The Position Heap of a Trie

Yuto Nakashima[1], Tomohiro I[1,2], Shunsuke Inenaga[1], Hideo Bannai[1], and Masayuki Takeda[1]

[1] Department of Informatics, Kyushu University, Japan
{yuto.nakashima,tomohiro.i,inenaga,bannai,takeda}@inf.kyushu-u.ac.jp
[2] Japan Society for the Promotion of Science (JSPS)

Abstract. The position heap is a text indexing structure for a single text string, recently proposed by Ehrenfeucht et al. [*Position heaps: A simple and dynamic text indexing data structure*, Journal of Discrete Algorithms, 9(1):100-121, 2011]. In this paper we introduce the position heap for a set of strings, and propose an efficient algorithm to construct the position heap for a set of strings which is given as a trie. For a fixed alphabet our algorithm runs in time linear in the size of the trie. We also show that the position heap can be efficiently updated after addition/removal of a leaf of the input trie.

1 Introduction

Classical text indexing structures such as suffix trees [17], suffix arrays [14], directed acyclic word graphs [6], and compact directed acyclic word graphs [5], allow us to find occurrences of a given pattern string in a text efficiently. Linear-time construction algorithms for these structures exist (e.g. [16,11,6,10]).

Very recently, a new, alternative text indexing structure called *position heaps* have been proposed [9]. Like the above classical indexing structures, the position heap of a text t allows us to find the occurrences of a given pattern p in t in $O(m+r)$ time, where m is the length of p and r is the number of occurrences of p in t. A linear-time algorithm to construct position heaps is also presented in [9], which is based on Weiner's suffix tree construction algorithm [17]. An on-line linear-time algorithm for constructing position heaps is proposed in [13], which is based on Ukkonen's on-line suffix tree construction algorithm [16].

In this paper, we extend the position heap data structure to the case where the input is a set W of strings. The position heap of W is denoted by $PH(W)$. We assume that the input set W of strings is represented as a trie. Since the trie is a compact representation of W, it is challenging to construct $PH(W)$ in time only proportional to the size of the trie, rather than to the total length of the strings in W. If n is the size of the input trie, then we propose an $O(n)$-time algorithm to construct $PH(W)$ assuming that the alphabet is fixed. We also show that we can augment $PH(W)$ in $O(n)$ time and space so that the occurrences of a given pattern string in the input trie can be computed in $O(m + r)$ time, where m is the pattern length and r is the number of occurrences to report.

L. Calderón-Benavides et al. (Eds.): SPIRE 2012, LNCS 7608, pp. 360–371, 2012.

A distinction between position heaps and the other classical indexing structures is that position heaps allow us efficient edit operations on arbitrary positions of the input text [9]. In this paper, we show that it is possible to update in $O(h \log n)$ time the position heap for a set of strings after addition/removal of a leaf of the input trie, where h is the height of the position heap. Although h can be as large as $O(n)$, the significance of our algorithm is that when $h = o(n/\log n)$ the position heap can be updated in $o(n)$ time, while a naïve approach of constructing the position heap for the edited trie from scratch requires $\Theta(n)$ time.

Related Work. Computing suffix trees for a set of strings represented as a trie was first considered by Kosaraju [12], and he introduced an $O(n \log n)$-time construction algorithm. Later, an improved algorithm that works in $O(n)$ time for a fixed alphabet was proposed by Breslauer [7]. An $O(n)$-time construction algorithm for integer alphabets is also known [15]. Our algorithm to construct position heap for a trie is based on the algorithms of [9] and [7].

2 Preliminaries

2.1 Notations on Strings

Let Σ be an *alphabet*. Throughout the paper we assume that Σ is fixed. An element of Σ^* is called a *string*. The length of a string w is denoted by $|w|$. The empty string ε is a string of length 0, namely, $|\varepsilon| = 0$. For a string $w = xyz$, x, y and z are called a *prefix*, *substring*, and *suffix* of w, respectively. The set of prefixes, substrings, and suffixes of a string w is denoted by $\mathit{Prefix}(w)$, $\mathit{Substr}(w)$, and $\mathit{Suffix}(w)$, respectively. The i-th character of a string w is denoted by $w[i]$ for $1 \leq i \leq |w|$, and the substring of a string w that begins at position i and ends at position j is denoted by $w[i..j]$ for $1 \leq i \leq j \leq |w|$. For convenience, let $w[i..j] = \varepsilon$ if $j < i$. For any string w, let w^R denote the reversed string of w, i.e., $w^R = w[|w|]w[|w|-1] \cdots w[1]$. For any character $a \in \Sigma$, we use the following convention that $a \cdot a^{-1} = \varepsilon$. Let $|a^{-1}| = -1$.

2.2 Position Heaps for Multiple Strings

Let $S = \langle w_1, w_2, \ldots, w_k \rangle$ be a sequence of strings such that for any $1 < i \leq k$, $w_i \notin \mathit{Prefix}(w_j)$ for any $1 \leq j < i$. For convenience, we assume that $w_1 = \varepsilon$.

Definition 1 (Sequence hash trees [8]). *The* sequence hash tree *of a sequence $S = \langle w_1, w_2, \ldots, w_k \rangle$ of strings, denoted $\mathit{SHT}(S)$, is a trie structure that is recursively defined as follows: Let $\mathit{SHT}(S)^i = (V_i, E_i)$. Then*

$$\mathit{SHT}(S)^i = \begin{cases} (\{\varepsilon\}, \emptyset) & \text{if } i = 1, \\ (V_{i-1} \cup \{p_i\}, E_{i-1} \cup \{(q_i, c, p_i)\}) & \text{if } 1 \leq i \leq k, \end{cases}$$

where q_i is the longest prefix of w_i which satisfies $q_i \in V_{i-1}$, $c = w_i[|q_i| + 1]$, and p_i is the shortest prefix of w_i which satisfies $p_i \notin V_{i-1}$.

Note that since we have assumed that each $w_i \in S$ is not a prefix of w_j for any $1 \leq j < i$, the new node p_i and new edge (q_i, c, p_i) always exist for each $1 \leq i \leq k$. Clearly $SHT(S)$ contains k nodes (including the root).

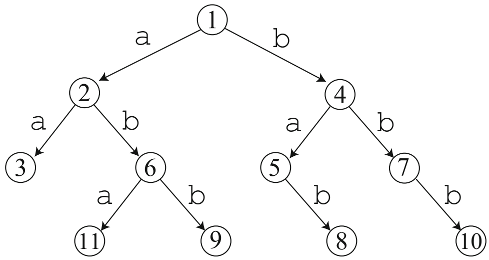

Fig. 1. $PH(W)$ for $W = \{$baa, ababa, abba, bbba$\}$, where $Suffix_{\prec}(W) = \langle \varepsilon$, a, aa, ba, baa, aba, bba, baba, abba, bbba, ababa$\rangle$. The node labeled with integer i represents p_i.

Let $W = \{w_1, w_2, \ldots, w_k\}$ be a set of strings such that $w_i \notin Suffix(w_j)$ for any $1 \leq i \neq j \leq k$. Let $Suffix(W)$ be the set of suffixes of strings in W, i.e., $Suffix(W) = \bigcup_{i=1}^{k} Suffix(w_i)$. Define the order $\prec$ on Σ^* by $x \prec y$ iff $|x| < |y|$, or $|x| = |y|$ and x^R is lexicographically smaller than y^R. Let $Suffix_{\prec}(W)$ be the sequence of strings in $Suffix(W)$ that are ordered w.r.t. $\prec$.

Definition 2 (Position Heaps for Multiple Strings). *The* position heap *for a set W of strings, denoted $PH(W)$, is the sequence hash tree of $Suffix_{\prec}(W)$, i.e., $PH(W) = SHT(Suffix_{\prec}(W))$.*

Lemma 1. *For any set W of strings, let $PH(W) = (V, E)$. For any $v \in V$, $Substr(v) \subseteq V$.*

Proof. For any $v \in V$ with $|v| < 2$, it is clear that $\{\varepsilon, v\} = Substr(v) \subseteq V$. In what follows, we consider $v \in V$ with $|v| \geq 2$. It suffices to show that $v[2..|v|] \in V$ since every prefix of v exists as an ancestor of v and any other substring of v can be regarded as a prefix of a suffix of v. By Definition 2, there exist strings $x_2 \prec x_3 \cdots \prec x_{|v|}$ in $Suffix_{\prec}(W)$ such that $x_i[1..i] = v[1..i]$ for any $2 \leq i \leq |v|$. It follows from the definition of $\prec$ that there exist strings $y_2 \prec y_3 \cdots \prec y_{|v|}$ in $Suffix_{\prec}(W)$ such that $y_i = x_i[2..|x_i|]$ for any $2 \leq i \leq |v|$. Since $y_i[1..i-1] = x_i[2..i] = v[2..i]$ for any $2 \leq i \leq |v|$, it is guaranteed that the node $v[2..i]$ exists in V at least after y_i is inserted to the position heap. Hence $v[2..|v|] \in V$ and the statement holds. □

2.3 Position Heaps and Common Suffix Tries

Our goal is to efficiently construct position heaps for multiple strings. In addition, in our scenario the input strings are given in terms of the following trie:

Definition 3 (Common-suffix tries). *The common-suffix trie of a set W of strings, denoted $CST(W)$, is a reversed trie such that*

1. *each edge is labeled with a character in Σ;*
2. *any two in-coming edges of any node v are labeled with distinct characters;*
3. *each node v is associated with a string that is obtained by concatenating the edge labels in the path from v to the root;*
4. *for each string $w \in W$ there exists a unique leaf with which w is associated.*

An example of $CST(W)$ is illustrated in Fig. 2.

Let n be the number of nodes in $CST(W)$. Clearly, n equals to the cardinality of $\mathit{Suffix}(W)$ (including the empty string). Hence, $CST(W)$ is a natural representation of the set $\mathit{Suffix}(W)$. If N is the total length of strings in W, then $n \leq N+1$ holds. On the other hand, when the strings in W share many suffixes, then $N = \Theta(n^2)$ (e.g., consider the set of strings $\{ab^i \mid 1 \leq i \leq n\}$). Therefore, $CST(W)$ can be regarded as a compact representation of the set W of strings.

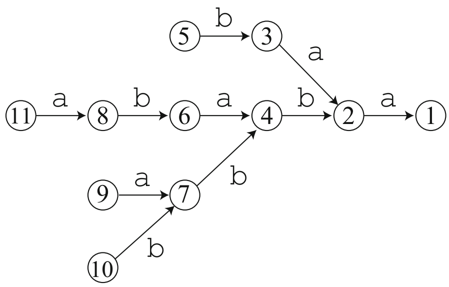

Fig. 2. $CST(W)$ for $W = \{$`baa`, `ababa`, `abba`, `bbba`$\}$. Each node u is associated with $id(u)$.

Our problem of interest is the following:

Problem 1 (Constructing position heap for trie). Given $CST(W)$ for a set W of strings, construct $PH(W)$.

For any $1 \leq i \leq n$, let s_i denote the ith suffix of $\mathit{Suffix}_{\prec}(W)$. Clearly there is a one-to-one correspondence between the elements of $\mathit{Suffix}_{\prec}(W)$ and the nodes of $CST(W)$. Hence, if the path from a node to the root spells out s_i, then we identify this node with s_i. The *parent* of node s_i, denoted $\mathit{parent}(s_i)$, is defined to be $s_i[2..|s_i|]$ (recall that $CST(W)$ is a reversed trie). Any node in the path from s_i to the root of $CST(W)$ is an ancestor of s_i.

Let $id(s_i) = i$. Given $CST(W)$ of size n, we can sort the children of each node in lexicographical order in a total of $O(n)$ time, for a fixed alphabet. Then $id(s_i)$ for all nodes s_i of $CST(W)$ can be readily obtained by a standard breadth-first traversal of $CST(W)$.

For any $1 \leq i \leq n$, where n is the number of nodes of $CST(W)$, let $CST(W)^i$ denote the subtree of $CST(W)$ consisting of nodes s_j with $1 \leq j \leq i$. $PH(W)^i$ is the position heap for $CST(W)^i$ for each $1 \leq i \leq n$, and in our algorithm which follows, we construct $PH(W)$ incrementally, in increasing order of i.

3 Construction of Position Heaps for Common-Suffix Tries

In this section we propose an algorithm that constructs position heaps for common-suffix tries in linear time. Our algorithm is based on a linear time algorithm of Breslauer [7] which constructs suffix trees for common-suffix tries. His algorithm is based on Weiner's linear-time suffix tree construction algorithm for a single string [17]. Below we introduce the *suffix link* for each node of a position heap, which is an analogue of the suffix link for each node of a suffix tree.

Definition 4 (Suffix links). *For any node v of $PH(W) = (V, E)$ and character $a \in \Sigma$, let*

$$\mathit{slink}(a, v) = \begin{cases} av & \text{if } av \in V, \\ \text{undefined} & \text{otherwise.} \end{cases}$$

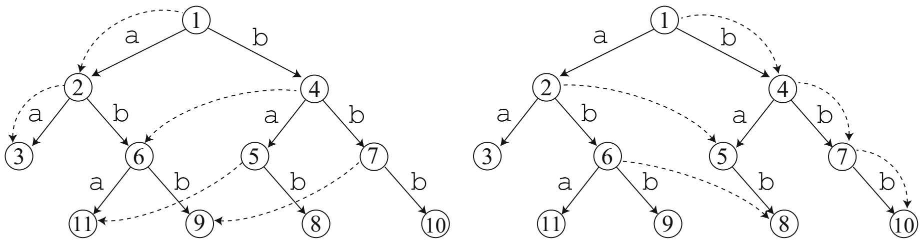

Fig. 3. The broken arrows in the the left (resp. right) diagram show $slink(\mathtt{a}, v)$ (resp. $slink(\mathtt{b}, v)$) for $PH(W)$ of Fig. 1

Fig. 3 shows suffix links for the position heap of Fig. 1.

For convenience, we annotate the position heap with an auxiliary node $\perp$ that represents a^{-1} for any character $a \in \Sigma$, and assume that there are $|\Sigma|$ edges from $\perp$ to the root ε, each of which is labeled with a unique character in Σ. Then $slink(a, \perp) = \varepsilon$ for any character $a \in \Sigma$.

We will use the following data structure that maintains a rooted semi-dynamic tree with marked/unmarked nodes such that the *nearest marked ancestor* in the path from a given node to the root can be found very efficiently.

Lemma 2 ([18,2]). *A semi-dynamic rooted tree can be maintained in linear space so that the following operations are supported in amortized constant time: (1) find the nearest marked ancestor of any node; (2) insert an unmarked node; (3) mark an unmarked node.*

We define the nearest marked ancestor of a node of position heaps as follows:

Definition 5 (Nearest marked ancestor on position heap). *For any node v of $PH(W) = (V, E)$ and character $a \in \Sigma$, let $nma(a, v) = u$ be the lowest ancestor of v such that $slink(a, u) \in V$.*

To answer the query for $nma(a, v)$ in $O(1)$ time given any node u and *any* character $a \in \Sigma$, we construct $|\Sigma|$ copies of $PH(W)$ such that each copy maintains $nma(a, v)$ for all its node v and a character $a \in \Sigma$. In each copy of $PH(W)$ w.r.t. $a \in \Sigma$, we create exactly one edge between $\perp$ and the root that is labeled with a, and one suffix link for a between them as well, since these suffice for this copy tree. This way each copy tree forms a tree, and semi-dynamic nearest marked ancestor queries can be maintained as was mentioned in Lemma 2. Since Σ is fixed, we need only a constant number of copies, thus our data structure of $nma(a, v)$ queries requires a total of $O(n)$ space by Lemma 2.

In the example of Fig. 3, $nma(\mathtt{a}, 9) = 2$, $nma(\mathtt{b}, 9) = 6$, and so on.

Lemma 3 (Level ancestor query [4,3]). *Given a static rooted tree, we can preprocess the tree in linear time and space so that the ℓth node in the path from any node to the root can be found in $O(1)$ time for any $\ell \geq 0$, if such exists.*

For any node u of $CST(W)$ and integer $\ell \geq 0$, let $la(u, \ell)$ denote the ℓth ancestor of u in the path from u to the root. By the above lemma $la(u, \ell)$ can be found in $O(1)$ time after $O(n)$ time and space preprocessing.

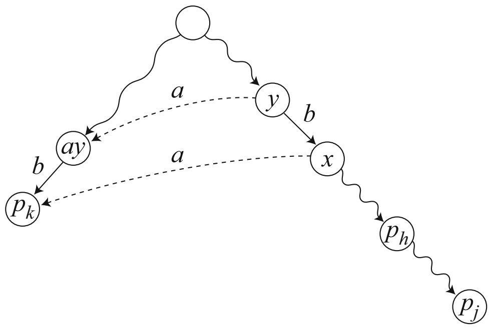

Fig. 4. Illustration for Lemma 4. The straight lines represent edges, and the wavy lines represent paths. The broken arrows represent suffix links w.r.t. character a.

Assume that for $1 < i \leq n$ we have already constructed $PH(W)^{i-1}$ together with the suffix links and the $|\Sigma|$ copies of $PH(W)^{i-1}$ for nma query, and that we are updating them w.r.t. $PH(W)^i$. We need to determine p_i of $PH(W)^{i-1}$, which is the shortest prefix of s_i that is not represented by $PH(W)^{i-1}$. If we search $PH(W)^{i-1}$ for p_i in a naïve way from the root, then it takes $O(|p_i|)$ time, and this leads to overall $O(n^2)$ time complexity. To efficiently find p_i, we will use the following lemma. For any character $a \in \Sigma$ and any node v of $PH(W)^{i-1}$, let $nma_{i-1}(a, v)$ denote the nearest marked ancestor of v w.r.t. a on $PH(W)^{i-1}$.

Lemma 4. *For any* $2 \leq i \leq n$, *let* $j = id(parent(s_i))$. *Then* $p_i = axc$, *where* $a = s_i[1]$, $x = nma_{i-1}(a, p_j)$, *and* $c = s_i[|x| + 2]$.

Proof. $PH(W)^0$ is an empty tree, and since $s_1 = \varepsilon$, $p_1 = \varepsilon$. If $i = 2$, then clearly $j = 1$. For any character $a \in \Sigma$, $nma_1(a, \varepsilon) = \perp$. Since $\perp$ represents a^{-1} and $a \cdot a^{-1} = \varepsilon$ for any character a, it holds that

$$\begin{aligned} p_2 &= s_2[1] \cdot nma_1(s_2[1], p_1) \cdot s_2[|nma_1(s_2[1], p_1)| + 2] \\ &= s_2[1] \cdot nma_1(s_2[1], \varepsilon) \cdot s_2[|nma_1(s_2[1], \varepsilon)| + 2] \\ &= (s_2[1] \cdot (s_2[1])^{-1}) \cdot s_2[-1 + 2] = \varepsilon \cdot s_2[1] = s_2[1]. \end{aligned}$$

For the induction hypothesis, assume that the lemma holds for any $2 \leq i' < i$. Let k be the largest integer such that p_h is the longest proper prefix of p_j with $s_k[1] = s_i[1] = a$, where $h = id(parent(s_k))$. Since p_h is a proper prefix of p_j, $k < i$. By the induction hypothesis, $p_k = ayb$ where $y = nma_{k-1}(a, p_h)$ and $b = s_k[|y| + 2]$. Then p_k is the new node for $PH(W)^k$. Let $x = yb$. Since $slink(a, x) = ax = p_k$ on $PH(W)^k$, $nma_k(a, p_h) = x$. By the assumption of k, $nma_k(a, p_h) = nma_{i-1}(a, p_h) = nma_{i-1}(a, p_j) = x$, and $ax = p_k$ is the longest prefix of s_i that is represented by $PH(W)^{i-1}$ (see also Fig. 4). Hence $p_i = axc$ where $c = s_i[|x| + 2]$. Thus the lemma holds. □

Theorem 1. *Given* $CST(W)$ *with* n *nodes representing a set* W *of strings over a fixed alphabet* Σ, $PH(W)$ *can be constructed in* $O(n)$ *time.*

Proof. We construct the position heap in increasing order of id's of the nodes of $CST(W)$. First we create $PH(W)^1$ which consists only of the root node ε, the auxiliary node $\perp$, and edges and suffix links between $\perp$ and ε. This can be done in $O(1)$ time as Σ is fixed.

Suppose we have already constructed $PH(W)^{i-1}$ for $1 < i \leq n$. Let $j = id(parent(s_i))$, and let a be the edge label from s_i to s_j, i.e., $a = s_i[1]$. Let $x = nma_{i-1}(a, p_j)$. As was shown in Lemma 4, we can locate $p_i = axc$ using a nearest marked ancestor query and the suffix link, as $p_i = slink(a, x)c$ with $c = s_i[|x|+2]$. Then we create a new edge (ax, c, axc). By Lemma 2, node x can be found from node p_j in amortized $O(1)$ time. The character c can be determined in $O(1)$ time by Lemma 3, using the level ancestor query on $CST(W)$.

The auxiliary data structures are updated as follows: By Lemma 1, xc is a node of $PH(W)^i$. We create a new suffix link $slink(a, xc) = axc$, and mark node xc in the copy of $PH(W)^i$ w.r.t. character a. xc is a children of x and can be found in $O(1)$ time from x since Σ is fixed. Marking node xc in the copy tree can be conducted in amortized $O(1)$ time by Lemma 2.

Consequently, $PH(W)$ can be constructed in a total of $O(n)$ time. □

4 Pattern Matching with Augmented *PH*(*W*)

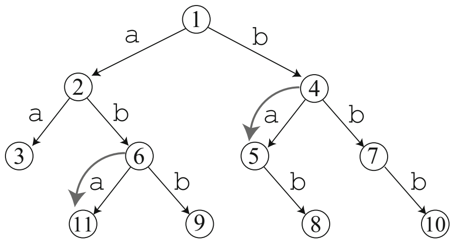

Fig. 5. Illustration for $PH(W)$ of Fig. 1 annotated with maximal reach pointers, which are shown by shadowed arcs. The maximal reach pointers such that $mrp(p_i) = p_i$ are omitted for simplicity.

In this section we describe how to solve the following pattern matching problem for a set of strings W using $PH(W)$.

Problem 2. Given $CST(W)$ for a set W of strings and a pattern string $q \in \Sigma^*$, return all i such that $s_i[1..|q|] = q$, where s_i is a node of $CST(W)$.

In our algorithm to solve Problem 2, we will use the following pointers.

Definition 6 (Maximal reach pointer). *Let n be the number of nodes in $CST(W)$. For any node s_i of $CST(W)$, $1 \leq i \leq n$, let p_i be the shortest prefix of s_i that is not represented by $PH(W)^{i-1}$. Then $mrp(p_i)$ is a pointer from p_i to the longest prefix of s_i that is represented in $PH(W)^n$.*

Fig. 5 shows $PH(W)$ of Fig. 1 annotated with maximal reach pointers. See also $CST(W)$ of Fig. 2. $s_6 = \texttt{aba}$ and $p_6 = \texttt{ab}$, and since there is a node aba in $PH(W)$, $mrp(\texttt{ab}) = \texttt{aba}$.

In what follows, we describe how we can compute all occurrences of a give pattern q in $CST(W)$ using $PH(W)$. The following lemma is useful.

Lemma 5. *Given integer i with $1 \leq i \leq n$ and a node p of $PH(W)$, by using $mrp(p)$ it takes $O(1)$ time to determine whether i is an occurrence of p in $CST(W)$, i.e., $s_i[1..|p|] = p$.*

Proof. The proof is essentially the same as the proof for the case where the input is a single string, given in [9]. □

We begin with the case where a given pattern q is represented by $PH(W)$.

Lemma 6. *If pattern string q is represented by $PH(W)$, then we can compute all occurrences of q in $CST(W)$ in $O(m + r)$ time, where $m = |q|$ and r is the number of occurrences to report.*

Proof. We search $PH(W)$ for pattern q from the root. This takes $O(m)$ time as the alphabet is fixed. For each proper prefix p_i of q found in the path from the root to q, we can check whether i is an occurrence of q or not in $O(1)$ time by Lemma 5. Since there are m such prefixes, this takes a total of $O(m)$ time.

There can be other occurrences of q. Let p_j be any node of $PH(W)$ that is in the subtree rooted at q. Since q is a prefix of p_j, q is also a prefix of s_j, and thus j is an occurrence of q in $CST(W)$. We traverse the subtree rooted at q and report all positions corresponding to the nodes in the subtree, in $O(r)$ time. □

Secondly, we consider the case where pattern q is *not* represented by $PH(W)$.

Lemma 7. *If pattern string q is not represented by $PH(W)$, then there are at most $|q| - 1$ occurrences of q in $CST(W)$.*

Proof. Let r be the number of occurrences of q in $CST(W)$, and assume on the contrary that $r \geq |q|$. Let k be the largest occurrence of q in $CST(W)$. Then, the length of p_k must be at least $|q|$, since there are $r - 1$ occurrences of q in $CST(W)$ that are smaller than k, and $r - 1 \geq |q| - 1$. Thus q is a prefix of p_k. Since p_k is a node of $PH(W)$, q is also a node of $PH(W)$. However, this contradicts the assumption that q is not represented by $PH(W)$. □

Each occurrence of q mentioned in the above lemma corresponds to a unique prefix of q that is represented by $PH(W)$. Using this property, we can find occurrences of q as will be described in the following lemma:

Lemma 8. *If pattern string q is not represented by $PH(W)$, then we can compute all occurrences of q in $CST(W)$ in $O(m)$ time where $m = |q|$, using $PH(W)$ annotated with the maximal reach pointers.*

Proof. We factorize the pattern string as $q = q(1)q(2) \cdots q(g)$ such that $q(1)$ is the longest prefix of q that is represented by $PH(W)$, and for each $2 \leq j \leq g$, $q(j)$ is the longest prefix of $q[\sum_{h=1}^{j-1} |q(h)| + 1..|q|]$ that is represented by $PH(W)$. This factorization can be computed in $O(m)$ time using $PH(W)$ if it exists. This factorization does not exist if and only if q contains a character c which does not exist in $CST(W)$. In this case q clearly does not occur in $CST(W)$. In what follows, we assume the above factorization of q exists, and we process each factor $q(j)$ in increasing order of j, as follows. For any $1 \leq j < g$, we consider a set L_j of positions where $q[1..\sum_{h=1}^{j} |q(h)|] = q(1)q(2) \cdots q(j)$ occurs in $CST(W)$, which are candidates for an occurrence of q.

- If $j = 1$: We compute L_1 which consists of i such that p_i is a prefix of $q(1)$ and $mrp(p_i) = q(1)$. Note that any i with $mrp(p_i) \neq q(1)$ cannot be an occurrence of q since $q(1) \cdot q(2)[1]$ is not represented by $PH(W)$. Namely $q(1)$ occurs at i for any $i \in L_1$ and q does not occur at i' for any $i' \notin L_1$. Clearly $|L_1| \leq |q(1)|$ and L_1 can be computed in $O(|q(1)|)$ time.

– If $2 \le j < g$: Assume that L_{j-1} is already computed. For any $i \in L_{j-1}$, let $e(i) = id(la(s_i, \sum_{h=1}^{j-1} |q(h)|))$, i.e., $s_{e(i)}$ is the $(\sum_{h=1}^{j-1} |q(h)|)$-th ancestor of s_i in $CST(W)$. By Lemma 3 we can compute $e(i)$ in $O(1)$ time. Note that $q(1)q(2)\cdots q(j)$ occurs at i if and only if $q(j)$ occurs at $e(i)$. Then we compute L_j which consists of $i \in L_{j-1}$ such that $mrp(p_{e(i)}) = q(j)$. This can be done in $O(|L_{j-1}| + |q(j)|)$ time, where $|q(j)|$ is the cost of locating $q(j)$ in $PH(W)$. We note that $|L_j| \le |q(j)|$ holds.
– If $j = g$: We have L_{g-1}. In a similar way to the above case, $q(1)q(2)\cdots q(g)$ occurs at i if and only if $q(g)$ occurs at $e(i)$ for some $i \in L_{g-1}$. It follows from Lemma 5 that we can determine whether $e(i)$ is an occurrence of $q(g)$ in $O(1)$ time for any $i \in L_{g-1}$, and hence we can compute all positions where q occurs in $CST(W)$ in $O(|L_{g-1}| + |q(g)|)$ time.

In total, it takes $O(|q(1)| + \sum_{j=2}^{g}(|L_{j-1}| + |q(j)|)) = O(|q(1)| + \sum_{j=2}^{g}(|q(j-1)| + |q(j)|)) = O(m)$ time. □

What remains is how to compute the maximal reach pointers of the nodes of $PH(W)$. We have the following result.

Lemma 9. *Given $PH(W)$ with n nodes, we can compute $mrp(p_i)$ in a total of $O(n)$ time for all $1 \le i \le n$, assuming Σ is fixed.*

Proof. We can compute $mrp(p_i)$ for all $1 \le i \le n$ in a similar way to the computation of the suffix links described in the proof of Theorem 1. We compute $mrp(p_i)$ in increasing order of i. Clearly $mrp(p_1) = mrp(\varepsilon) = \varepsilon$. Assume that we have already computed $mrp(p_{i-1})$ for $1 < i \le n$. Let $j = id(parent(s_i))$ and $y = mrp(p_j)$. Since $j < i$, by the induction hypothesis $mrp(p_j)$ has been computed. y is the longest prefix of s_j that is represented by $PH(W)^n$, and hence $mrp(p_i)$ is at most $|y| + 1$ long, since otherwise it contradicts Lemma 1. This implies that $mrp(p_i) = s_i[1] \cdot nma_n(s_i[1], y) = slink(s_i[1], nma_n(s_1[1], y))$. By using the suffix link and by Lemma 2, $mrp(p_i)$ can be computed in amortized $O(1)$ time for a fixed alphabet. This completes the proof. □

Following the above lemmas, we obtain the main result of this section:

Theorem 2. *We can augment $PH(W)$ in $O(n)$ time and space so that all occurrences of a given pattern in $CST(W)$ can be computed in $O(m + r)$ time, where m is the length of the pattern and r is the number of occurrences to report.*

5 Updating $PH(W)$ When $CST(W)$ Is Edited

Ehrenfeucht et al. [9] showed how to update the position heap of a single string when a block of characters of size b is inserted/deleted from the string, in amortized $O((h' + b)h' \log n')$ time, where h' is the maximum height of the position heap and n' is the maximum length of the string while editing. We note that in that dynamic scenario the time complexity of pattern matching requires an extra multiplicative $\log n$ factor compared to the static scenario, since operations (including random access) on a string represented by a dynamic array require $O(\log n)$ amortized time.

In this section, we consider updates on the position heap when the input common-suffix trie is edited. As a first step towards rich edit operations, we deal with the following operations:

- `AddLeaf`: Add a new leaf node from an arbitrary node u in the common-suffix trie with edge label $a \in \Sigma$, where no edges from u to its children are labeled with a, and update the position heap accordingly.
- `RemoveLeaf`: Remove an arbitrary leaf and its corresponding edge from the common suffix trie, and update the position heap accordingly.

We will use the following result for dynamic trees.

Theorem 3 ([1]). *A dynamic tree with n nodes can be maintained in $O(n)$ space so that insertion/deletion of a node, and level ancestor queries are supported in $O(\log n)$ time.*

Since node-to-node correspondence of between the common-suffix trie and the position heap can be dynamically changed, we maintain a pointer $cstp(p)$ for any node p of the position heap such that $cstp(p)$ always points to the corresponding node of the common-suffix trie.

Here we give some remarks on $id(v)$ of node v in $CST(W)$. In the previous sections, $id(v)$ is equivalent to the order of v in $Suffix_{\prec}(W)$. However when W is updated, maintaining such values requires $\Theta(n)$ time. To overcome this, we assign to v a rational number $id(v) = (id(pre_W(v)) + id(suc_W(v)))/2$, where $pre_W(v)$ and $suc_W(v)$ are the predecessor/successor of v in $Suffix_{\prec}(W)$, respectively. We maintain *pre* and *suc* by a dynamic list *bflist*. By Theorem 3, insertion, deletion and random access on *bflist* can be supported in $O(\log n)$ time.

In what follows, we show how to maintain (1) the data structure for level ancestor queries on $CST(W)$, (2) the augmented position heap $PH(W)$, and (3) *bflist* so that we can solve the pattern matching problem on $CST(W)$ in $O(m \log n)$ time, where the $\log n$ factor comes from level ancestor queries on the dynamic common-suffix trie. By Definition 2, the main task of updating the position heap is to keep a heap property w.r.t. $id(cstp(p))$.

Theorem 4. *Operations* `AddLeaf` *and* `RemoveLeaf` *can be supported in $O(h \log n)$ and $O(h)$ time, respectively, where h is the height of $PH(W)$.*

Proof. In both operations, the data structure for level ancestor queries can be updated in $O(\log n)$ time by Theorem 3. Let $CST(W')$ denote the new common-suffix trie after addition/removal of a leaf. Also we will distinguish pointers to the common-suffix trie before and after the update by $cstp$ and $cstp'$, respectively.

`AddLeaf`: Let v be the new leaf added to $CST(W)$, and let u be the parent of v in $CST(W')$. Firstly we search for $pre_{W'}(v)$ and $suc_{W'}(v)$ to determine $id(v)$. We can find them in $O(\log^2 n)$ time as follows:

- If u has a child in $CST(W)$, then at least one of $pre_{W'}(v)$ and $suc_{W'}(v)$ must be a child of u, which can be found in $O(1)$ time as Σ is fixed. Then the other can be found in $O(1)$ time using *bflist*.

– If u has no child, we search for node $v' = \arg\max\{id(z) \mid id(parent(z)) < id(u)\}$, i.e., $v' = pre_{W'}(v)$. Since every $id(v)$ is monotonically increasing in breadth-first order, $id(parent(v))$ is also monotone, and hence we can find v' in $O(\log^2 n)$ time by a binary search on *bflist* based on level ancestor queries. $suc_{W'}(v)$ can be obtained from $pre_{W'}(v)$ in constant time using *bflist*.

Next we traverse $PH(W)$ from the root until finding the first node p which satisfies $id(v) < id(cstp(p))$. If such does not exist, the traversal is finished at node p' such that $cstp(p')$ is the longest prefix of v that is represented in $PH(W)$. Since v cannot be a prefix of p', we make new leaf $q = v[1..|p'| + 1]$ from p'. If p exists, $cstp'(p) = v$ and floated $cstp(p)$ is pushed down, i.e., $cstp'(q) = cstp(p)$ with $q = cstp(p)[1..|p| + 1] \in PH(W')$, and if q exists in $PH(W)$, floated $cstp(q)$ is pushed down recursively until getting $q \notin PH(W)$.

While we push down floated node pointers, we make the corresponding maximal reach pointers accompanied. Also, for $r \in PH(W')$ with $cstp'(r) = v$, $mrp(r)$ can be computed in $O(h)$ time by traversing $PH(W')$ from the root. mrp is dynamically maintained by updating $mrp(r)$ to q for any $r \in PH(W')$ such that $mrp(r) = parent(q)$ and $cstp(r)[|q|] = q[|q|]$.

Since only the nodes in the path from the root to q (the new leaf in $PH(W')$) are affected by the update, updating from $cstp$ to $cstp'$ and updating mrp takes $O(h \log n)$ time, where the $\log n$ factor comes from level ancestor queries on the common-suffix trie. Hence the update on `AddLeaf` takes $O(h \log n + \log^2 n) = O((h + \log n) \log n) = O(h \log n)$ time overall, where the last equation is derived from $n \leq |\Sigma|^h$ and $\log_2 n \in O(h)$.

`RemoveLeaf`: Let v be the node to be removed from $CST(W)$, and let p be the node in $PH(W)$ such that $cstp(p) = v$. What is required is to "remove affection" of v from $PH(W)$, i.e., clear $cstp(p)$ and if needed float up descendants keeping a heap property. More specifically, if p has a child $cstp'(p) = cstp(q)$ where q is the child of p with the minimum id among the children of p, and if q has a child, then repeat floating up the child recursively until getting q which has no child in $PH(W)$. Finally we get the leaf node q to be deleted from the position heap.

While we float up node pointers, we make the corresponding maximal reach pointers accompanied. In addition, the update of mrp is accomplished by updating $mrp(r)$ to be $parent(q)$ for any $r \in PH(W)$ with $mrp(r) = q$.

Since only the nodes in the path from the root to q are affected by the update, all the updates require a total of $O(h)$ time. Note that it is different from the case of `AddLeaf` in that no level ancestor queries on the common-suffix trie are required. Hence the update on `RemoveLeaf` takes in total $O(h + \log n) = O(h)$ time. □

References

1. Alstrup, S., Holm, J., de Lichtenberg, K., Thorup, M.: Maintaining information in fully dynamic trees with top trees. ACM Transactions on Algorithms 1(2), 243–264 (2005)
2. Amir, A., Farach, M., Idury, R.M., La Poutré, J.A., Schäffer, A.A.: Improved dynamic dictionary matching. Information and Computation 119(2), 258–282 (1995)

3. Bender, M.A., Farach-Colton, M.: The level ancestor problem simplified. Theor. Comput. Sci. 321(1), 5–12 (2004)
4. Berkman, O., Vishkin, U.: Finding level-ancestors in trees. J. Comput. Syst. Sci. 48(2), 214–230 (1994)
5. Blumer, A., Blumer, J., Haussler, D., Mcconnell, R., Ehrenfeucht, A.: Complete inverted files for efficient text retrieval and analysis. J. ACM 34(3), 578–595 (1987)
6. Blumer, A., Blumer, J., Haussler, D., Ehrenfeucht, A., Chen, M.T., Seiferas, J.: The smallest automaton recognizing the subwords of a text. Theoret. Comput. Sci. 40, 31–55 (1985)
7. Breslauer, D.: The suffix tree of a tree and minimizing sequential transducers. Theoretical Computer Science 191(1-2), 131–144 (1998)
8. Coffman, E., Eve, J.: File structures using hashing functions. Communications of the ACM 13, 427–432 (1970)
9. Ehrenfeucht, A., McConnell, R.M., Osheim, N., Woo, S.-W.: Position heaps: A simple and dynamic text indexing data structure. Journal of Discrete Algorithms 9(1), 100–121 (2011)
10. Inenaga, S., Hoshino, H., Shinohara, A., Takeda, M., Arikawa, S., Mauri, G., Pavesi, G.: On-line construction of compact directed acyclic word graphs. Discrete Applied Mathematics 146(2), 156–179 (2005)
11. Kärkkäinen, J., Sanders, P., Burkhardt, S.: Linear work suffix array construction. J. ACM 53(6), 918–936 (2006)
12. Kosaraju, S.R.: Efficient tree pattern matching. In: Proc. FOCS 1989, pp. 178–183 (1989)
13. Kucherov, G.: On-Line Construction of Position Heaps. In: Grossi, R., Sebastiani, F., Silvestri, F. (eds.) SPIRE 2011. LNCS, vol. 7024, pp. 326–337. Springer, Heidelberg (2011)
14. Manber, U., Myers, G.: Suffix arrays: A new method for on-line string searches. SIAM J. Computing 22(5), 935–948 (1993)
15. Shibuya, T.: Constructing the suffix tree of a tree with a large alphabet. IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences E86-A(5), 1061–1066 (2003)
16. Ukkonen, E.: On-line construction of suffix trees. Algorithmica 14(3), 249–260 (1995)
17. Weiner, P.: Linear pattern-matching algorithms. In: Proc. of 14th IEEE Ann. Symp. on Switching and Automata Theory, pp. 1–11 (1973)
18. Westbrook, J.: Fast Incremental Planarity Testing. In: Kuich, W. (ed.) ICALP 1992. LNCS, vol. 623, pp. 342–353. Springer, Heidelberg (1992)