 4000, 6000, 8000, 10000, 12000, 14000, 16000, 18000, 20000, 22000, 24000. I got the average system time of constructing of suffix trees for the two algorithms with English text data and DNA data for each size n. Each average value was calculated from 20 outputs with the same size of random input data.

3.1.1 Time Complexity

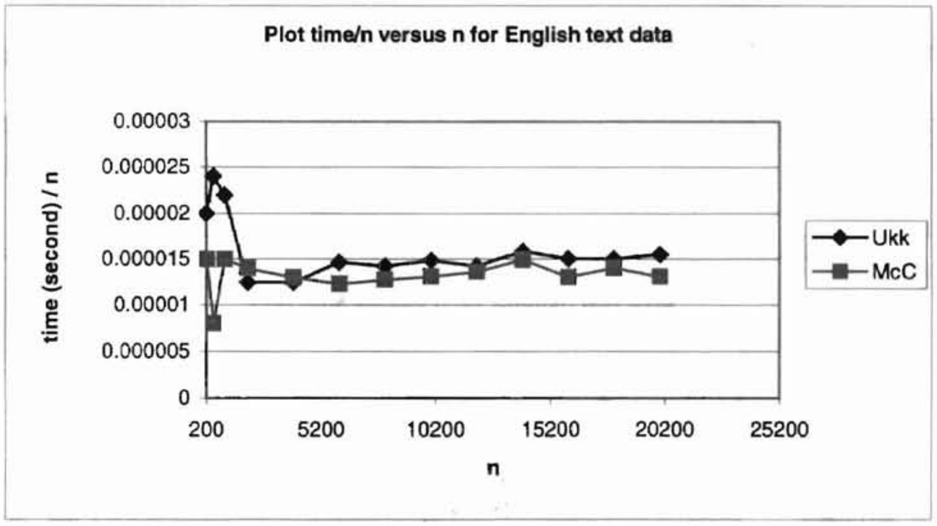

Figure 10: Plot time/n versus n for English text data

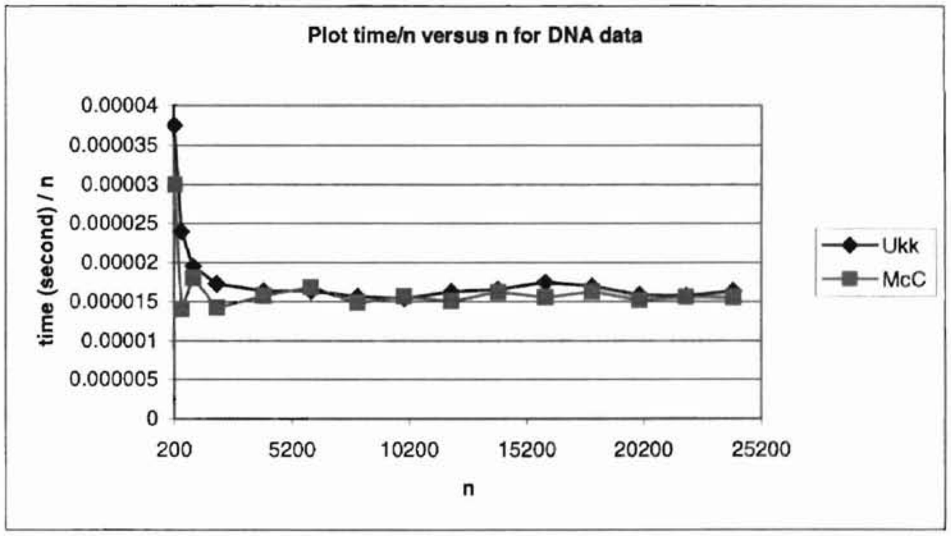

Figure 11: Plot time/n versus n for DNA data

From Figure 10 and Figure 11 that plot time/n versus n for English text data and DNA data, we can see that with the increment of n, the graphs approach horizontal straight lines (y=0.000015, a very small number). This verifies empirically that the time complexity of the two algorithms is O(n).

3.1.2 Comparison of the Two Algorithms in the Time Being Used for Constructing of Suffix Tree

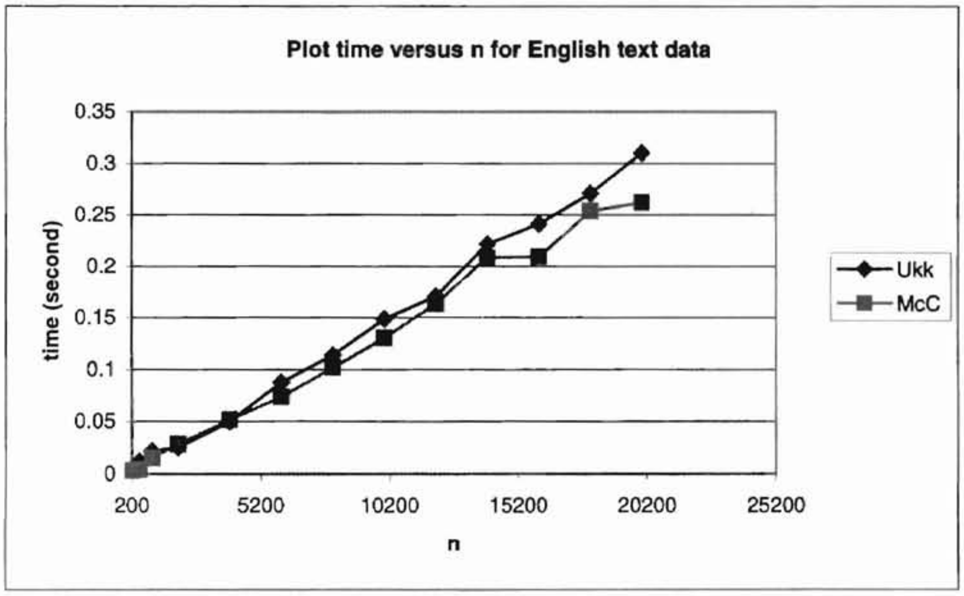

Figure 12: Plot time versus n for English text data

We know that both of the two algorithms are linear time algorithms. But which one is more efficient?

From Figure 12 and 13, we can see that the time spent by both algorithms increases with the increment of the size of text string. And, for both English text data and DNA data, the time spent by Ukkonen's algorithm in constructing a suffix tree is slightly more than McCreight's algorithm, though the difference is not big.

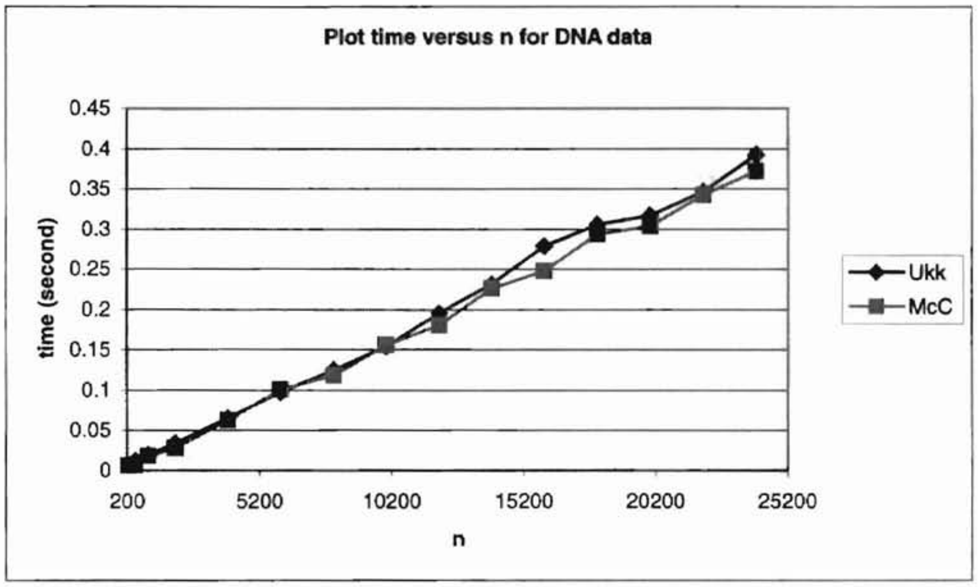

Figure 13: Plot time versus n for DNA data

We can conclude that McCreight's algorithm is slightly more efficient than Ukkonen's algorithm in the time being used for constructing suffix trees.

3.1.3 Deviation of the Two Algorithms

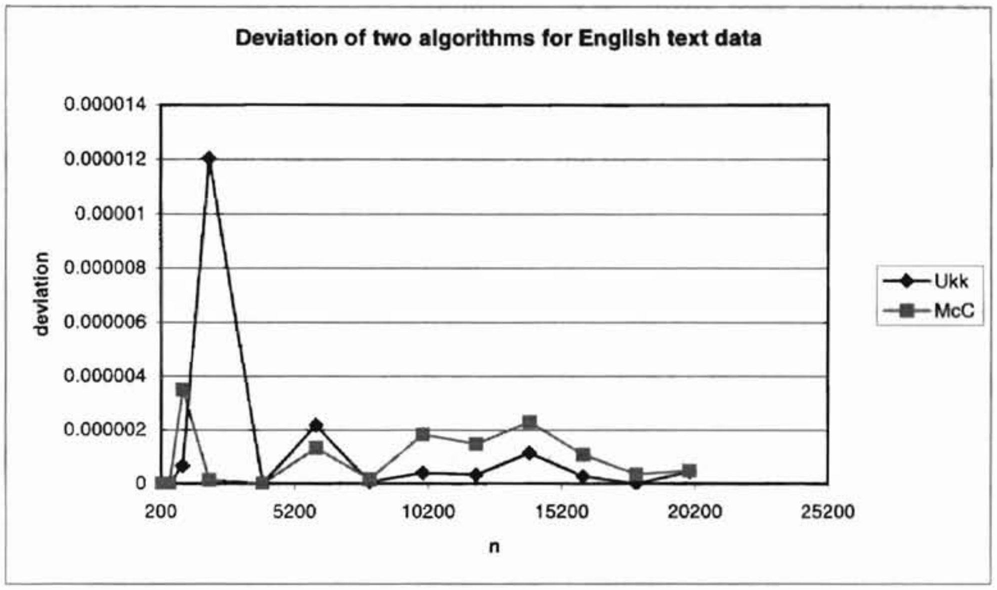

Figure 14: Deviation of two algorithms for English text data

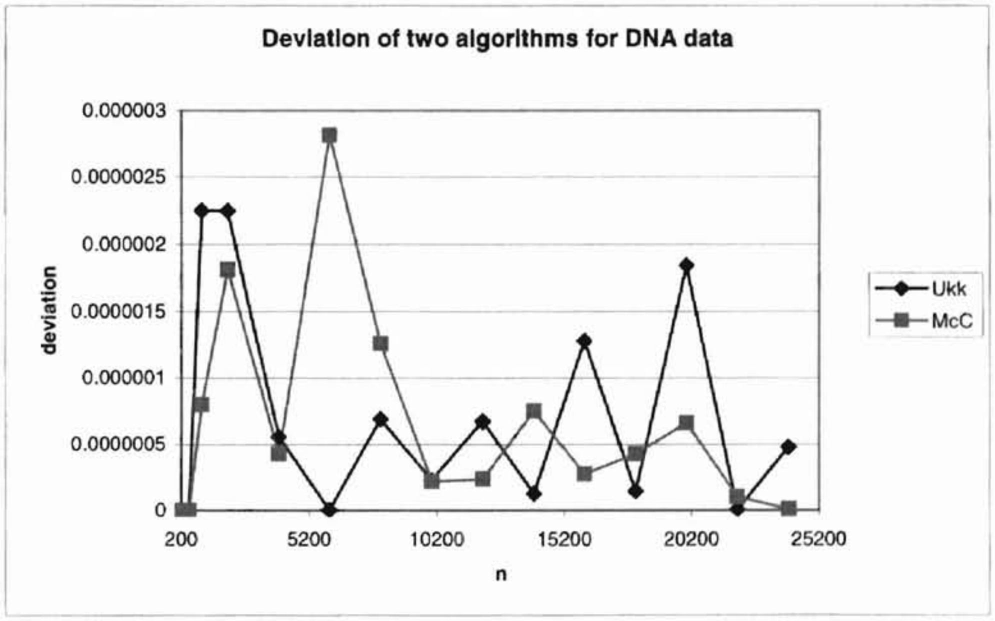

Figure 15: Deviation of two algorithms for DNA data

I also plot the deviation of time/n versus n for both algorithms using two different sets of data. From Figure 14 and Figure 15, we can see that the difference of deviations for the two algorithms is not significant.

3.2 Space Complexity

Now, let's discuss the space complexity of the suffix trees constructed by the two algorithms. Since the suffix tree is composed of edges (nodes), from the number of edges of a suffix tree, we know the space occupied by the tree.

We know that the definition of the suffix tree for both of the algorithms is unique. The space occupied by the suffix trees that constructed by the two algorithms should be exactly the same. My programs also verify this point.

Using the English text data and DNA data, I ran the programs for text string sizes n: 200, 500, 1000, 2000, 4000, 6000, 8000, 10000, 12000, 14000, 16000, 18000, 20000, 22000, 24000. I got the average edge number of the suffix tree with random DNA data for each size n. Each average value was calculated from 20 outputs with the same size of random input data. Since the edge number is the same for both algorithms for one input data, I just used one set of data for each of the graphs, not as in the above part for time complexity.

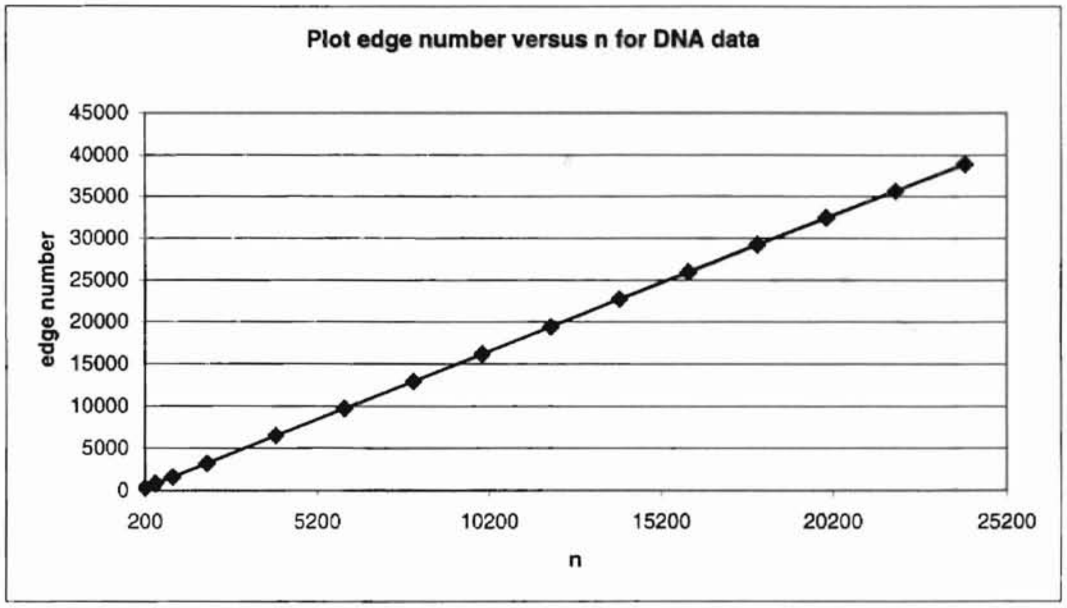

Figure 16: Plot edge number versus n for DNA data

We can see from Figure 16 that with the increment of n the edge number is increased. The graph is almost a perfect straight line. From Figure 17 that plot edge number/n versus n, we can see the value of edge number/n is within [1.617, 1.624], a small range.

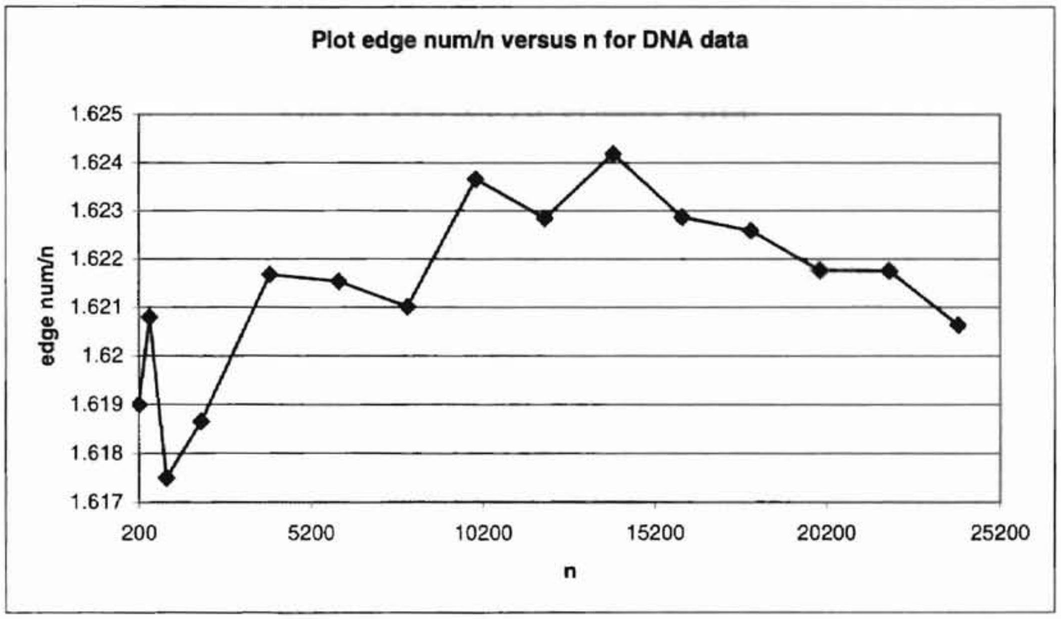

Figure 17: Plot edge number/n versus n for DNA data

CHAPTER IV

APPLICATIONS OF SUFFIX TREE

For convenience, I just use one algorithm to test the applications of suffix tree. It is McCreight's algorithm.

4.1 String Matching

We know that a suffix tree is a very useful data structure in solving string matching problems. When we have a text string and want to know if some pattern strings are included in the text string and how many times they appear in the text string, we can take advantage of the suffix tree. Using the text string as input, we can construct a suffix tree in linear time (use either of the two algorithms).

Since a path from the root of a suffix tree to a leaf represents a suffix of the text string, we can search the pattern from the root of the suffix tree to an internal node in linear time to find out if the pattern appears in the text string. And since the number of leaves of the subtree (the root of the subtree is the internal node that the end of pattern belongs to) is equal to the number of times it is repeated, it is not difficult to get the repeated times of the pattern by calling a recursive function.

Using the English text data, I ran my programs to construct a suffix tree and try to search for some patterns from the tree.

```
Script started on Mon May 24 16:39:12 1999
[ltongyu@Linux-home thesis]$ out
System time for constructing suffix tree is: 0.270000
Please input the pattern string you want to search for.
 the
The pattern was found.
The number of times the pattern appears is 185
Do you want to continue to search for some patterns?
y
Please input the pattern string you want to search for.
hello
No match for this pattern.
Do you want to continue to search for some patterns?
n
*************** End of Program ***************
[ltongyu@Linux-home thesis]$ exit
Script done on Mon May 24 16:39:33 1999
```

Above is a typescript file I got when I ran my programs on Linux. The length of the input file (the book) is 20228. When I typed the executable file name "out", a suffix tree was constructed and the elapsed system time was displayed, 0.27 seconds. The user then was asked to input the pattern string. When I typed a word "the", the pattern "the" was searched for in the tree and "The pattern was found." was displayed. And, when the pattern is found, the program will call some functions to get the number of times the pattern appears. We can see from the output that the number of occurrences of the pattern "the" is 185. The program will continue to ask the user if he/she wants to continue to search for patterns. If the user's answer is "y" or "Y", the program will continue.

We can see that the pattern "Hello" was not found, and "No match for this pattern." was displayed. If the user type "n" or "N" when being asked "Do you want to continue to search some patterns?", the program will terminate.

We can use a book, a dictionary, a lexicon or any other text files as the input text strings, searching for any words or sentences quickly by running the programs.

4.2 Other Applications

Bioinformatics is a very popular research area recently. It combines biological sciences and computational methods together to help the research in DNA sequence search, DNA sequence assembly, RNA sequence search, protein analyses etc.

DNA (deoxyribonucleic acid) is the primary genetic material in all living organisms - a molecule composed of two complementary strands that are wound around each other in a double helix formation. The strands are connected by base pairs that look like rungs in a ladder. Each base will pair with only one other: adenine (A) pairs with thymine (T), guanine (G) pairs with cytosine (C). The sequence of each single strand can therefore be deduced by the identity of its partner.

Genes are sections of DNA that code for a defined biochemical function, usually the production of a protein.

DNA is composed of four kinds of components: A, T, C and G. When I ran the programs to analyze the system time and the space it took to construct a suffix tree for the two algorithms, I wrote one program to generate different sizes of random text strings over the alphabet: A, T, C and G.

4.2.1 DNA Sequence Search

We can use the DNA sequence of a real gene as input text string data to construct a suffix tree. We use another small piece of DNA sequence as the pattern string to find out if the piece of DNA is a part of the gene and how many times it is repeated. This is very useful in biological research to determine the character or effects of a certain DNA piece in the whole gene.

My thesis is not focused on Bioinformatics, but I tried to test my programs in this area using some real data. The data is the DNA sequence of a gene I downloaded from a project of University of Washington Genome Center. The name of the project is HLA class one locus. The size of the gene is 16888 and it also can be found in Genbank.

Using the DNA sequence of the gene as input text data, I tested my program using some small DNA sequences to find out if the sequences appeared in the gene and how many times they repeated.

```
Script started on Mon May 24 16:27:44 1999
[ltongyu@Linux-home thesis]$ out
System time for constructing suffix tree is: 0.250000
Please input the pattern string you want to search for.
AGAAGAT
The pattern was found.
The number of times the pattern appears is 2
Do you want to continue to search for some patterns?
y
Please input the pattern string you want to search for.
GAAGATTTC
The pattern was found.
The number of times the pattern appears is 1
Do you want to continue to search for some patterns?
y
Please input the pattern string you want to search for.
AGAAGATTTC
No match for this pattern.
Do you want to continue to search for some patterns?
n
*************** End of Program ***************
[ltongyu@Linux-home thesis]$ exit
Script done on Mon May 24 16:28:26 1999
```

From the above output, we can see that the system time it took to construct the suffix tree was 0.25 seconds. The DNA sequence AGAAGAT repeated 2 times in the gene and GAAGATTTC appeared only one time. The sequence AGAAGATTTC was not found in the gene.

We can also use the whole Genbank or part of Genbank as the text string data. For example, when somebody wants to know if one gene has already been registered in the Genbank, he/she can use the DNA sequence of the gene as a pattern to search for.

4.2.2 Other Applications in Bioinformatics

A suffix tree is also useful in RNA sequence search, protein sequence analyses and DNA error copies search, etc.

CHAPTER V

SUMMARY, CONCLUSIONS AND FUTURE WORK

5.1 Summary

The suffix tree is a very important data structure in string search algorithms. It provides linear time solutions to many string matching problems.

McCreight's algorithm inserts the suffixes of the text string into an initially empty tree, starting with the longest suffix. The method is not on-line, and the intermediate trees are not suffix trees.

Ukkonen's algorithm reads the text string from left to right, character by character, and incrementally constructs suffix trees for the prefixes of the text string seen so far. It is on-line construction.

In my thesis, I implemented two linear time algorithms to construct a suffix tree – McCreight's algorithm and Ukkonen's algorithm in C++; compared the time and space used by the two algorithms using some random data; tested the applications of suffix trees in string matching and Bioinformatics.

5.2 Conclusions

5.2.1 Time Complexity

From the outputs of the programs and graphs, I verified empirically that the time complexity of both algorithms is O(n).

I compared the time spent by the two algorithms by running the programs using certain sets of data. I found out that McCreight's algorithm is more efficient than Ukkonen's algorithm in the time being used in constructing a suffix tree.

The difference in performance of the two algorithms is not significant.

5.2.2 Space Complexity

Since the definition of the suffix tree for the two algorithms is same, the structure of the suffix tree should be the same. I verified this point through the programs. The space spent by the suffix trees constructed by the two algorithms for a certain input text string is exactly the same. The space complexity is O(n).

5.2.3 Applications of Suffix Tree

A suffix tree is a very useful data structure that embodies a compact index to all the distinct, non-empty substrings of a given text string. The suffix tree is not only useful in plain English text string searching but is also useful in many areas of Bioinformatics, for example, DNA sequence search, DNA sequence assembly, RNA sequence search and protein sequence analyses, etc.

5.3 Future Work

The suffix tree is a trie-like data structure representing all suffixes of a string. Such trees have a central role in many algorithms on strings. I have already tested string matching and DNA sequence searching in my thesis.

In practical pattern-matching applications, exact matching is not always pertinent. It is often more important to find objects that match a given pattern in a reasonably approximate way. This is approximate pattern matching, and is very useful in Bioinformatics to find out the effects of the sets of analogue DNA, RNA or protein in living organisms.

Suffix tree can be used to solve the approximate string matching problem. Beside the applications in Bioinformatics, research and implementation of the algorithms in determining longest common substrings; sequentially compressing data; ascertaining whether or not a given string is square-free are also very interesting areas in the future study of suffix trees.

REFERENCES

[1] A. V. Aho, J. E. Hopcroft and J. D. Ullman, *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, Mass., 1974, Ch. 9, 317-361.

[2] A. Apostolico and F. P. Preparata, "Optimal off-line detection of repetitions in a string," *Theoretical Computer Science*, 22: 297-315, 1983.

[3] A. Apostolico and W. Szpankowski, "Self-alignments in words and their applications," *Journal of Algorithms*, 446-467, 1992.

[4] R. S. Boyer and J. S. Moore, "A fast string search algorithm," *Communications of the ACM*, 20 (10): 762-772, 1977.

[5] W. Chang and E. Lawler, "Approximate string matching in sublinear expected time," *In proceedings of 32nd IEEE Sumposium on Foundations of Computer Science*, 116-124, 1990.

[6] M. Crochemore and W. Rytter, *Text Algorithms*. Oxford University Press, Oxford, UK, 1994.

[7] L. Devroye, W. Szpankowski and B. Rais, "A note on the height of suffix trees," *SIAM Journal on Computing*, 21(1): 48-53, 1992.

[8] Z. Galil and K. Park, "An improved algorithm for approximate string matching," *SIAM Journal on Computing*, 19: 989-999, 1990.

[9] R. Giegerich and S. Kurtz, "A comparison of imperative and purely functional suffix tree constructions," *Science of Computer Programming* 25: 187-218, 1995.

[10] R. Giegerich and S. Kurtz, "From Ukkonen to McCreight and Weiner: A Unifying View of Linear-Time Suffix Tree Construction," *Algorithmica*, 19: 331-353, 1997.

[11] R. M. Karp and M. O. Rabin, "Efficient randomized pattern-matching algorithms," *IBM J. Res. Develop.*, 31 (2): 249-260, March 1987.

[12] D. E. Knuth, *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*, Addison-Wesley, Reading, Mass., 1968, 305-422.

[13] D. E. Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching*. Addison-Wesley, Reading, Mass., 1973, Ch. 6.3, 490-493.

[14] D. E. Knuth, J. H. Morris and V. R. Pratt, "Fast pattern matching in strings," *SIAM Journal on Computing*, 6(2): 322-350, 1977.

[15] E. McCreight, "A space-economical suffix tree construction algorithm," *J. Assoc. Comput.* Mach., 23 (1976), 262-272.

[16] M. Rodeh, V. Pratt and S. Even, "Linear algorithm for data compression through string matching," *Journal of the ACM*, 28: 16-24, 1981.

[17] G. A. Stephen, *String Searching Algorithms*. World Scientific, Singapore, 1994.

[18] E. Ukkonen and D. Wood, "Approximate string matching with suffix automata," *Algorithmica,* 10 (1993), 353-364.

[19] E. Ukkonen, "On-line Construction of Suffix-Trees," *Algorithmica*, 14: 249-260, 1995.

[20] P. Weiner, "Linear pattern matching algorithms," *Proc. IEEE 14th Ann. Symp. On Switching and Automata Theory*, 1973, 1-11.

APPENDICES

APPENDIX A: SOURCE CODES FOR IMPLEMENTATION OF McCREIGHT'S AND UKKONEN'S ALGORITHMS

```
////////////////////////////////////////////////////////////////////////
//
// Contents: suffix_tree.h
//
// Tongyu Li
//
// Date: May, 1999
//
////////////////////////////////////////////////////////////////////////

//
// This is the header file of suffix_tree, which is also the parent of
// MCC and Ukk.
//

#ifndef SUFFIX_TREE_H
#define SUFFIX_TREE_H

#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

#define N 30000
#define MAX_HASH_TABLE 505605

typedef struct{
      int begin_node_index; //the index of begin node in the edge
      int end_node_index; //the index of end node in the edge
      int first_char; //the index of first character in the edge
      int last_char; //the index of last character in the edge
}edge;

class Suffix_tree{

public: //functions

      Suffix_tree(); //constructor
      int hash_function(int, int); //hash function
      edge  find_edge(int, int); //find the edge
      bool insert_to_hash_table(int, int, int); //insert edge
      bool insert_to_hash_table(int, int, int, int);
            //insert edge (overload)
      bool remove_from_hash_table(int, int); //remove the edge
      char get_element(int i){return text_string[i];}
            //get string's element
      //get the last element of text string
      char get_last_element()
            {return text_string[text_string_length-1];}
      int get_length(){return text_string_length;} //get string length
      void add_suffix_link(int i, int su){nodes[i]=su;}
            //add suffix link
      int get_suffix_link(int i){return nodes[i];} //get suffix link
      int get_next_node(){return nodes_num++;}
            //get the next node number
      int get_node_num(){return nodes_num;} //get node number
      void print_edges(); //print all the used edges
      edge search_string(char*); //search the matching string
      bool check_tree(); //check if the suffix tree is correct
      void appear_times(int);
            //calculate the number of appearence of a pattern
      int get_app_times(){return app_times;} //get the appear times
      double get_add_time(){return add_time;}
      double get_min_time(){return min_time;}
      int get_edge_num();

private: //data

      ifstream in_ptr; //input file
      char text_string[N]; //text string
      int text_string_length; //text string length
      edge edge_array[MAX_HASH_TABLE]; //array of edges
      int nodes[N*2];
            //the index of the array is node number and
            //the element contains the information of suffix link
            //for this node.
      int nodes_num; //node number
      int app_times; //appear times of a pattern
      double add_time; //time need add to total time
      double min_time; //time need minus from total time
};

#endif
```

```
//////////////////////////////////////////////////////////////////////
//
// Contents: suffix_tree.cc
//
// Tongyu Li
//
// Date: May, 1999
//
//////////////////////////////////////////////////////////////////////

//
// This is the source file of suffix_tree.
//

#include "suffix_tree.h"
#include <sys/times.h>

Suffix_tree::Suffix_tree() //constructor
{
      in_ptr.open("input"); //open the input file

      if(!in_ptr) //check if opened properly
      {
            cerr<<"Cannot open the file -- input."<<endl;
            exit(1);
      }

      char* line=new char[257];
      int i, j=0;

      while(in_ptr.getline(line, 256)) //read the file line by line
            for(i=0; i<strlen(line); i++)
                  text_string[j++]=line[i];
      text_string[j]='#';
            //add a special character to the end of string
      text_string[j+1]='\0'; //the end of string
      text_string_length=strlen(text_string); //string length is j+2
      for(i=0; i<N*2; i++) //initialize the array of nodes
            nodes[i]=-1;

      for(i=0; i<MAX_HASH_TABLE; i++) //initialize the array of edges
      {
            edge_array[i].begin_node_index=-1;
            edge_array[i].end_node_index=-1;
            edge_array[i].first_char=-1;
            edge_array[i].last_char=-1;
      }

      nodes_num=0; //initialize node_num to 0
      delete[] line;
      in_ptr.close();
      app_times=0;
      add_time=0;
      min_time=0;
}
```

```
//
//The edges of suffix tree are stored in the hash table. The hash
//function is decided by the node that the edge is connected to and
//the first character of the edge. In order to enlarge the difference
//of nodes, multiply node times 109 before adding character.
//

int Suffix_tree::hash_function(int node, int character)
{
      return((node*109  + character) % MAX_HASH_TABLE);
}

//find the edge from hash table
edge Suffix_tree::find_edge(int node, int character)
{
      int a=hash_function(node, character); //call hash function first

      int count=0;
      int ch;

      double temp;

      struct tms tms_st, tms_fin;
      clock_t st, fin;

      st=times(&tms_st);

      while(count<=MAX_HASH_TABLE) //loop times <= hash table size
      {
            if(edge_array[a].begin_node_index==node) //if find
              {
                  ch=text_string[edge_array[a].first_char];
                  if(ch==character)
                        return edge_array[a];
            }

            if(edge_array[a].begin_node_index==-1) //if not find
                  return edge_array[a];
            count++;
            a=(++a)%MAX_HASH_TABLE;
      }

      fin=times(&tms_fin);

      temp=(tms_fin.tms_stime-tms_st.tms_stime)/(double)CLK_TCK;

      if(count>1)
      {
            min_time += temp;
            add_time += temp/(double)count;
      }
}

//
//There are two situations when inserting an edge to hash function.
//First, we don't know the end node. Second, the end node already
//exists.
```

```
//

//insert edge to hash table
bool Suffix_tree::insert_to_hash_table(int node, int begin_index, int
end_index)
{
	//call hash function
	int h = hash_function(node, text_string[begin_index]);

	int count=0;

	double temp;

	struct tms tms_st, tms_fin;
	clock_t st, fin;

	st=times(&tms_st);

	//if the slot is not empty
	while(edge_array[h].begin_node_index != -1 &&
	edge_array[h].begin_node_index != -2 && count <= MAX_HASH_TABLE)
	{
		h=(++h)%MAX_HASH_TABLE;
		count++;
	}

	fin=times(&tms_fin);
	temp=(tms_fin.tms_stime-tms_st.tms_stime)/(double)CLK_TCK;

	if(count<=MAX_HASH_TABLE) //find a proper slot
	{
		edge_array[h].begin_node_index = node;
		edge_array[h].end_node_index = nodes_num++;
		edge_array[h].first_char = begin_index;
		edge_array[h].last_char = end_index;

		if(count>1)
		{
			min_time += temp;
			add_time += temp/(double)count;
		}
		return true;
	}

	return false; //not find a proper slot
}

//overload function of insert
bool Suffix_tree::insert_to_hash_table(int node, int end_node, int
begin_index, int end_index)
{
	//call hash function
	int h = hash_function(node, text_string[begin_index]);
	int count=0;

	double temp;
```

```
        struct tms tms_st, tms_fin;
        clock_t st, fin;

        st=times(&tms_st);

        //if the slot is not empty
        while(edge_array[h].begin_node_index != -1 &&
        edge_array[h].begin_node_index != -2 && count <= MAX_HASH_TABLE)
        {
                h=(++h)%MAX_HASH_TABLE;
                count++;
        }

        fin=times(&tms_fin);
        temp=(tms_fin.tms_stime-tms_st.tms_stime)/(double)CLK_TCK;

        if(count<=MAX_HASH_TABLE) //find a proper slot
        {
                edge_array[h].begin_node_index = node;
                edge_array[h].end_node_index = end_node;
                edge_array[h].first_char = begin_index;
                edge_array[h].last_char = end_index;

                if(count>1)
                {
                        min_time += temp;
                        add_time += temp/(double)count;
                }

                return true;
        }

        return false; //not find a proper slot
}

//
//Remove an edge from hash table. First, find the index of the edge.
//We will not remove it really, just change the begin node index to -2.
//

bool Suffix_tree::remove_from_hash_table(int node, int begin_index)
{
        int a = hash_function(node, text_string[begin_index]);
        int count = 0;

        double temp;

        struct tms tms_st, tms_fin;
        clock_t st, fin;

        st=times(&tms_st);

        while(edge_array[a].begin_node_index != node ||
        edge_array[a].first_char!= begin_index)
        {
                if(count>MAX_HASH_TABLE)
                        break;
                a = (++a)%MAX_HASH_TABLE;
                count++;
        }

        fin=times(&tms_fin);
        temp=(tms_fin.tms_stime-tms_st.tms_stime)/(double)CLK_TCK;

        if(count<=MAX_HASH_TABLE)
        {
                edge_array[a].begin_node_index=-2;

                if(count>1)
                {
                        min_time += temp;
                        add_time += temp/(double)count;
                }

                return true;
        }
        return false;
}

//print the edges of suffix tree
void Suffix_tree::print_edges()
{
        int i, j;
        for(i=0; i<MAX_HASH_TABLE; i++)
        {
                if(edge_array[i].begin_node_index != -1 &&
edge_array[i].begin_node_index!=-2)
                {
                        cout<<"Begin node "<<edge_array[i].begin_node_index;
                        cout<<" End node "<<edge_array[i].end_node_index
                                <<endl;
                        cout<<"begin "<<edge_array[i].first_char<<endl;
                        cout<<"end "<<edge_array[i].last_char<<endl;
                }
        }
}

//get the total used edge number
int Suffix_tree::get_edge_num()
{
        int e=0;
        for(int i=0; i<MAX_HASH_TABLE; i++)
                if(edge_array[i].begin_node_index!=-1 &&
edge_array[i].begin_node_index!=-2)
                        e++;
        return e;
}


//search for the matching string
edge Suffix_tree::search_string(char* pattern)
{
        int m, n;
        app_times=0;

        edge e=find_edge(0, pattern[0]); //find the edge from root

        if(e.begin_node_index==-1)
              return e; //no matching

        m=e.first_char + 1;
        n=1;

        for(;;) //compare the character one by one
        {
              while(m<=e.last_char && n<strlen(pattern))
              {
                    if(text_string[m++] != pattern[n++])
                    {
                          e.begin_node_index=-1;
                          return e; //no matching
                    }
              }

              if(n==strlen(pattern)) //find
                    return e;

              e=find_edge(e.end_node_index, pattern[n]);

              if(e.begin_node_index==-1) //no matching
                    return e;

              m=e.first_char + 1;
              n++;
        }
}

//check if all the suffixes of text string are included in the
//suffix tree
bool Suffix_tree::check_tree()
{
        int n=0, i, k;
        char* p;
        edge e;
        p=new char[text_string_length];
        strcpy(p, text_string);

        while(n<text_string_length-1)
        {
              e=search_string(p);
              if(e.begin_node_index==-1) //if one suffix is not included
                    return false;
              n++;
              i=0;
              for(k=n; k<text_string_length; k++)
                    p[i++]=text_string[k];
              p[i]='\0';
        }

        return true; //if all the suffixes are included
```

```
}

//find the number of times the pattern appears
void Suffix_tree::appear_times(int n)
{

       for(char a=' '; a<='~'; a++)
       {
              edge e=find_edge(n, a);
              if(e.begin_node_index != -1)
              {
                     app_times++;
                     appear_times(e.end_node_index); //call recurrsively
              }
       }
}
```

```
///////////////////////////////////////////////////////////////////////
//
// Contents: McCreight.h
//
// Tongyu Li
//
// Date: May, 1999
//
///////////////////////////////////////////////////////////////////////

//
// This is the header file of MCC, which is to implement McCreight's
// algorithm of constructing of suffix tree.
//

#ifndef MCC_H
#define MCC_H

#include "suffix_tree.h"

class MCC:public Suffix_tree{

public: //functions

      MCC(); //constructor
      bool fast_find(int); //fast find string using suffix links
      void slow_find(); //find string one character by one character
      void create_suffix_tree(); //create the suffix tree
      void check_nodes(); //check nodes
      edge Is_implicit(int); //check if the edge is implicit

private: //data

      edge father; //father edge
      edge head; //head edge
      int root; //root of tree
      int node_v; //node v
      int head_n; //next head (node)
      int curr_j; //current scan character index
};

#endif
```

```
////////////////////////////////////////////////////////////////////////
//
// Contents: McCreight.cc
//
// Tongyu Li
//
// Date: May, 1999
//
////////////////////////////////////////////////////////////////////////

//
// This is the source file of MCC.
//

#include "McCreight.h"

MCC::MCC() //constructor, initialize the private data
{
	head.begin_node_index=-1;
	head.end_node_index=-1;
	head.first_char=-1;
	head.last_char=-1;

	father=head;
	root=-1;
	curr_j=-1;
}

bool MCC::fast_find(int node_u) //fast find father edge from node_u
{
	edge e=find_edge(node_u, get_element(father.first_char));

	if(e.begin_node_index==-1) //root situation
	{
		if(curr_j==get_length()-1) //if the last character
		{
			insert_to_hash_table(root, curr_j, curr_j);
			curr_j++;
			return true;
		}

		//if curr_j is not the last character
		insert_to_hash_table(root, curr_j, curr_j);
		father=find_edge(root, get_element(curr_j));
		head_n=father.end_node_index;
		curr_j++;
		node_v=root;
		return true;
	}

	//if find the edge

	if(node_u!=root) //if node_u is not root, update curr_j
		curr_j=head.first_char-(father.last_char-
			father.first_char+1);

	int l1=e.last_char-e.first_char;
int l2=father.last_char-father.first_char;

if(l1==l2) //if father edge is equal to e edge
{
      node_v=e.end_node_index;
      curr_j += l1+1;
      father=e;
      return false;
}

edge e1;

if(l1>l2) //if e edge is longer than father edge, split e edge
{
      remove_from_hash_table(node_u, e.first_char);
      insert_to_hash_table(node_u, e.first_char,
            (e.first_char+l2));
      e1=find_edge(node_u, get_element(e.first_char));
      insert_to_hash_table(e1.end_node_index, e.end_node_index,
            (e.first_char+l2+1), e.last_char);
      node_v=e1.end_node_index;
      head_n=node_v;
      father=e1;
      curr_j += l2+1;
      return true;
}

//if edge is shorter than father edge, continue to search

int i=father.first_char;

while(l1<l2) //continue to search edge by edge
{
      i += l1+1;
      e=find_edge(e.end_node_index, get_element(i));
      l2 -= l1+1;
      l1=e.last_char-e.first_char;
} //end while

if(l1==l2) //l1=l2
{
      node_v=e.end_node_index;
      curr_j += father.last_char - father.first_char + 1;
      father=e;
      return false;
}

//if l1>l2, split the edge
remove_from_hash_table(e.begin_node_index, e.first_char);
insert_to_hash_table(e.begin_node_index, e.first_char,
      e.first_char+l2);
e1=find_edge(e.begin_node_index, get_element(e.first_char));
insert_to_hash_table(e1.end_node_index, e.end_node_index,
      e.first_char+l2+1, e.last_char);
node_v=e1.end_node_index;
head_n=node_v;
curr_j += father.last_char-father.first_char+1;
```

```
        father=e1;
        return true;
}

void MCC::slow_find() //slow find, character by character
{
        edge e=find_edge(node_v, get_element(head.first_char));

        int check=1;
        edge e1;

        if(e.begin_node_index != -1) //if find the edge
        {
             int m, n;
             curr_j++;
             father=e;
             m=e.first_char+1;
             n=curr_j;

             for(;;)
             {
               while(m<=e.last_char && n<get_length()) //check one by one
               {
                       if(get_element(m++)==get_element(n++))
                               curr_j++;
                       else
                       {
                               check=0;
                               break;
                       }
               }

               if(check==0) //find the point not matching, split edge
               {
                       remove_from_hash_table(e.begin_node_index,
                               e.first_char);
                       insert_to_hash_table(e.begin_node_index,
                               e.first_char, m-2);
                       e1=find_edge(e.begin_node_index,
                               get_element(e.first_char));
                       insert_to_hash_table(e1.end_node_index,
                               e.end_node_index, m-1, e.last_char);
                       father=e1;
                       head_n=e1.end_node_index;
                       return;
               }

               //the whole edge is matching, continue find next edge
               e=find_edge(e.end_node_index, get_element(n));

               if(e.begin_node_index != -1) //if find the edge
               {
                       father=e;
                       curr_j++;
                       m=e.first_char+1;
                       n++;
               }
```

```
            else //not find matching edge
            {
                  head_n=father.end_node_index;
                  return;
            }

            check=1; //update check

         }//end for(;;)

      } //end if

      if(node_v==root) //not find matching edge and node_v is root
      {
            insert_to_hash_table(root, curr_j, curr_j);
            e1=find_edge(root, get_element(curr_j));
            head_n=e1.end_node_index;
            add_suffix_link(head_n, root);
            father=e1;
            curr_j++;
      }
      else //not find the matching edge and node_v is not root
            head_n=node_v;
}

void MCC::create_suffix_tree() //create the suffix tree
{
      get_next_node();

      if(get_length()==1) //if the text string has only one character
      {
            insert_to_hash_table(0, 0, get_length() - 1);
            return;
      }

      //else insert the first edge
      insert_to_hash_table(0, 0, 0);
      father=find_edge(0, get_element(0));
      root=father.begin_node_index;
      insert_to_hash_table(father.end_node_index, 1, get_length()-1);
      head=find_edge(father.end_node_index, get_element(1));
      add_suffix_link(head.begin_node_index, root);

      edge e, e1;
      char* ch=new char[get_length()];
      int i, u, f;

      for(i=1; i<get_length(); i++) //insert the edges one by one
      {
            curr_j=i; //current scan character index is i

            u=get_suffix_link(father.begin_node_index);
                  //get suffix link

            if(u!=-1) //if find a suffix link
            {
                  if(!fast_find(u))
```

```
                        slow_find();
                }

                else //root situation
                {
                        node_v=root;
                        slow_find();
                }

                if(curr_j < get_length())
                        //if curr_j < string length, insert
                {
                        if(head.begin_node_index != node_v) //add suffix link
                                add_suffix_link(head.begin_node_index, node_v);

                        insert_to_hash_table(head_n, curr_j, get_length()-1);

                        //update head
                        head=find_edge(head_n, get_element(curr_j));
                }
        } //end for
}

void MCC::check_nodes() //check the nodes next to root
{
        edge e, e1;
        char a;

        for(a=' '; a<='~'; a++)
        {
                e=find_edge(root, a);

                if(e.begin_node_index != -1)
                {
                        e1=Is_implicit(e.end_node_index);

                        if(e1.begin_node_index != -1) //if implicit, remove
                        {
                                remove_from_hash_table(e.begin_node_index,
                                        e.first_char);
                                remove_from_hash_table(e1.begin_node_index,
                                        e1.first_char);
                                insert_to_hash_table(root, e1.end_node_index,
                                        e.first_char, e1.last_char);
                        }
                }
        }
}

edge MCC::Is_implicit(int n) //check if the node is implicit
{
        char a;
        edge e, e1;
        int i=0;
        e1.begin_node_index=-1;

        for(a=' '; a<='~'; a++)
        {
              e=find_edge(n, a);

              if(e.begin_node_index != -1)
              {
                    i++;
                    if(i>=2) //if edge number is >= 2, then explicit
                    {
                          e1.begin_node_index=-1;
                          return e1;
                    }
                    else
                          e1=e;
              }
        }

        return e1; //only one edge under the node, implicit
  }
```

```
////////////////////////////////////////////////////////////////////////
//
// Contents: Ukkonen.h
//
// Tongyu Li
//
// Date: May, 1999
//
////////////////////////////////////////////////////////////////////////

//
// This is the header file of Ukk, which is to implement Ukkonen's
// algorithm for constructing a suffix tree.
//

#ifndef UKK_H
#define UKK_H

#include "suffix_tree.h"

class Ukk:public Suffix_tree{

public: //functions

      Ukk(); //constructor
      bool Is_explicit() const //if edge is explicit
      {return (active_point.last_char<active_point.first_char)
            ? true : false;}
      void split_edge(edge); //split the edge
      bool test_and_split(); //test and split
      void canonize(); //canonize
      void update(); //update the tree
      void create_suffix_tree(); //create the suffix tree

private: //data

      edge active_point;
                  //active point may not be the whole edge, so the
                  //end_node_index can be ignored, active_point is a
                  //part of one edge.
      int state_r; //the state r
      int old_r; //the old state
      int curr_scan; //the last index of the string that being scanned
};

#endif
```

```
////////////////////////////////////////////////////////////////////////
//
// Contents: Ukkonen.cc
//
// Tongyu Li
//
// Date: May, 1999
//
////////////////////////////////////////////////////////////////////////

//
// This is the source file of Ukk.
//

#include "Ukkonen.h"

Ukk::Ukk() //constructor, initialize the private data
{
      active_point.begin_node_index=0;
      active_point.end_node_index=-1;
      active_point.first_char=0;
      active_point.last_char=-1;

      state_r=0;
      old_r=-1;
      curr_scan=0;
}

void Ukk::split_edge(edge e) //split the edge
{
      //remove the original edge first
      remove_from_hash_table(e.begin_node_index, e.first_char);

      int m=e.first_char + active_point.last_char -
            active_point.first_char;

      //insert the first part of edge
      insert_to_hash_table(e.begin_node_index, e.first_char, m);

      edge e1=find_edge(e.begin_node_index, get_element(e.first_char));

      //add the suffix link
      add_suffix_link(e1.end_node_index,
            active_point.begin_node_index);

      //insert the second part of edge
      insert_to_hash_table(e1.end_node_index, e.end_node_index, m+1,
            e.last_char);

      state_r=e1.end_node_index; //update state_r
}

bool Ukk::test_and_split() //test and split
{
      edge e;

      if(Is_explicit()) //if the active edge is explicit
      {
            e=find_edge(state_r, get_element(curr_scan));

            if(e.begin_node_index!=-1) //if find the edge
                  return true;
            else //if not find the edge
                  return false;
      }

      //if the edge is implicit
      e=find_edge(state_r, get_element(active_point.first_char));

      int i=e.first_char+active_point.last_char-
            active_point.first_char+1;

      if(get_element(i)==get_element(curr_scan)) //if find the matching
            return true;

      //if not find the matching, split the edge
      split_edge(e);

      return false;
}

void Ukk::canonize() //canonize
{
   if(!Is_explicit()) //if the edge is implicit
   {
       edge e=find_edge(active_point.begin_node_index,
            get_element(active_point.first_char));

       int i=e.last_char - e.first_char;

      //update active_point.first_char and i
      while(i <= (active_point.last_char - active_point.first_char))
      {
         active_point.first_char += i + 1;
         active_point.begin_node_index = e.end_node_index;

         if(active_point.first_char <= active_point.last_char)
         {
             e=find_edge(active_point.begin_node_index,
                  get_element(active_point.first_char));
             i=e.last_char - e.first_char;
         }
      } //end while

      state_r=active_point.begin_node_index; //update state_r
   } //end if
}

void Ukk::update() //update the suffix tree
{
  old_r=0; //old state initialized to 0
  state_r=active_point.begin_node_index;

  while(!test_and_split())
  {
    //create open transition
    insert_to_hash_table(state_r, curr_scan, get_length()-1);

    if(old_r > 0) //if old_r is not root, add suffix link
      add_suffix_link(old_r, state_r);

    old_r=state_r; //update the old_r

    if(active_point.begin_node_index==0)
      active_point.first_char++;
    else
      active_point.begin_node_index=get_suffix_link
                         (active_point.begin_node_index);

    state_r=active_point.begin_node_index; //update state_r

    canonize();

  } //end while

  if(old_r > 0) //if old_r is not root, add suffix link
      add_suffix_link(old_r, state_r);

  active_point.last_char++; //update active point

  canonize();
}

void Ukk::create_suffix_tree() //create suffix tree
{
      for(int i=0; i<get_length(); i++)
            //update tree one char by one char
      {
            curr_scan = i;
            update();
      }
}
```

```
////////////////////////////////////////////////////////////////////////
//
// Contents: main.cc (for McCreight's algorithm)
//
// Tongyu Li
//
// Date: May, 1999
//
////////////////////////////////////////////////////////////////////////

//
// This is the test program of McCreight's algorithm for the whole
// project.
//

#include "suffix_tree.h"
#include "Ukkonen.h"
#include "McCreight.h"
#include <iomanip.h>
#include <sys/times.h>

#define MAX_LEN 80

main()
{
      char* p=new char[MAX_LEN];
      char* answer=new char[80+1];
      int app_t; //appear times
      edge e;
      double sys_time;
      struct tms tms_start, tms_finish;
      clock_t start, finish;

      MCC m; //initialize the object

      cout<<setiosflags(ios::fixed)<<setprecision(6);

      start=times(&tms_start); //start to count time
      m.create_suffix_tree();
      finish=times(&tms_finish); //finish counting time

      sys_time=(tms_finish.tms_stime-tms_start.tms_stime)
                  /(double)CLK_TCK;
      sys_time += m.get_add_time(); //add the necessary extra time
      sys_time -= m.get_min_time(); //minus the useless time

      cout<<"System time for constructing suffix tree is: "
                  <<sys_time<<endl;

      m.check_nodes(); //check the nodes

      m.print_edges(); //print all the edges in the tree

      if(m.check_tree()) //check if the suffix tree is correct
            cout<<"After checking, the suffix tree is correct."<<endl;
      else //if the tree is not correct
      {
            cerr<<"The tree is not correct."<<endl;
            exit(1);
      }

      do{
            cout<<"Please input the pattern string you want to search
                        for."<<endl;
            cin.getline(p, MAX_LEN-1); //get the pattern

            e=m.search_string(p); //search the pattern

            if(e.begin_node_index!=-1) //if found
            {
                  cout<<"The pattern was found."<<endl;

                  m.appear_times(e.end_node_index); //get appear times

                  app_t=m.get_app_times();

                  if(app_t==0) //if pattern appears one time
                        cout<<"The number of times the pattern appear
                              is 1"<<endl;
                  else
                        cout<<"The number of times the pattern appear
                              is "<<app_t<<endl;
            }

            else //if pattern not found
                  cout<<"No match for this pattern."<<endl;

            cout<<"Do you want to continue to search for some
                        patterns?"<<endl;
            cin.getline(answer, 80);

      }while(answer[0]=='y' || answer[0]=='Y'); //continue to search

      cout<<"*************** End of Program ***************"<<endl;
}
```

```
///////////////////////////////////////////////////////////////////////
//
// Contents: main.cc (for Ukkonen's algorithm)
//
// Tongyu Li
//
// Date: May, 1999
//
///////////////////////////////////////////////////////////////////////

//
// This is the test program of Ukkonen's algorithm for the whole
// project.
//

#include "suffix_tree.h"
#include "Ukkonen.h"
#include "McCreight.h"
#include <iomanip.h>
#include <sys/times.h>

#define MAX_LEN 80

main()
{
      char* p=new char[MAX_LEN];
      char* answer=new char[80+1];
      int app_t; //appear times
      edge e;
      double sys_time;
      struct tms tms_start, tms_finish;
      clock_t start, finish;

      Ukk m; //initialize the object

      cout<<setiosflags(ios::fixed)<<setprecision(6);

      start=times(&tms_start); //start to count time
      m.create_suffix_tree();
      finish=times(&tms_finish); //finish counting time

      sys_time=(tms_finish.tms_stime-tms_start.tms_stime)
                  /(double)CLK_TCK;
      sys_time += m.get_add_time(); //add the necessary extra time
      sys_time -= m.get_min_time(); //minus the useless time

      cout<<"System time for constructing suffix tree is: "
            <<sys_time<<endl;

      m.print_edges(); //print all the edges in the tree

      if(m.check_tree()) //check if the suffix tree is correct
            cout<<"After checking, the suffix tree is correct."<<endl;
      else //if the tree is not correct
      {
            cerr<<"The tree is not correct."<<endl;
            exit(1);
```

```
        }

        do{
              cout<<"Please input the pattern string you want to search
                    for."<<endl;
              cin.getline(p, MAX_LEN-1); //get the pattern

              e=m.search_string(p); //search the pattern

              if(e.begin_node_index!=-1) //if found
              {
                    cout<<"The pattern was found."<<endl;

                    m.appear_times(e.end_node_index); //get appear times

                    app_t=m.get_app_times();

                    if(app_t==0) //if pattern appears one time
                          cout<<"The number of times the pattern appear
                                is 1"<<endl;
                    else
                          cout<<"The number of times the pattern appear
                                is "<<app_t<<endl;
              }

              else //if pattern not found
                    cout<<"No match for this pattern."<<endl;

              cout<<"Do you want to continue to search for some
                    patterns?"<<endl;
              cin.getline(answer, 80);

        }while(answer[0]=='y' || answer[0]=='Y'); //continue to search

        cout<<"*************** End of Program ***************"<<endl;
  }
```

```
#
# Makefile
#

out:  suffix_tree.o Ukkonen.o McCreight.o main.cc
      g++ -g -o out suffix_tree.o Ukkonen.o McCreight.o main.cc

suffix_tree.o:    suffix_tree.h suffix_tree.cc
      g++ -c -g suffix_tree.cc

Ukkonen.o:  Ukkonen.h Ukkonen.cc
      g++ -c -g Ukkonen.cc

McCreight.o:      McCreight.h McCreight.cc
      g++ -c -g McCreight.cc

clean:
      rm -f *.o
```

APPENDIX B: SOURCE CODES FOR GENERATING RANDOM DNA DATA

```
///////////////////////////////////////////////////////////////////////////
//
// Contents: random_char.cc
//
// Tongyu Li
//
// Date: May, 1999
//
///////////////////////////////////////////////////////////////////////////

//
// This is the program that generates random DNA data.
//

#include <iostream.h>
#include <fstream.h>
#include <math.h>
#include <stdlib.h>

#define A 16807
#define M 2147483647
#define Q 127773
#define R 2836

#define size 20000

float get_random(); //get random float number between 0 and 1
int get_JRand(int, int); //get a random integer
char shuffle(char*); //shuffle function

long int seed=6;

main()
{
      char* s=new char[4+1];
      s[0]='A';
      s[1]='C';
      s[2]='G';
      s[3]='T';
      s[4]='\0';

      char* name=new char[3];
      char c;

      int i, j;

      for(i=1; i<=20; i++) //name the output file and write to the file
      {
            if(i<10)
            {
                  name[0]=i+48;
                  name[1]='\0';
            }
            else
            {
                  name[0]=i/10+48;
                  name[1]=i%10+48;
                  name[2]='\0';
            }

            ofstream out(name); //open the file

            for(j=0; j<size; j++)
            {
                  if(j!=0 && j%60==0) //there are 60 char in one line
                        out.put('\n');

                  out.put(shuffle(s)); //write to the file
            }

            out.close(); //close the file
      }
}

float get_random() //get a pseudorandom float number between 0 and 1
{
      float URand;

      do{
            seed = seed % Q * A - seed / Q * R;

            if(seed < 0)
                  seed = seed + M;
            URand = (float)seed/M;
      }while(URand <=0 || URand >= 1);

      return URand;
}

int get_JRand(int JL, int JH)
      //get a pseudorandom integer between JL and JH
{
      int JRand;
      float URand;

      do{
            URand = get_random();
            JRand = JL + (int)((JH + 1 - JL) * URand);
      }while(JRand > JH);

      return JRand;
}

char shuffle(char* string) //shuffle function
{
      int i, j, temp;

      for(i=strlen(string)-1; i>=1; i--)
      {
            j=get_JRand(0, i);
            temp=string[i];
            string[i]=string[j];
            string[j]=temp;
        }
        return string[0];
}
```

VITA

Tongyu Li

Candidate for the Degree of

Master of Science

Thesis: IMPLEMENTATION OF SUFFIX TREES USING TWO LINEAR TIME ALGORITHMS AND THEIR APPLICATION TO STRING MATCHING

Major Field: Computer Science

Biographical:

Personal Data: Born in Shanxi, China, August 12, 1970, daughter of Mr. Dechong Li and Mrs. Xianyun Liu.

Education: Received Bachelor of Science in Biology from Hebei University, Baoding, China, in July 1992. Completed requirements for the Master of Science Degree in Computer Science at the Computer Science Department at Oklahoma State University in December 1999.

Experience: Working as a sales representative in Tianjin Light Industrial Products Import and Export Corporation, China, from August 1992 to September 1996.