RESEARCH ARTICLE OPEN ACCESS

Edge-Oriented On-line Construction of Generalized Suffix Tree

Jinchao Guo[1], Jiuhui Pan[2]
[1]PG Student, Department of computer, Jinan University, China
[2]Professor, Department of computer, Jinan University, China

----------**********----------

Abstract:

The Suffix tree is a versatile data structure in string processing. Among the available construction algorithms, the algorithms based on suffix links are popular because they can operate online in linear time. However, the original algorithm based on node-oriented suffix links takes too much time to look up the correct branch. To improve the efficiency, our approach creates suffix links between edges. Relatively, it reduces the number of branch looking up operations and maintains the worst case linear time complexity.

Keywords— **Generalized Suffix tree, Linear-time algorithm, Edge-oriented suffix links**

----------**********----------

1 INTRODUCTION

The suffix tree is a trie-like data structure which contains all the suffixes of the given sequence. It is a versatile data structure in string processing, and it has been proven in [1].

In 1973, Weiner [2] introduced suffix trees and firstly gave a linear-time right-to-left algorithm to construct the tree. The construction was greatly simplified by McCreight [3] in 1976. His algorithm was left-to-right but it was off-line, which means the sequence has to be scanned before suffix tree construction procedure starts up. Twenty years later, Ukkonen [4] derived the first linear-time left-to-right on-line algorithm. The Ukkonen's algorithm [4] is for streaming sequences that it processes the sequence symbol by symbol from left to right, and the tree constructed is always the suffix tree for the scanned part of the sequence.

In all these algorithms, branch which identifies the correct outgoing edge of a given node whose label starts with a given symbol is time-consuming. M Senft and T Dvořák [5] tried to improve the performance with reduced branch operations. Instead of using the traditional top-down descent, they chose a simpler bottom-up climbing. However, on one hand, their algorithm produces an extra cost to maintain leaf suffix links; on the other hand, it requires $\Omega(n^{1.5})$ time in the worst case that the climbing performs too more times.

Our work replaces the node-oriented suffix link with the edge-oriented suffix link. With the help of it, the correct outgoing edge is identified at once.

The rest of this paper is organized as follows. First, the related definition of the suffix tree is discussed in Section 2. Secondly, the related construction algorithms and out approach are introduced in Section 3. Finally, the conclusion is presented in Section 4.

I. PRELIMINARIES

A. *Strings*

We use Σ to denote the alphabet. A string is a finite length sequence of symbols from Σ. And then the empty string denoted by ε is the string with no symbols. In this paper, xy denotes the concatenation of string x and string y.

We denote the suffix tree over a string $T = t_0 t_1 \cdots t_{n-1}$ of length $|T| = n$ by $ST(T)$. Each string x such that $x = t_i \cdots t_j$ where $0 \le i \le j \le n-1$ is a substring of T, each string p such that $p = t_0 \cdots t_i$ where $0 \le i \le n-1$ is a prefix of T, and each string $T_i = t_i \cdots t_{n-1}$ where $0 \le i \le n$ is a suffix of T. A generalized suffix tree GST is a suffix tree made for a set of strings instead of a single string. It represents all suffixes from this set of string.

In suffix trees or suffix tries, there are states that correspond to the substring of T one-to-one. We denote by $\overline{x}$ the state that corresponds to a substring x.

B. *Suffix Trie and Suffix Tree*

Formally, suffix trie is an augmented DFA (deterministic finite-state automaton) which has a tree-shaped transition graph and which is augmented with the so-called suffix link and auxiliary state $\perp$. A suffix trie is a 6-tuple, $(S = P \cup \{\perp\}, \Sigma, \delta, root, F, l)$, consisting of

- a finite set of states (S) consisting of a finite set of states (P) corresponding to substring one-to-one and an auxiliary state $(\perp)$
- a finite set of input symbols called the alphabet (Σ)
- a transition function $(\delta: S \times \Sigma \to S)$
- an initial state $(root \in P)$
- a set of final states called the leaves (F)
- a suffix link $(l: P \to P)$.

The transition function δ is defined as follows. $\delta(\bar{x}, a) = \bar{y}$ for all $\bar{x}, \bar{y} \in S$ such that $y = xa$ where $a \in \Sigma$ then we can say $\bar{x}$ has an a-transition. State *root* corresponds to the empty string ε, and the set F of final states corresponds to the set of all the suffixes. Auxiliary state $\perp$ is such a state that for all $a \in \Sigma$, $\delta(\perp, a) = root$. Suffix link l is defined as a function for each state $\bar{x} \neq root$ as $l(\bar{x}) = \bar{y}$ such that $x = ay$ for some $a \in \Sigma$, and $l(root) = \perp$.

Fig. 1 shows the suffix trie over *"abcabd"*, it wastes space that each state is a node of the tree. These nodes also increase the lookup operation at the tree.

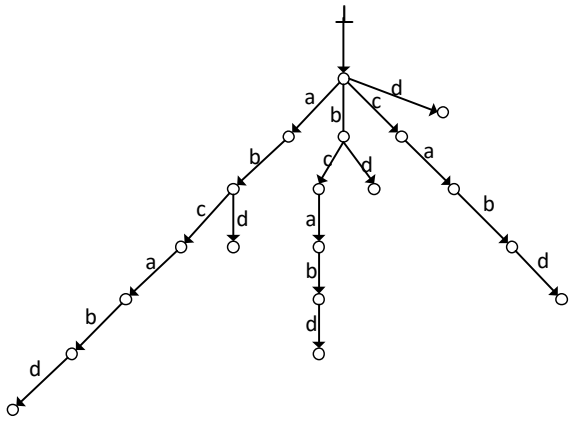

Fig. 1 Suffix trie over *"abcabd"*

Fig. 2 shows the suffix tree over *"abcabd"*, which obviously has less nodes. This is achieved by representing only a subset $P' \cup \{\perp\}$ of the states of the suffix trie. We call the states in $F' \cup \{\perp\}$ nodes. The other states are called implicit states. Set P' consists of all branching states (which have at least two outgoing edges) and all leaves (which have no outgoing edges).

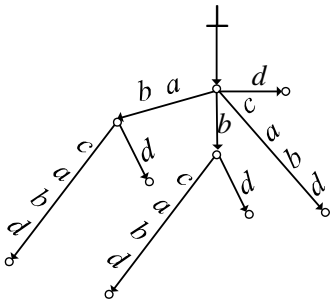

Fig. 2 Suffix tree over *"abcabd"*

In the suffix tree, each edge connects two nodes and is labelled with a non-empty substring but unlike in a suffix trie, the labels are not single symbols. Implicit states become invisible on the edge between nodes but they are still states that correspond to the substring of T one-to-one along with the nodes.

C. *Active Point and End Point*

we introduce two kinds of important states in Ukkonen's algorithm [4], active point and end point. Let $s_1 = \overline{t_1 \cdots t_{i-1}}, s_2, s_3, \cdots, s_i = root, s_{i+1} = \perp$ be the states of $ST(T_{i-1})$ on the boundary path. Obviously, $l(s_k) = s_{k+1}$ for all $1 \leq k \leq i$. Let j be the smallest index such that s_j is not a leaf, and let j' be the smallest index such that $s_{j'}$ has a t_i-transition. We call s_j the active point and s_f the end point of $ST(T_{i-1})$. As s_1 is a leaf and $\perp$ is a nonleaf that has a t_i-transition, both j and j' are well defined and $j < j'$.

II. CONSTRUCTION

A. *Base Algorithm*

Before giving details of our modification, we describe and discuss the base algorithm based on suffix links.

We present algorithm *init* in Fig. 3 and algorithm *construct* in Fig 4. To construct a ST, *init()* is evaluated at first to construct an empty suffix tree. Then *construct(s)* is executed for the string s.

```
1: init()
2: create auxiliary node ⊥, root node root and auxiliary
   edge ⊢ between them
3: end
```

Fig. 3 Algorithm *init*

```
1. construct(text : String)
2. cursor ← root
3. foreach symbol c in text do
4. step(id, c, cursor)
5. end
6. end
```

Fig 4 Algorithm *construct*

Algorithm *construct(s)* is on-line. It works in steps, from left to right. There is one step for each symbol of the text. Each step turns the old tree $ST(T_{i-1})$ into a new suffix tree $ST(T_i)$ over the string up to the current symbol c.

To update $ST(T_{i-1})$ into $ST(T_i)$, we need to add to $ST(T_{i-1})$ a t_i-transition for each of the states $s_h, 1 \leq h < j'$. For $1 \leq h < j$, we do this by expanding the existing edge because s_h is a leaf. For $j \leq h < j'$, we do this by inserting a t_i-outgoing edge. Since each edge is created to a leaf, which means the edges represent the substring ending at the current end, the existing edge expand automatically. Therefore, in each step, all we need to do are inserting t_i-outgoing edges for each of the states $s_h, j \leq h < j'$. In this paper, we use *cursor* to denote s_h. For each *text*, *cursor* is initialled to *root* at line 2 in Fig 4.

Algorithm *step* is presented in Fig. 5 and algorithm *moveDown* is presented in Fig. 6. Firstly, check if *cursor* is the end point. If so, move down the *cursor* along the edge with labels matching the current symbol, create the new suffix links, and

then break the loop to begin the next step. Otherwise, the *cursor* cannot move down any more, line 7 creates a c-outgoing edge on *cursor*. Next, line 8 creates the new suffix links and line 9 moves the *cursor* sideways using existing suffix links.

```
 1: step(id: Integer; c: Symbol; cursor: State)
 2:     loop
 3:         if moveDown(cursor, c) then
 4:             link()
 5:             break
 6:         else
 7:             split()
 8:             nLink()
 9:             moveSideways()
10:         end
11:     end
12: end
```

Fig. 5 Algorithm *step*

In step *split*, if *cursor* is a node, create a new edge and make it an outgoing edge of the *cursor*. Otherwise, the *cursor* is not a child of an edge, split the edge at *cursor* and make *cursor* a new node.

Construction algorithms differ in creating and using suffix links, what procedures *link*, *nLink* and *moveSideways* do. We show the procedure *moveSideways* in Fig. 7. The path of moving the last cursor x sideways to the new cursor x' is different. Next, we will discuss two existing algorithms and our approach respectively.

B. Ukkonen's Original Algorithm

In Ukkonen's original algorithm, if *cursor* is the end point and a node at the same time, that is to say, cursor has a t_i – outgoing edge; or if cursor is not the end point, a suffix link from the last *cursor* to the current *cursor* is created if it not exists.

```
 1: moveDown(cursor: State; c: Symbol)
 2:     if cursor is a node then
 3:         if cursor has an outgoing edge whose label starts
    with c then
 4:             move cursor down on this edge
 5:             return true
 6:         else
 7:             return false
 8:         end
 9:     else
10:         if the next symbol of cursor is c then
11:             move cursor one symbol down on the edge
12:             return true
13:         else
14:             return false
15:         end
16:     end
17: end
```

Fig. 6 Algorithm *moveDown*

As shown in Fig. 7, to find the next cursor x', follow the suffix link of u to u'. Then identify the correct edge below u' towards x'. Moving down from u' requires one or more branch operations.

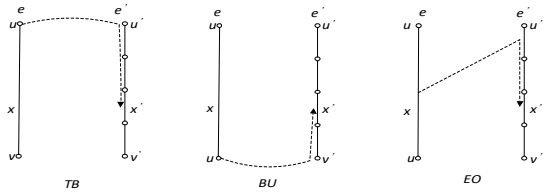

Fig. 7 procedure of *moveSideways* in Ukkonen's original top-bottom algorithm(TB),M Senft and T Dvořák's bottom-up algorithm(BU) and our edge-oriented algorithm(EO)

C. Bottom-up Approach

In M Senft and T Dvořák's bottom-up approach, creating suffix links is similar. But there is an extra cost which is in the need to maintain leaf suffix links. Based on the fact that a leaf suffix link always leads from the last created leaf to the new created leaf, suffix links are created by a smart leaf numbering.

```
 1: Procedure link
 2:     if cursor is a node then
 3:         if the last created edge has no a suffix link then
 4:             create a suffix link from it to the outgoing edge
    of cursor whose label starts with the current symbol
    (This outgoing edge has been identified before)
 5:         end
 6:         if the last new edge (the bottom part of the last
    edge split) has no a suffix link then
 7:             create a suffix link from it to the outgoing edge
    of cursor whose label starts with its first symbol (one
    branch operation)
 8:         end
 9:     end
10: end
11: Procedure nLink
12:     if the last created edge has no a suffix link then
13:         create a suffix link from it to the new created edge
14:     end
15:     if the last new edge has no a suffix link then
16:         if the edge is split on the cursor then
17:             create a suffix link from it to the bottom part of
    the edge split
18:         else
19:             create a suffix link from it to the outgoing edge
    of cursor whose label starts with its first symbol (one
    branch operation)
20:         end
21:     end
22: end
```

Fig. 8 procedure of creating edge-oriented suffix links

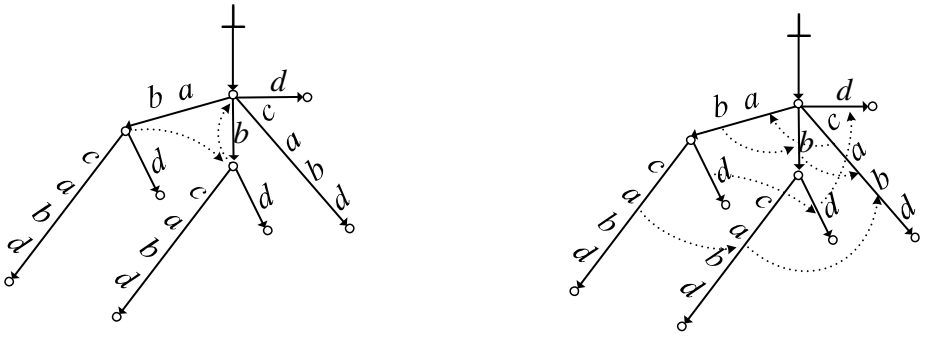

Fig. 9 Suffix tree over the string abcabd, with dotted lines showing suffix links as in original algorithm (left) and our approach (right)

In procedure *moveSideways*, as shown in Fig. 7, follow the suffix link of v to v' rather than that of its parent u to u' .There are no branch operations. But climbing from u' requires maintaining the parents of nodes. Moreover, in the worst time, climbing may be performed many steps. [5] have proven that its time complexity in worst case is not linear.

D. Our Approach

To reduce branch operations, we slightly modify suffix links. The original algorithm follows a suffix link from u to u', and identified the correct outgoing edge. We can avoid this first branch operation by having a suffix link from e pointing to e' directly. Such edge-oriented suffix links are defined as follows. If for first non-root state u on edge e and first state u' on edge e' there is a suffix link from u to u' , and the labels of edge e and e' begin with the same symbol, then there is an edge-oriented suffix link from e to e'.

As shown in Fig.9, we create one suffix link for each edge except auxiliary edge and the last created edge. When an edge is split, the top part remains and the bottom part is a new edge. The procedure of creating suffix links is presented in Fig. 8.

After splitting one edge, to find the correct destination for the suffix link of the bottom part may require a branch operation (line 7 and 19), not necessary in the node-oriented algorithms. However, the branch operation takes place once and the corresponding suffix link can be used forever. Since there are at most n edges will be created, the complexity of creating suffix links retains linear.

In procedure *moveSideways*, following the suffix link of e, we can avoid the first branch operation to move the *cursor* directly to e' but one or more extra branch operations are occasionally required. And when the edge has no a suffix link, which means the edge is the last created edge, the *cursor* can be set to the root directly.

E. Generalized Suffix Tree Construction

In the generalized suffix tree for a set of strings $D = T_1, T_2, \cdots, T_n$ of total length n, there are n leaf states corresponding to n suffixes. On one hand, we pad each string with a unique out-of-alphabet marker to ensure no suffix is a substring of another. On the other hand, to identify that which suffix each leaf state corresponds to, we mark each leaf state with a string id and the offset of the suffix in the string. It can easily be done. For each string T_i , $\mathtt{construct}(T_i)$ is executed; and after each edge created to the leaf state, we mark this leaf state with string id i and the offset of current symbol.

III. CONCLUSION

The suffix tree is important in pattern matching with a wide variety of applications. Improvements in its efficiency of construction continues to be a lively area of research. Using edge-oriented suffix links, our approach reduces the branch operations to improve on-line generalized suffix tree construction time while maintaining linear worst-case time complexity.

REFERENCES

[1] B. Smyth, "Computing patterns in strings," *Addison–Wesley,* 2003.

[2] P. Weiner, "Linear pattern matching algorithms.," *Switching and Automata Theory,* pp. 1-11, 1970.

[3] E. McCreight, "A space-economical suffix tree construction algorithm," *JACM,* pp. 262-272, 1976.

[4] E. Ukkonen, "On-line construction of suffix trees.," *Algorithmica,* pp. 249-260, 1995.

[5] T. D. M Senft, "On-line suffix tree construction with reduced branching.," *Journal of Discrete Algorithms,* pp. 48-60, 2012.