Reachability on Suffix Tree Graphs

Yasuto Higa[1], Hideo Bannai[1], Shunsuke Inenaga[1,2], and Masayuki Takeda[1,3]

[1] Department of Informatics, Kyushu University, Japan
{bannai,y-higa,shunsuke.inenaga,takeda}@i.kyushu-u.ac.jp
[2] Japan Society for the Promotion of Science
[3] SORST, Japan Science and Technology Agency (JST)

Abstract. We analyze the complexity of graph reachability queries on *ST-graph*s, defined as directed acyclic graphs (DAGs) obtained by merging the suffix tree of a given string and its suffix links. Using a simplified reachability labeling algorithm presented by Agrawal *et al.* (1989), we show that for a random string of length n, its ST-graph can be preprocessed in $O(n \log n)$ expected time and space to answer reachability queries in $O(\log n)$ time. Furthermore, we present a series of strings that require $\Theta(n\sqrt{n})$ time and space to answer reachability queries in $O(\log n)$ time for the same algorithm. Exhaustive computational calculations for strings of length $n \leq 33$ have revealed that the same strings are also the worst case instances of the algorithm. We therefore conjecture that reachability queries can be answered in $O(\log n)$ time with a worst case time and space preprocessing complexity of $\Theta(n\sqrt{n})$.

Keywords: algorithms and data structures, suffix trees, graph reachability

1 Introduction

The reachability query for two nodes u, v of a given directed graph is to answer whether or not there exists a path in the graph that starts from u and ends at v. For any given graph, the query can be answered in $O(n + e)$ time by conducting a simple depth-first traversal on the graph, where n is the number of nodes and e is the number of edges in the graph.

There have been several studies on preprocessing a graph in order to answer reachability queries more efficiently [5, 1, 7, 3, 9]. A simple approach would be to construct the transitive closure of the graph, achieving $O(1)$ time query at the cost of $O(n^2)$ time and space for the preprocessing. For graphs with specific properties, there exists methods with smaller complexity bounds. Graph reachability for planar graphs with a single source node and sink node was considered in [5]. Reachability queries for such graphs can be answered in $O(1)$ time given $O(n + e)$ time and $O(n)$ space preprocessing. For partial lattices, a method which achieves $O(1)$ time query with $O(n^2)$ time and $O(n\sqrt{n})$ space preprocessing was shown in [7], where n is the size of the ground set. When considering arbitrary graphs with n vertices and e edges, it has been shown in [3] that for any labeling scheme, there exists a graph such that the reachability labeling has total size of $\Omega(n\sqrt{e})$.

In this paper, we consider the graph reachability query problem on *ST-graph*s, which are DAGs derived from suffix trees and suffix links. ST-graphs are not planar, are not partial lattices. A suffix tree of a given string is a data structure that captures important information concerning the substrings of the string [10]. We present and analyze a version of the interval labeling algorithm of [1] tailored for ST-graphs. It can be shown that for a random string, the ST-graph can be preprocessed in $O(n \log n)$

expected time and space to answer reachability queries in $O(\log n)$ time. Furthermore, we present a series of strings for which their *ST*-graphs require $\Theta(n\sqrt{n})$ time and space of preprocessing when the algorithm is applied. Exhaustive computational calculations indicate that the series gives the worst case instances of the algorithm for the strings of length up to 33, strongly supporting that the worst case complexity of the preprocessing is $\Theta(n\sqrt{n})$ time and space. Assuming this is true, this would break the $O(n^2)$ barrier for total time and space used when conducting $O(n)$ queries.

Reachability on *ST*-graphs solve the problem of whether or not the string represented by the path from the root to the given query nodes are substrings of each other. The algorithm has possible applications in substring pattern set discovery, where the objective is to find best set of substrings that characterizes a given set (or sets) of strings: Consider two substring patterns such that one is a substring of the other. The set of strings which contain the former pattern as a substring is obviously a subset of the set of strings which have the latter pattern as a substring. This property may allow us choose the best pattern set more efficiently, for example, by quickly detecting non-interesting pattern sets.

2 Preliminaries

Let Σ be a finite *alphabet*. An element of Σ^* is called a *string*. Strings x, y, and z are said to be a *prefix*, *substring*, and *suffix* of string $w = xyz$, respectively. The length of a string w is denoted by $|w|$. The empty string is denoted by ε, that is, $|\varepsilon| = 0$. Unless otherwise noted, we shall only consider strings of a fixed alphabet. Also, we assume that all strings end with a unique character $\$ \in \Sigma$ that does not occur anywhere else in the strings.

A *suffix tree* $suftree(s)$ for a given string s is a rooted tree whose edges are labeled with non-empty substrings of s, satisfying the following characteristics. For any node v in the suffix tree, let $path(v)$ denote the string spelled out by concatenating the edge labels on the path from the root to v. For each leaf node v, $path(v)$ is a distinct suffix of s, and for each suffix of s, there exists such a leaf v. Furthermore, each internal node has at least two children, and the first character of the labels on the edges to its children are distinct. The parent of node v is denoted by $parent_s(v)$, and the set of children of node v is denoted by $children_s(v)$. The *length* of node v is defined to be $|path(v)|$. The *depth* of node v with respect to the suffix tree is the number of edges on the path from the root to the node, and is always less than or equal to $|path(v)|$. The *height* of a suffix tree is the maximum depth of all nodes with respect to the suffix tree. Also, let $subtree_s(v)$ be the subtree of the suffix tree rooted at node v.

For a node v where $path(v) = \sigma x$ for some $\sigma \in \Sigma$ and $x \in \Sigma^*$, we denote the *suffix link* of v as $parent_l(v) = u$ where $path(u) = x$. It is easy to see that a unique $parent_l(v)$ exists for each node v in $suftree(s)$, except for the root node. Therefore, the suffix links also form a tree structure, which we denote by $suflinktree(s)$. Let $children_l(v) = \{u : parent_l(u) = v\}$. Note that the depth of node v with respect to the suffix link tree is always equivalent to $|path(v)|$. Also, let $subtree_l(v)$ be the subtree of the suffix link tree rooted at node v.

2.1 *ST*-graphs

Let V be the set of nodes of $suftree(s)$. Let us denote the set of (backward) edges of $suftree(s)$ by $E_s = \{(v, parent_s(v)) : v \in V\}$ and the set of edges of the $suflinktree(s)$

by $E_l = \{(v, parent_l(v)) : v \in V\}$. We define the *ST-graph* of a string s as the directed graph $G = (V, E)$ where $E = E_s \cup E_l$. It is well known that the suffix tree and its suffix links for a string of length n can be constructed and represented in $O(n)$ time and space [10, 6, 8, 4]. Figure 1 shows an example of an *ST*-graph for the string `ababbabbba$`. It is easy to see that the graph is not planar nor a partial lattice.

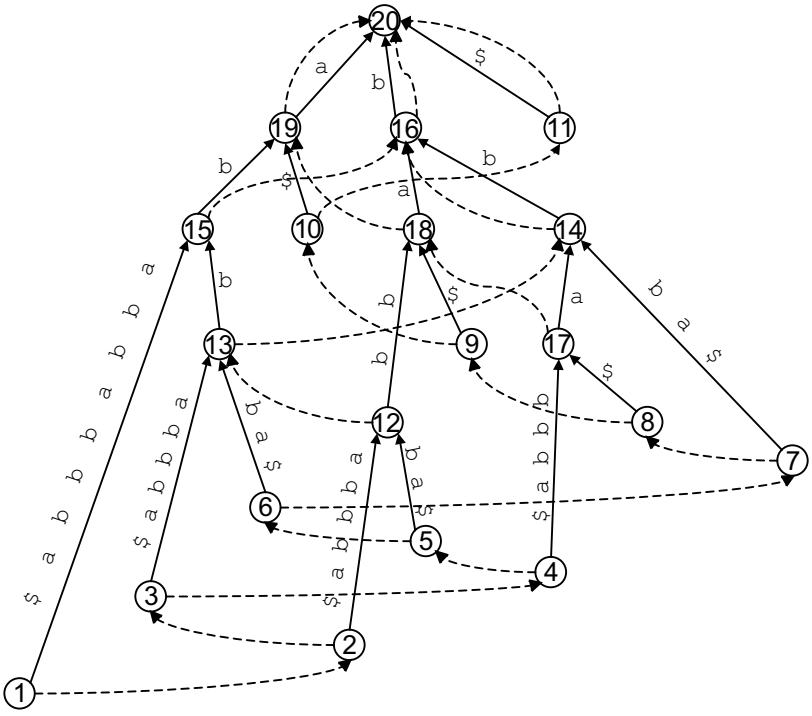

Figure 1. A graph induced from the suffix tree of string `ababbabbba$`. Solid edges represent edges of the suffix tree. The dashed edges represent suffix links.

node	interval	labels	node	interval	labels
1	[1,1]	[1,1]	11	[1,11]	[1,11]
2	[1,2]	[1,2]	12	[12,12]	[1,5],[12,12]
3	[1,3]	[1,3]	13	[12,13]	[1,6],[12,13]
4	[1,4]	[1,4]	14	[12,14]	[1,8],[12,14]
5	[1,5]	[1,5]	15	[15,15]	[1,6],[12,13],[15,15]
6	[1,6]	[1,6]	16	[12,16]	[1,9],[12,16],[17,18]
7	[1,7]	[1,7]	17	[17,17]	[1,8],[17,17]
8	[1,8]	[1,8]	18	[17,18]	[1,9],[12,12],[17,18]
9	[1,9]	[1,9]	19	[17,19]	[1,10],[12,13],[15,15],[17,19]
10	[1,10]	[1,10]	20	[1,20]	[1,20]

Table 1. Post-order interval and labels assigned to *ST*-graph of Fig. 1 by Algorithm 1

The problem we shall consider in this paper is as follows:

Problem 1 (ST-graph reachability query). Given the *ST*-graph $G = (V, E)$ of string s and an arbitrary pair of node $u, v \in V$, $rquery(u, v)$ returns `true` if there exists a path from node u to v in G, and `false` otherwise.

The query $rquery(u, v)$ is equivalent to the query of whether the string $path(v)$ is a substring of $path(u)$ or not.

Lemma 2. *Given an ST-graph $G = (V, E)$ of string s and nodes $u, v \in V$,*

$$rquery(u, v) = \texttt{true} \iff path(v) \text{ is a substring of } path(u)$$

Proof. ($\Rightarrow$) Suppose v is reachable from u. An edge $(p, q) \in E_s$ implies that $path(q)$ is a substring (prefix) of $path(p)$. An edge $(p, r) \in E_l$ implies that $path(r)$ is a substring (suffix) of $path(p)$. Since any path from u to v consists of edges in $E_s \cup E_l$, this implies that v is a substring of u.
($\Leftarrow$) Suppose $path(v)$ is a substring of $path(u)$, i.e. there exists $x, z \in \Sigma^*$ such that $path(u) = xpath(v)z$. If $x = \varepsilon$, then $path(v)$ is a prefix of $path(u)$ which implies $u \in subtree_s(v)$, and that v is reachable from u using edges of E_s. For $x = x_1 \cdots x_k$ ($k \geq 1$), the nodes reachable from u using edges in E_l will have corresponding paths: $x_2 \cdots x_k path(v)z$, $x_3 \cdots x_k path(v)z$, ..., $path(v)z$. Let v' be the node where $path(v') = path(v)z$. Then, since $path(v)$ is a prefix of $path(v')$, v is reachable from v', and therefore reachable from u. □

Corollary 3. *For any two distinct nodes $u, v \in V$ in an ST-graph (V, E), if there exists a path from u to v, then there exists a path: $v_0 \rightarrow v_1 \rightarrow \ldots \rightarrow v_t$ where $t > 0$, $u = v_0$, $v_t = v$, and for some $0 \leq i \leq t$, $(v_{j-1}, v_j) \in E_l$ for all $1 \leq j \leq i$ and $(v_{j-1}, v_j) \in E_s$ for all $i + 1 \leq j \leq t$.*

Proof. Since there exists a path from u to v, $path(v)$ is a substring of $path(u)$ by Lemma 2. The Corollary follows from the argument for the ($\Leftarrow$) part of the proof of Lemma 2. □

3 Interval Labeling Algorithm

In this paper, we analyze the complexity of the interval labeling algorithm for general DAGs presented in [1], when applied to *ST*-graphs. The original algorithm works as follows: First, create a spanning tree of the DAG by examining the nodes in topological order. The parent of each node is chosen so that the number of ancestors of each node is the largest. Then, intervals based on a postorder numbering of the spanning tree are assigned to each node. Further, the intervals of each node are propagated to all its ancestor nodes of the DAG, and as a result, each node will hold a set of interval labels. During the propagation, for a given set of interval labels at each node, *redundant* intervals which are subsumed by larger intervals in the same set are removed.

The interval labeling algorithm of [1] modified for *ST*-graphs is given in Algorithm 1, and Algorithm 2 shows how to answer $rquery(,)$ using the labels. Algorithm 1 has been simplified as follows: First, the topological ordering and spanning tree construction is not necessary. This is because each node v will only have at most two parents, one from the suffix tree, and the other from the suffix link tree. It is easy to verify that the path from the root to v using the edges of the suffix link tree is always at least as long as the path using the edges of the suffix tree. Therefore, the suffix link tree already corresponds to the desired spanning tree, by which intervals are assigned to each node based on a postorder numbering. Second, although the original interval labeling algorithm requires labels to be propagated across all edges, this is not required for *ST*-graphs, and are propagated only across edges in E_s.

The correctness of the algorithm can be proved as follows. Suppose node v is reachable from node u. Then, node v must hold an interval label which subsumes the interval of u for Algorithm 2 to correctly answer $rquery(u, v)$. From lines 5-8

in Algorithm 1, node v holds all intervals of nodes in $subtree_s(v)$, which includes the interval for node v_i that can be reached by traversing suffix links starting from u as in Corollary 3. Since the interval of v_i subsumes the interval of u defined by the postorder numbering assigned in lines 1-4, v will contain an interval label which subsumes the interval of u.

4 Complexity

In this section, we will derive estimates on the complexity of Algorithm 1. In particular, we will consider the expected running time, as well as lower bounds for the worst case.

Lemma 4. *Assuming a constant size alphabet, Algorithm 1 runs in time linear in the total number of labels assigned to the nodes of the ST-graph.*

Proof. The maximum in-degree of each node is bounded by $O(|\Sigma|)$. This is because for the suffix tree, all labels on the edges must begin with a different character of the alphabet. For the suffix link tree, $|\{v : path(v) = \sigma path(u), \sigma \in \Sigma\}| \leq |\Sigma|$ for any node u. Therefore, assuming that $|\Sigma|$ is constant, merging the sorted labels (and removing subsumed intervals) from the in-coming nodes can be done in time linear in the total size of the in-coming labels. □

From Lemma 4, we have only to count the number of labels that will be assigned to the ST-graph by Algorithm 1 in order to estimate the complexity of the algorithm.

4.1 Expected running time

A simple bound relating the height of the suffix tree and the total number of labels assigned to the ST-graph is shown in the following lemma.

Lemma 5. *For an ST-graph for a string of length n whose suffix tree has height h, the total number of interval labels assigned by Algorithm 1 is at most $O(nh)$.*

Proof. Notice that since the interval labels are only propagated through edges of the suffix tree, the maximum number of labels at a given node v is bounded by the number of nodes in $subtree_s(v)$. Therefore, at a given depth of the suffix tree, there can only be a maximum total of $O(n)$ labels, i.e., the total number of nodes in the suffix tree, since the subtrees of nodes of the same depth cannot intersect. This results in a maximum total of $O(nh)$ labels for all nodes. □

Theorem 6. *The expected running time of Algorithm 1 for a random string of length n is $O(n \log n)$.*

Proof. It is known that the expected height of the suffix tree of a random string of length n is $O(\log n)$ [2]. The theorem follows from Lemma 4 and Lemma 5. □

4.2 Worst Case Lower Bounds

In this subsection, we will give a lower bound for the worst case complexity of Algorithm 1. We will present a series of strings of length n whose ST-graphs will have $\Theta(n\sqrt{n})$ labels assigned by the algorithm. Prior to this, we show related properties of suffix trees and suffix link trees.

Algorithm 1: Assign labels to each node of the *ST*-graph.

Input: *ST*-graph G(V,E) of string s
Output: Labeled *ST*-graph (v.int and v.labels for all v ∈ V)
1 **foreach** *node* v ∈ V *in post-order of suflinktree*(s) **do**
2 | v.int := [**min**{post-order number of $subtree_l$(v)}, post-order number of v];
3 | v.labels := {v.int};
4 **endfch**
5 **foreach** *node* v ∈ V *in post-order of suftree*(s) **do**
6 | v.labels := merge and sort v.labels and {c.labels : c ∈ $children_s$(v)};
7 | Remove $[i, j] \in$ v.labels if $\exists [i', j'] \in$ v.labels s.t. $i' \leq i \leq j \leq j'$;
8 **endfch**

Algorithm 2: *rquery*(u, v) on *ST*-graphs using labels assigned by Algorithm 1.

Input: Labeled *ST*-graph G(V,E) of string s and nodes u, v ∈ V
Output: *rquery*(u, v)
1 $[i, j]$ = u.int;
2 **if** $\exists [i', j'] \in$ v.label *such that* $i' \leq i \leq j \leq j'$ **then return** `true`;
3 **return** `false`;

Properties of *ST*-graphs. For $|\Sigma| = 2$, there can only be one string of a given length n, and the structure of the *ST*-graph is determined uniquely (recall that all strings terminate with a uniquely occurring character \$ $\in \Sigma$). It is not difficult to show that the total number of labels is $3(n-1) = O(n)$ in such case. In what follows we therefore consider the case for $|\Sigma| \geq 3$.

Lemma 7. *The number of interval labels assigned to each node of the ST-graph for any string s is bounded by the number of leaves in suflinktree*(s)*. More generally, the exact number of labels assigned to node v is* $\boldsymbol{min}_{W \subseteq subtree_s(v)}\{|W| : subtree_s(v) \subseteq \cup_{w \in W} subtree_l(w)\}$.

Proof. Let ℓ be the number of leaves in *suflinktree*(s). In the post-order traversal on *suflinktree*(s) (lines 1-4 of Algorithm 1), each node receives exactly one interval label, and there are exactly ℓ different values for the first element of the intervals. In post-order traversal on *suftree*(s) (lines 5-8 of Algorithm 1), we remove subsumed intervals and therefore each node gets at most ℓ interval labels. The exact number of labels follows from a similar argument. □

Lemma 8. *If and only if* $|children_s(v)| = 2$ *for any internal node* $v \neq$ *root of suftree*(s)*, the number of internal nodes (excluding the root) is the maximum, which is* $n - 3$.

Proof. Because of \$ there are always n leaves in *suftree*(s). Since there is always an edge labeled with \$ from the root, $|children_s(root)| = |\Sigma|$. □

Lemma 9. *Assume suftree*(s) *satisfies the condition in Lemma 8. For the three following groups of the internal nodes of suftree*(s)*,*

n_1*: internal nodes with two leaf-children;*
n_2*: internal nodes with one leaf-child and one internal-child;*
n_3*: internal nodes with two internal-children;*

where $n_1 + n_2 + n_3 = n - 3$*, we have*

$$n_2 = n - 2n_1 - 1, \text{ and}$$
$$n_3 = n_1 - 2.$$

Proof. Because of $, *suftree*($s$) always has n leaves. Since there are n_1 internal nodes with two leaf-children, and since the leaf corresponding to suffix $ is a child of the root, we have $n_2 = n - 2n_1 - 1$. Finally $n_3 = n - 3 - n_1 - n_2 = n_1 - 2$. □

Lemma 10. *Assume suftree*(s) *satisfies the condition in Lemma 8. For any node* v*, if* $|children_l(v)| \geq 2$*, then* v *is a group* n_3 *node in suftree*(s)*.*

Proof. Let $x = path(v)$, and let $u, w \in children_l(v)$. Since u and w are nodes in the suffix tree, there exists at least four possible suffixes whose paths must pass node v. We denote these paths as $x\sigma_1 y_1, x\sigma_2 y_2, x\sigma_3 y_2, x\sigma_4 y_4$ where $\sigma_i \in \Sigma$ and $y_i \in \Sigma^*$ for $1 \leq i \leq 4$. It must be that $\sigma_{i_1} = \sigma_{i_2}$ and $\sigma_{i_3} = \sigma_{i_4}$ for some $\{i_1, i_2, i_3, i_4\} = \{1, 2, 3, 4\}$. Since all four paths must be distinct, it follows that there must exist distinct child nodes of v where $x\sigma_{i_1} y_{i_1}$ and $x\sigma_{i_2} y_{i_2}$ diverge, and $x\sigma_{i_3} y_{i_3}$ and $x\sigma_{i_4} y_{i_4}$ diverge respectively. □

Lemma 11. *Assume suftree*(s) *satisfies the condition in Lemma 8. The number* ℓ *of leaves in suflinktree*(s) *is at most* $n_1 + 1$*.*

Proof. Let ℓ be the number of leaves in *suflinktree*(s). Since all leaves of *suftree*(s) are included in one path of suffix links and this path leads to the root, the maximum number of internal nodes v of *suflinktree*(s), such that $|children_l(v)| \geq 2$, is $\ell - 3$ excepting the root. From Lemma 10, this leads to $\ell - 3$ internal nodes with two internal-children in *suftree*(s). Therefore, $\ell \leq n_3 + 3 = n_1 + 1$ by Lemma 9. □

Lemma 12. *Assume suftree*(s) *satisfies the condition in Lemma 8. If* $\ell = n_3 + 3$*, then the longest internal node in group* n_3 *has at most* $\ell - 1$ *labels.*

Proof. It follows from Lemmas 9, 10, and 11. □

Lower Bound. Consider the following series of strings of length $n = \frac{3}{2}i(i+3) + 5$ where $i = 1, 2, 3, \ldots$:

$$\mathtt{X_i} = \mathtt{ab^iab^{i+1}ab^iab^1ab^iab^{i-1}ab^iab^2ab^iab^{i-2}a \cdots ab^kab^iab^{i-k}a \cdots ab^iab^{\lceil i/2 \rceil}ab^iab^ia\$}$$

In what follows, we analyze $\mathtt{X_i}$ in terms of internal nodes of *suftree*($\mathtt{X_i}$), shown in Figure 2. We consider the structure of *suftree*($\mathtt{X_i}$) in Lemmas 14, 15, 16, 18, and the structure of *suflinktree*($\mathtt{X_i}$) in Lemmas 19, 20, 21, 22. An important point in the lemmas is that if a substring p appears only once in the string, then there is no explicit internal node v such that $path(v) = p$, and the substring will correspond to a position on an edge leading to a leaf node (or the leaf node itself). Also, if $p\sigma_1$ and $p\sigma_2$ are both substrings of the string for distinct $\sigma_1, \sigma_2 \in \Sigma$, then there exists a node v such that $path(v) = p$.

Lemma 13. *suftree*($\mathtt{X_i}$) *satisfies the condition in Lemma 8.*

Proof. Any occurrence of a in $\mathtt{X_i}$ is followed by either b or $, and any occurrence of b in $\mathtt{X_i}$ is followed by either a or b. Moreover, $ appears only at the end of $\mathtt{X_i}$. Thus, for any internal node v of *suftree*($\mathtt{X_i}$), we have $|children_s(v)| = 2$. □

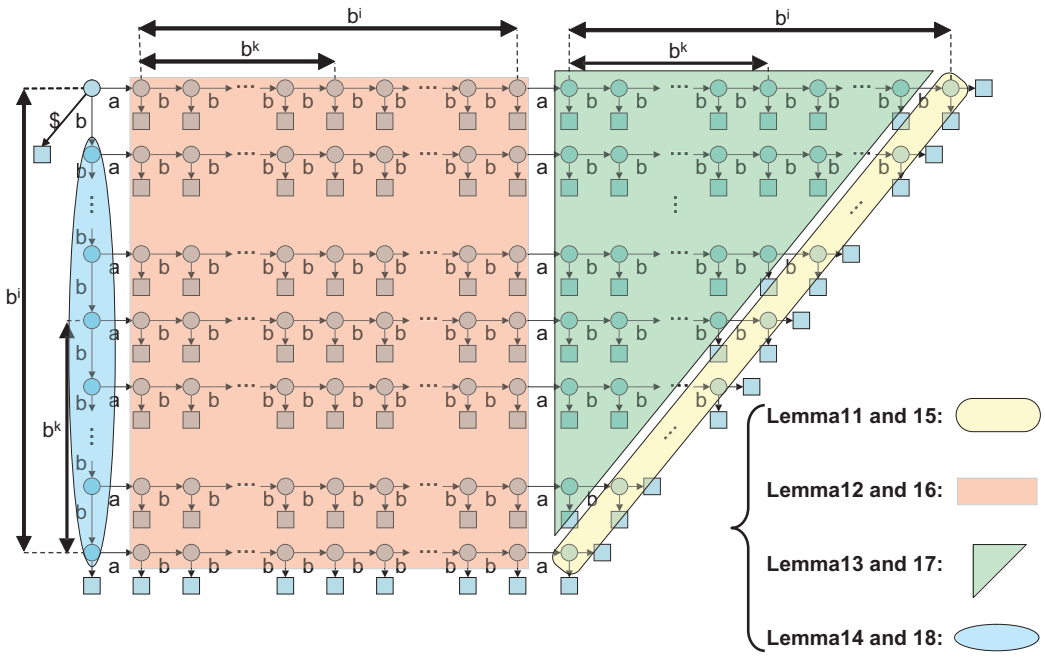

Figure 2. Illustration for *suftree*($\mathtt{X}_i$). The four groups of the internal nodes are dealt in Lemmas 14, 15, 16, and 18, respectively.

Lemma 14. *For any k ($0 \leq k \leq i$), there exists an internal node corresponding to* $\mathtt{b}^{i-k}\mathtt{ab}^{i}\mathtt{ab}^{k}$*, and this node belongs to group n_1 of suftree*($\mathtt{X}_i$).

Proof. We have three cases to consider:

- When $k = 0$.
  Since there are two strings $\mathtt{b}^{i}\mathtt{ab}^{i}\mathtt{ab}$ and $\mathtt{b}^{i}\mathtt{ab}^{i}\mathtt{a\$}$ in $\mathtt{X}_i$, there is an internal node for $\mathtt{b}^{i}\mathtt{ab}^{i}\mathtt{a}$. Since there is no other occurrence of $\mathtt{b}^{i}\mathtt{ab}^{i}\mathtt{a}$, the two children of this node are both a leaf node and thus it belongs to group n_1.
- When $0 < k < i$.
  Consider a substring $\mathtt{Y}_i$ of $\mathtt{X}_i$ such that

$$\mathtt{Y}_i = \mathtt{ab}^{k-1}\mathtt{ab}^{i}\mathtt{a}\underline{\mathtt{b}^{i-k+1}\mathtt{ab}^{i}\mathtt{ab}^{k}}\mathtt{ab}^{i}\mathtt{a}\underline{\mathtt{b}^{i-k}\mathtt{ab}^{i}\mathtt{ab}^{k+1}}\mathtt{ab}^{i}\mathtt{a}.$$

  Then, $\mathtt{b}^{i-k}\mathtt{ab}^{i}\mathtt{ab}^{k}$ appears twice in this substring, as underlined.
  Now we show that each of $\mathtt{b}^{i-k}\mathtt{ab}^{i}\mathtt{ab}^{k}\mathtt{a}$ and $\mathtt{b}^{i-k}\mathtt{ab}^{i}\mathtt{ab}^{k}\mathtt{b}$ appears only once in $\mathtt{X}_i$. For $\mathtt{b}^{i-k}\mathtt{ab}^{i}\mathtt{ab}^{k}\mathtt{a}$, it is clear because $\mathtt{ab}^{k}\mathtt{a}$ appears exactly once in $\mathtt{X}_i$. String $\mathtt{b}^{i-k}\mathtt{ab}^{i}\mathtt{ab}^{k}\mathtt{b} = \mathtt{b}^{i-k}\mathtt{ab}^{i}\mathtt{ab}^{k+1}$ appears in $\mathtt{Y}_i$ (the second underlined part). This is the only occurrence of $\mathtt{b}^{i-k}\mathtt{ab}^{i}\mathtt{ab}^{k+1}$ in $\mathtt{X}_i$, because the prefix $\mathtt{b}^{i-k}\mathtt{a}$ would appear in substrings $\mathtt{ab}^{i-k+x}\mathtt{ab}^{i}\mathtt{ab}^{k-x}\mathtt{a}$ with $x \geq 0$, but then the suffix $\mathtt{b}^{i}\mathtt{ab}^{k+1}$ cannot match.
- When $k = i$.
  By similar discussions to the case that $k = 0$. □

Lemma 15. *For any x ($0 \leq x \leq i$) and y ($0 \leq y \leq i$), there exists an internal node corresponding to* $\mathtt{b}^{x}\mathtt{ab}^{y}$*, and this node belongs to group n_2.*

Proof. We have three cases to consider:

- When $y = 0$.
  Since there are more than one occurrences of $\mathtt{b}^{\mathtt{i}}\mathtt{ab}$, there are more than one occurrences of $\mathtt{b}^{\mathtt{x}}\mathtt{ab}$. In addition, since there is exactly one occurrence of $\mathtt{b}^{\mathtt{i}}\mathtt{a\$}$, there is exactly one occurrence of $\mathtt{b}^{\mathtt{x}}\mathtt{a\$}$.
- When $0 < y < i$.
  Since there are more than one occurrences of $\mathtt{b}^{\mathtt{x}}\mathtt{ab}^{\mathtt{i}}$, there are more than one occurrences of $\mathtt{b}^{\mathtt{x}}\mathtt{ab}^{\mathtt{y}+1}$. In addition, since there is exactly one occurrence of $\mathtt{b}^{\mathtt{i}}\mathtt{ab}^{\mathtt{y}}\mathtt{a}$ for each $0 < y < i$, $\mathtt{b}^{\mathtt{x}}\mathtt{ab}^{\mathtt{y}}\mathtt{a}$ appears exactly once.
- When $y = i$.
  There is exactly one occurrence of $\mathtt{b}^{\mathtt{x}}\mathtt{ab}^{\mathtt{i}}\mathtt{a}$ for each $0 \leq x \leq i$. Since there is exactly one occurrence of $\mathtt{b}^{\mathtt{i}}\mathtt{ab}^{\mathtt{i}+1}$, $\mathtt{b}^{\mathtt{x}}\mathtt{ab}^{\mathtt{i}+1}$ appears exactly once for each $0 \leq x \leq i$.

□

Lemma 16. *For any y ($0 \leq y < k$), where $1 \leq k \leq i$, there exists an internal node corresponding to $\mathtt{b}^{\mathtt{i}-\mathtt{k}}\mathtt{ab}^{\mathtt{i}}\mathtt{ab}^{\mathtt{y}}$, and this node belongs to group n_2.*

Proof. By similar arguments to Lemmas 14 and 15. □

The next corollary follows Lemmas 15 and 16.

Corollary 17. *For any k ($0 \leq k \leq i$), let $path(u) = \mathtt{b}^{\mathtt{i}-\mathtt{k}}\mathtt{a}$ and $path(v) = \mathtt{b}^{\mathtt{i}-\mathtt{k}}\mathtt{ab}^{\mathtt{i}}\mathtt{ab}^{\mathtt{k}}$. Then, the path from u to v contains $i + k + 2$ internal nodes (including u and v).*

Lemma 18. *For any x ($0 < x \leq i$), there is an internal node corresponding to $\mathtt{b}^{\mathtt{x}}$. The node for $\mathtt{b}^{\mathtt{i}}$ belongs to group n_2, and the other nodes for $\mathtt{b}^{\mathtt{x}}$ with $0 < x < i$ belong to group n_3.*

Proof. Since $\mathtt{b}^{\mathtt{i}}\mathtt{a}$ and $\mathtt{b}^{\mathtt{i}+1}$ appear in $\mathtt{X}_{\mathtt{i}}$, there is an internal node for $\mathtt{b}^{\mathtt{x}}$ for any $0 < x \leq i$. Since $\mathtt{b}^{\mathtt{i}}\mathtt{a}$ appears more than once and $\mathtt{b}^{\mathtt{i}+1}$ appears exactly once, the node for $\mathtt{b}^{\mathtt{i}}$ belongs to group n_2. All the nodes for $\mathtt{b}^{\mathtt{x}}$ with $0 < x < i$ belong to group n_3, since both $\mathtt{b}^{\mathtt{x}}\mathtt{a}$ and $\mathtt{b}^{\mathtt{x}+1}$ appear more than once. □

From here on, we consider the suffix links of $suftree(\mathtt{X}_{\mathtt{i}})$.

Lemma 19. *For any k ($0 \leq k \leq i$), let v be the internal node such that $path(v) = \mathtt{b}^{\mathtt{i}-\mathtt{k}}\mathtt{ab}^{\mathtt{i}}\mathtt{ab}^{\mathtt{k}}$. Then we have $|children_l(v)| = 0$.*

Proof. For contrary, assume that $|children_l(v)| \geq 1$. Then there must exist a node corresponding to either $\mathtt{ab}^{\mathtt{i}-\mathtt{k}}\mathtt{ab}^{\mathtt{i}}\mathtt{ab}^{\mathtt{k}}$ or $\mathtt{b}^{\mathtt{i}-\mathtt{k}+1}\mathtt{ab}^{\mathtt{i}}\mathtt{ab}^{\mathtt{k}}$. However, we show that these strings can appear at most once in $\mathtt{X}_{\mathtt{i}}$.

First, we consider $\mathtt{ab}^{\mathtt{i}-\mathtt{k}}\mathtt{ab}^{\mathtt{i}}\mathtt{ab}^{\mathtt{k}}$:

- When $k = 0$. $\mathtt{ab}^{\mathtt{i}}\mathtt{ab}^{\mathtt{i}}\mathtt{a}$ appears only once (at the end of $\mathtt{X}_{\mathtt{i}}$).
- When $0 < k < i$. Prefix $\mathtt{ab}^{\mathtt{i}-\mathtt{k}}\mathtt{a}$ appears only once in $\mathtt{X}_{\mathtt{i}}$.
- When $k = i$. Prefix $\mathtt{aa}$ never appears in $\mathtt{X}_{\mathtt{i}}$.

Now let us consider $\mathtt{b}^{\mathtt{i}-\mathtt{k}+1}\mathtt{ab}^{\mathtt{i}}\mathtt{ab}^{\mathtt{k}}$. By Lemma 14, there is an internal node corresponding to $\mathtt{b}^{\mathtt{i}-\mathtt{k}+1}\mathtt{ab}^{\mathtt{i}}\mathtt{ab}^{\mathtt{k}-1}$ and this node belongs to group n_1. This implies that $\mathtt{b}^{\mathtt{i}-\mathtt{k}+1}\mathtt{ab}^{\mathtt{i}}\mathtt{ab}^{\mathtt{k}}$ can appear at most once in $\mathtt{X}_{\mathtt{i}}$. □

The above lemma implies that the internal nodes v such that $path(v) = \mathtt{b}^{\mathtt{i}-\mathtt{k}}\mathtt{ab}^{\mathtt{i}}\mathtt{ab}^{\mathtt{k}}$ are the leaves of $suflinktree(\mathtt{X}_{\mathtt{i}})$. See also the upper diagram in Figure 3.

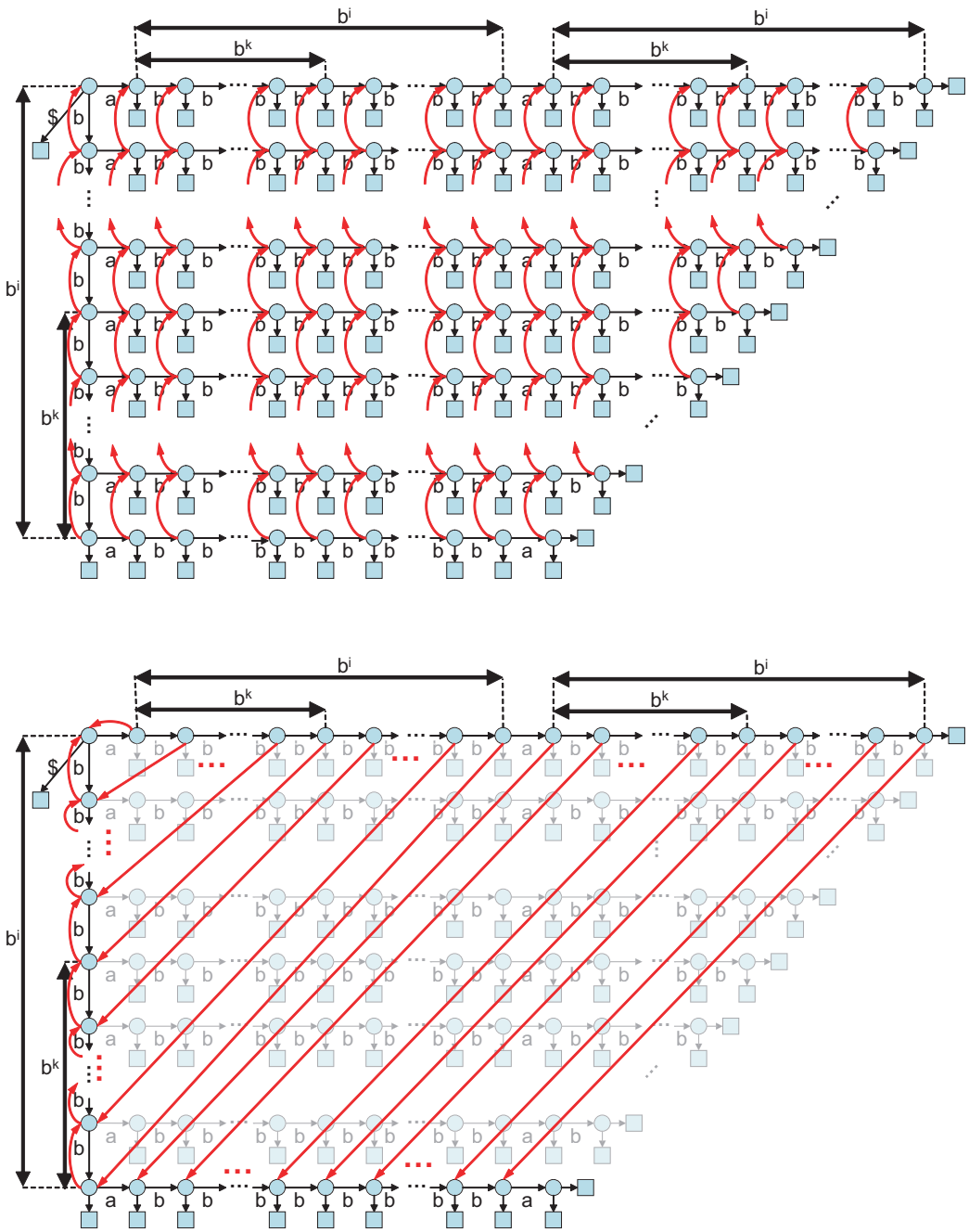

Figure 3. Illustration for the suffix links of *suftree*($\mathtt{X}_i$). For the sake of visibility, the suffix links of $\mathtt{X}_i$ are shown in two rounds. Moreover, for simplicity, the suffix links for the leaves are omitted here.

Lemma 20. *For any internal node $v_{x,y}$ such that $path(v_{x,y}) = \mathtt{b}^x\mathtt{ab}^y$ ($0 \le x \le i$ and $0 \le y \le i$), we have $|children_l(v_{x,y})| = 1$.*

Proof. We have three cases to consider:

- When $x = 0$.
  There is no occurrence of $\mathtt{aab}^y$, and we have two distinct occurrences of $\mathtt{b}^i\mathtt{ab}^i$ in $\mathtt{X}_i$. Thus we have $|children_l(v_{0,y})| = 1$.
- When $0 < x < i$.
  $\mathtt{ab}^x\mathtt{ab}^y$ appears only once, since there is only one occurrence of $\mathtt{ab}^x\mathtt{ab}^i$. Because

$\mathtt{b}^{\mathtt{i}}\mathtt{ab}^{\mathtt{y}}$ appears more than once, there are more than one occurrence of $\mathtt{b}^{\mathtt{x}}\mathtt{ab}^{\mathtt{y}}$. Thus we have $|children_l(v_{x,y})| = 1$.

– When $x = i$.
We have at least two occurrences of $\mathtt{ab}^{\mathtt{i}}\mathtt{ab}^{\mathtt{y}}$. Since $\mathtt{b}^{\mathtt{i+1}}$ appears only once, there is only one occurrence of $\mathtt{b}^{\mathtt{i+1}}\mathtt{ab}^{\mathtt{y}}$. Thus we have $|children_l(v_{i,y})| = 1$.

□

Lemma 21. *For any internal node v_z such that $path(v_z) = \mathtt{b}^{\mathtt{i-k}}\mathtt{ab}^{\mathtt{i}}\mathtt{ab}^{\mathtt{z}}$ $(0 \le z < k)$, where $1 \le k \le i$, we have $|children_l(v_z)| = 1$.*

Proof. By similar arguments to Lemma 20. □

Lemma 22. *For any internal node v_x such that $path(v_x) = \mathtt{b}^{\mathtt{x}}$ $(0 < x < i)$, we have $|children_l(v_x)| = 2$. In addition, for the node v_i such that $path(v_i) = \mathtt{b}^{\mathtt{i}}$, we have $|children_l(v_i)| = 1$.*

Proof. Due to Lemma 15, there exist internal nodes u_x such that $path(u_x) = \mathtt{ab}^{\mathtt{x}}$ for each $0 < x \le i$, and therefore have $(u_x, v_x) \in E_l$. Due to Lemma 18, we have $(v_{x+1}, v_x) \in E_l$ for $0 < x < i$. For $x = i$, however, we have $(v_{i+1}, v_i) \notin E_l$ because $\mathtt{b}^{\mathtt{i+1}}$ does not correspond to a node (since it occurs only once in the string). □

From the above lemmas, only the nodes corresponding to $\mathtt{b}^{\mathtt{x}}$ $(0 < x < i)$ are of in-degree two in *suflinktree*$(\mathtt{X_i})$. Plus, only the root node is of in-degree three in *suflinktree*$(\mathtt{X_i})$. See also Figure 3 for these observations.

The number of nodes covered in these lemmas are as follows:

Lemma	11 or 15	12 or 16	13 or 17	14 or 18
The number of nodes	$i+1$	$(i+1)^2$	$i(i+1)/2$	i

the sum of these nodes is $\frac{3}{2}i^2 + \frac{9}{2}i + 2 = n - 3$, which indicates that we have discussed all nodes of the ST-graph for the string $\mathtt{X_i}$.

Now we are finally ready to show the lower bound on the number of interval labels.

Theorem 23. *The number of labels assigned to a suffix tree by Algorithm 1 is $\Omega(n\sqrt{n})$.*

Proof. From Lemmas 7, 12, 13, 14, 15, 16, 18, 19, 20, 21, 22, Corollary 17, we have $\ell = i + 2$ and achieve the following equations on the total number of interval labels that are assigned to *suftree*$(\mathtt{X_i})$:

$$
\begin{aligned}
&(2 + 3 + \cdots + \ell - 1) \times (i + 1) + \ell \times \Big\{1 + \sum_{k=0}^{i} \{(i + k + 2) - (\ell - 2)\}\Big\} \\
&+\{(\ell - 1) + (\ell - 2) + \cdots 3\} + 1 \times (n + 1) \\
&= \{2 + 3 + \cdots + (i + 1)\} \times (i + 1) + (i + 2) \times \{1 + \sum_{k=0}^{i} (k + 2)\} \\
&+\{(i + 1) + i + \cdots 3\} + \frac{3i(i + 3)}{2} + 6 \\
&= \frac{i(i + 3)(i + 1)}{2} + \frac{(i + 2)\{2 + i(i + 1) + 4(i + 1)\}}{2} + \frac{(i - 1)(i + 4)}{2} + \frac{3i(i + 3)}{2} + 6 \\
&= \frac{2i^3 + 15i^2 + 31i + 20}{2}. \qquad (1)
\end{aligned}
$$

Since $n = \frac{3}{2}i(i+3)+5$ and $i \geq 1$, we have

$$i = \frac{\sqrt{24n-39}}{6} - \frac{3}{2}.$$

By substituting this for the i's in Equation (1), the total number of interval labels assigned to *suftree*($\mathtt{X_i}$) is shown to be $\Theta(n\sqrt{n})$. □

The worst case upper bound for Algorithm 2 is $O(\log n)$: From the argument in Lemma 4, the labels at each node can be stored as sorted arrays. Also, the maximum number of labels at each node is bounded by $O(n)$ (Lemma 7). Therefore, line 2 in Algorithm 2 can be run in $O(\log n)$ time using a standard binary search on the label array.

5 Computational Experiments

We exhaustively enumerated all strings of length $n \leq 33$ consisting of $\{\mathtt{a}, \mathtt{b}\}$ and ending with $, and applied Algorithm 1 to each string. For each n, the number of labels in the worst case was recorded. The results are shown in Table 2.

n	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
#labels	3	6	9	12	15	18	22	26	30	34	39	44	49	54	59	64
n	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33
#labels	69	74	79	85	91	97	103	109	115	121	127	133	139	145	151	158

Table 2. The maximum number of labels that is assigned by Algorithm 1 to any string of length n.

We note that for $n \leq 7$, the worst case corresponds to the string $\mathtt{a}^{\mathtt{n}-1}\$$, and therefore the total number of labels is $3(n-1)$. For $n = 11, 20, 32$, the worst case instances contain $\mathtt{X}_1$, $\mathtt{X}_2$, $\mathtt{X}_3$, and the total number of labels is as calculated in Theorem 23. Generally for $7 \leq n \leq 33$, we found that the total number of labels in the worst case can be written exactly with the following formula:

$$\mathbf{max}\{f(\lfloor\sqrt{(2n-3)/3}\rfloor), f(\lceil\sqrt{(2n-3)/3}\rceil)\}$$

where $f(k) = (k+2)n - k(k^2+3)/2 - 3$. We have also confirmed for smaller n and with larger sized alphabets, the worst case instances will only contain one type of character (excluding $) for $n \leq 7$, and two types of characters (excluding $) for $n \geq 7$, therefore corresponding to the instances given above. (At $n = 7$, both types had equal worst case label size of 18.) This seems natural since Lemma 8 indicates that the more types of characters used, the less number of nodes there are in the *ST*-graph.

Although we have not been able to give a rigorous proof for an upperbound of $O(n\sqrt{n})$, the above results strongly suggest this bound.

6 Discussion

We presented an algorithm that can processes an *ST*-graph for a string of length n, so that reachability queries between arbitrary pairs of nodes can be answered in $O(\log n)$ time. The expected time and space complexity of the preprocessing algorithm for a

random string is $O(n \log n)$. We also presented a series of strings for which the algorithm requires $\Theta(n\sqrt{n})$ time and space for preprocessing. Exhaustive computational search for $n \leq 33$ showed that the strings of the series also achieve the worst case instances of the algorithm. Although we have not been able to give a direct proof, this provides strong evidence that the worst case time complexity of the algorithm is also $\Theta(n\sqrt{n})$.

Since a suffix tree can have a height of $O(n)$, a naïve consideration of Lemma 5 only gives an $O(n^2)$ bound for the number of labels for an ST-graph of a suffix tree of height $O(n)$, rather than $O(n\sqrt{n})$. There seems to be a delicate tradeoff between deep paths in the suffix tree and deep paths in the suffix link tree. For example, the suffix tree for string $\mathtt{a}^{\mathtt{n}}\$$ will have a path of depth $O(n)$. However, the number of total labels in this case is also limited to $O(n)$, since there are only two leaves in the suffix link tree, which bounds the number of labels for each node to 2 (Lemma 7). There also exists strings $\mathtt{a}^{\mathtt{k}}\underbrace{\mathtt{bbccddeeff...}}_{\mathtt{k}\text{ character types}}\$$ $(n = 3k+1)$, where their suffix trees have a path of depth $O(n)$, and their suffix link trees have $O(n)$ leaves. However, the total number of labels in this case is also $O(n)$, since all but two of the leaves in the suffix link tree are very shallow.

6.1 Open Problems

There are several open problems that are of interest concerning reachability queries on ST-graphs.

1. Whether or not there exists an algorithm which can do better: $O(n\sqrt{n})$ preprocessing and $O(1)$ query, or ultimately $O(n)$ preprocessing and $O(1)$ query.
2. Whether or not we can simulate reachability queries on suffix *trie* graphs. All nodes in the suffix tree correspond to a substring of the original string. However, there can exist substrings in the string without a corresponding explicit node. An implicit node of a suffix tree is a position in the suffix tree which ends somewhere in the middle of an edge. So far we have considered reachability queries between explicit nodes of the ST-graph. Although reachability between implicit nodes of the ST-graph is straightforward with respect to the ST-graph, the result does not correspond to the substring relation between implicit nodes, as it does for explicit nodes shown in Lemma 2.

Acknowledgments

This work was supported in part by The Ministry of Education, Culture, Sports, Science and Technology, Grant-in-Aid for Young Scientists (B).

References

[1] R. Agrawal, A. Borgida, and H. V. Jagadish: *Efficient management of transitive relationships in large data and knowledge bases*, in Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data, 1989, pp. 253–262.

[2] A. Apostolico and W. Szpankowski: *Self-alignments in words and their applications.* Journal of Algorithms, 13(3) 1992, pp. 446–467.

[3] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick: *Reachability and distance queries via 2-hop labels*, in Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms, 2002, pp. 937–946.

[4] D. Gusfield: *Algorithms on Strings, Trees, and Sequences*, Cambridge University Press, 1997.

[5] T. Kameda: *On the vector representation of the reachability in planar directed graphs.* Information Processing Letters, 3(3) 1975, pp. 75–77.

[6] E. M. McCreight: *A space-economical suffix tree construction algorithm.* J. ACM, 23(2) 1976, pp. 262–272.

[7] M. Talamo and P. Vocca: *A data structure for lattice representation.* Theoretical Computer Science, 175(2) 1997, pp. 373–392.

[8] E. Ukkonen: *On-line construction of suffix trees.* Algorithmica, 14(3) 1995, pp. 249–260.

[9] H. Wang, H. He, J. Yang, P. S. Yu, and J. X. Yu: *Dual labeling: answering graph reachability queries in constant time*, in Proc. 22nd International Conference on Data Engineering, 2006, to appear.

[10] P. Weiner: *Linear pattern matching algorithms*, in Proc. 14th IEEE Annual Symp. on Switching and Automata Theory, 1973, pp. 1–11.