Journal of Discrete Algorithms 20 (2013) 3–11

Contents lists available at SciVerse ScienceDirect

Journal of Discrete Algorithms

www.elsevier.com/locate/jda

# On-line construction of position heaps ☆

Gregory Kucherov [a,b,*]

[a] Université Paris-Est & CNRS, Laboratoire d'Informatique Gaspard Monge, Marne-la-Vallée, France
[b] Department of Computer Science, Ben-Gurion University of the Negev, Be'er Sheva, Israel

ARTICLE INFO

*Article history:*
Available online 26 September 2012

*Keywords:*
String algorithms
Data structures
Text index
Position heap

ABSTRACT

We propose a simple linear-time on-line algorithm for constructing a position heap for a string (Ehrenfeucht et al., 2011 [8]). Our definition of position heap differs slightly from the one proposed in Ehrenfeucht et al. (2011) [8] in that it considers the suffixes ordered in the descending order of length. Our construction is based on classic suffix pointers and resembles Ukkonen's algorithm for suffix trees (Ukkonen, 1995 [17]). Using suffix pointers, the position heap can be extended into the augmented position heap that allows for a linear-time string matching algorithm (Ehrenfeucht et al., 2011 [8]).

© 2012 Elsevier B.V. All rights reserved.

## 1. Introduction

The theory of string algorithms developed beautiful data structures for string matching and text indexing. Among them, *suffix tree* and *suffix array* are most widely used structures, providing efficient solutions for a wide range of applications [7,11]. The DAWG (*Directed Acyclic Word Graph*) [1], also known as *suffix automaton* [5], is another elegant structure that can be used both as a text index [1] or as a matching automaton [6,7].

Recently, a new *position heap* data structure was proposed [8]. Similar to the suffix tree, DAWG or suffix array, position heap allows for a pre-processing of a text string in order to efficiently search for patterns in it. As for the above-mentioned data structures, a position heap for a string of length $n$ can be constructed in time $O(n)$. Then all locations of a pattern of length $m$ can be found in time $O(m + occ)$, where $occ$ is the number of occurrences.

The construction algorithm of [8] processes the string from right to left, like Weiner's algorithm does for suffix trees [18]. Moreover, the construction requires a so-called dual heap, which is an additional trie on the same set of nodes. The position heap and its dual heap are constructed simultaneously.

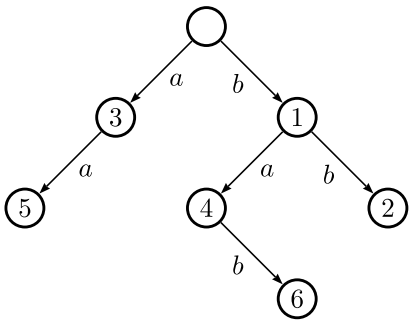
To obtain a linear-time pattern matching algorithm of [8], the position heap should be post-processed in order to add some additional information, resulting in the *augmented position heap*. The most important element of this information includes so-called *maximal-reach pointers* assigned to certain nodes. Computing these pointers makes use of the dual heap too.

In this paper, we propose a different construction of the position heap. First, we change the definition of the position heap by reversing the order of suffixes and thus allowing for the left-to-right traversal of the input string. The modified definition, however, preserves good properties of the position heap and does not affect the string matching algorithm proposed in [8]. For this modified definition, we propose an *on-line* algorithm for constructing the position heap. Our algorithm does not use the dual heap, replacing it by classic *suffix pointers* used for constructing suffix trees by Ukkonen's algorithm [17] or for constructing the DAWG [1]. Our algorithm is simple and can be compared to Ukkonen's algorithm for suffix trees,

☆ A preliminary version of this paper has been presented in the 18th International Symposium on String Processing and Information Retrieval (SPIRE), Pisa, Italy, August 2011.

* Correspondence to: Université Paris-Est & CNRS, Laboratoire d'Informatique Gaspard Monge, 5, Bd Descartes, 77454, Marne-la-Vallée, France.
*E-mail address:* Gregory.Kucherov@univ-mlv.fr.

1570-8667/$ – see front matter © 2012 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.jda.2012.08.002




1. $babbaab$
2. $bbab$
3. $ab$
4. $baaa$
5. $aabab$
6. $babaaba$

**Fig. 1.** Sequence hash tree for the set of strings shown on the right. Each node stores the rank of the corresponding string in the set.

as opposed to Weiner's algorithm that constructs the suffix tree by inserting suffixes right-to-left (i.e. shortest first). We deliberately use some terminology of Ukkonen's algorithm to underline this similarity.

We further show that the augmented position heap can be easily constructed using suffix pointers. Thus, we completely eliminate the use of the dual heap, replacing it by suffix pointers for constructing both the position heap and its augmented version. Even if this replacement does not provide an immediate improvement in space or running time, we believe that our construction is conceptually simpler and more natural.

Throughout the paper, we assume we are given a constant-size alphabet $A$. Positions of strings over $A$ are numbered from 1, that is, a string $w$ of *length* $k$ is $w[1]\dots w[k]$. The length $k$ of $w$ is denoted by $|w|$. $w[i..j]$ denotes substring $w[i]\dots w[j]$.

A *trie* (term attributed to Fredkin [10]) is a simple natural data structure for storing a set of strings. It is a tree with edges labeled by alphabet letters, such that for any internal node, the edges leading to the children nodes are labeled by distinct letters. In this paper, we assume the edges to be directed towards leaves, and call an edge labeled by a letter $a$ an $a$-edge. A *label* of a node (*path label*) is the string formed by the letters labeling the edges of the path from the root to this node. Given a trie, a string $w$ is said to be represented in the trie if it is a path label of some node. The corresponding node will then be denoted by $\overline{w}$.

## 2. Definition of position heap

To define position heaps, we first need to introduce the *sequence hash tree* proposed by Coffman and Eve back in 1970 [3] as a data structure for implementing hash tables. Assume we are given an ordered set of strings $W = \{w_1, \dots, w_n\}$ and assume for now that no $w_i$ is a prefix of $w_j$ for any $j < i$. The sequence hash tree for $W$, denoted $SHT(W)$, is a trie defined by the following iterative construction. We start with the tree $SHT_0(W)$ consisting of a single root node *root*.[1] We then construct $SHT(W)$ by processing strings $w_1, \dots, w_k$ in this order and for each $w_i$, adding one node to the tree. By induction, assume that $SHT_i(W)$ is the sequence hash tree for $\{w_1, \dots, w_i\}$. To construct $SHT_{i+1}(W)$, we find the shortest prefix $v$ of $w_{i+1}$ which is *not* represented in $SHT_i(W)$. Note that by our assumption, such a prefix always exists. Let $v = v'a$, $a \in A$, i.e. $v'$ is the longest prefix of $w_{i+1}$ represented in $SHT_i(W)$. Then $SHT_{i+1}(W)$ is obtained from $SHT_i(W)$ by adding a new node as a child of $v'$ connected to $v'$ by an $a$-edge and pointing to $w_{i+1}$. After inserting all strings of $W$, we obtain $SHT(W)$, that is $SHT(W) = SHT_k(W)$. Thus, $SHT(W)$ is a trie of $n+1$ nodes such that a node pointing to $w_i$ is labeled by some prefix of $w_i$. Note that the size of the sequence hash tree depends only on the number of strings in the set and does not depend on the length of those. An example of sequence hash tree is given in Fig. 1.

We now define the *position heap* of a string $T$. In [8], the position heap for $T$ is defined as the sequence hash tree for the set of suffixes of $T$, where *the suffixes are ordered in the ascending order of length*, i.e. from right to left. This insures, in particular, the condition that no suffix is a prefix of a previously inserted suffix, and then no suffix is already represented in the position heap at the time of its insertion.

In this paper, we define the position heap of $T$ to be the sequence hash tree for the set of suffixes of $T$, where *the suffixes are ordered in the descending order of length*, i.e. from left to right. From now on, we stick to this order. An immediate observation is that the assumption of the suffix hash tree does not hold anymore, and it may occur that an inserted suffix is already represented in the position heap by an existing node. One easy way to cope with this is to systematically assume that $T$ is ended by a special sentinel symbol \$, like it is generally assumed for the suffix tree.

On the other hand, as we will be interested in an on-line construction of the position heap, we will still need to construct the position heap for strings without the ending sentinel symbol. For that, we have to slightly change the definition of sequence hash tree of a set $W$, by allowing one node to point to several strings of $W$. The definition of the position heap extends then to any string, with the only difference that inserting a suffix may no longer lead to the creation of a new

[1] This definition agrees with the definition of [3] but is slightly different from that of [8] which defines the root to store $w_1$. The difference is insignificant, however.

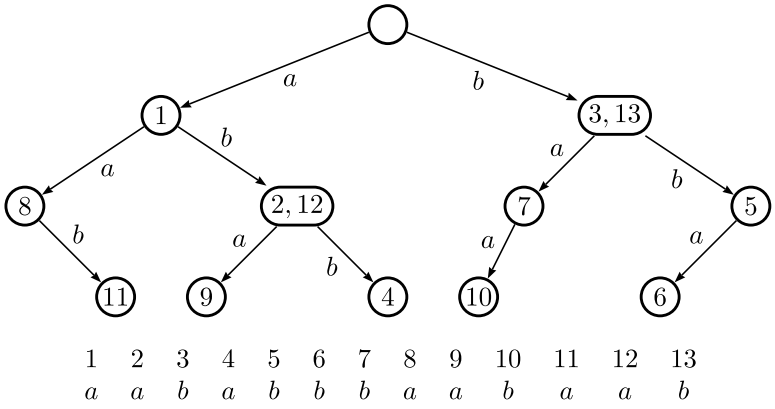



**Fig. 2.** Position heap for string *aababbbaabaab*. Double nodes store pairs of positions.

node, but to adding a pointer to this suffix to an existing node. This feature, however, will be used in a very restricted way, as the following observation shows.

**Lemma 1.** *Let $W$ be a set of distinct strings. Then every node of $SHT(W)$ points to at most two strings of $W$.*

**Proof.** The only situation when a new pointer gets inserted to an existing node is when the inserted string $w_{i+1}$ is already represented in $SHT_i(W)$. Since all strings of $W$ are distinct, this situation may occur only once for each node. Therefore, each node of $SHT(W)$ points to one or two strings of $W$. □

As a consequence of Lemma 1, a position heap contains two types of nodes, pointing respectively to one and two suffixes of $T$. The former will be called *regular nodes* and the latter *double nodes*. We naturally assume that a pointer to a suffix is simply the starting position of that suffix, therefore regular and double nodes store one and two string positions respectively. Hereafter we interchangeably refer to "suffixes" and "positions" when the underlying string is unambiguously defined.

Fig. 2 provides an example of a position heap.

## 3. Properties of position heap

Denote by $PH(T)$ the position heap for a string $T[1..n]$ as defined in the previous section. In the following theorem, we summarize some key properties of the position heap.

**Theorem 1.** *(See [8].) Consider $PH(T[1..n])$. The following properties hold.*

(i) *A substring $w$ of $T$ is represented in $PH(T)$ iff $T$ contains occurrences of strings $w[1..1], w[1..2], w[1..3], \ldots, w[1..|w|]$, appearing at increasing positions in this order.*
(ii) *The labels of all nodes of $PH(T)$ form a factorial set. That is, if a string is represented in $PH(T)$, all its substrings are represented too.*
(iii) *The depth of $PH(T)$ is no more than $2h(T)$, where $h(T)$ is the length of the longest substring $w$ of $T$ which occurs $|w|$ times in $T$ (possibly with overlap).*
(iv) *If a substring $w$ occurs in $T$ at least $|w|$ times, then $w$ is represented in $PH(T)$. Inversely, if $w$ is not represented in $PH(T)$ and $w'$ is the longest prefix of $w$ which is represented, then $w$ cannot occur in $T$ more than $|w'|$ times.*

**Proof.** (i) The 'if'-part follows immediately from the definition of $PH(T)$ and the left-to-right order of suffixes. If a substring $w$ is represented in $PH(T)$, then nodes $\overline{w[1..1]}, \overline{w[1..2]}, \overline{w[1..3]}, \ldots, \overline{w[1..|w|]}$ have been created in this respective order. The creation of each such node $\overline{w[1..\ell]}$ has been triggered by an insertion of a suffix starting with $w[1..\ell]$. Since suffixes are inserted from left to right, property (i) follows.

Properties (ii)–(iv) have been established in [8] but remain valid for our definition of position heap when suffixes are inserted from left to right. Actually, these properties are valid for *any* order of inserting suffixes into the position heap.

(ii) It is sufficient to show that if some string $w[1..\ell]$ is represented in $PH(T)$, then both $w[i..\ell-1]$ and $w[2..\ell]$ are represented too. For $w[1..\ell-1]$, this is obvious from construction. For $w[2..\ell]$, this can be seen from Property (i). Indeed, if strings $w[1..1], w[1..2], w[1..3], \ldots, w[1..\ell]$ appear in $T$ in this relative order, then we have strings $w[2..2], w[2..3], \ldots, w[2..\ell]$ appearing in $T$ at increasing positions too. By Property (i), this ensures that $w[2..\ell]$ is represented in $PH(T)$.

(iii) Let $\overline{w}$ be one of the deepest nodes of $PH(T)$, i.e. the depth of $PH(T)$ is $d = |w|$. From Property (i), strings $w[1..\lceil d/2\rceil], w[1..\lceil d/2\rceil + 1], \ldots, w[1..d]$ occur at $T$ at distinct positions, and therefore $w[1..\lceil d/2\rceil]$ occurs at least $\lceil d/2\rceil$ times in $T$. Then $h(T) \geqslant \lceil d/2\rceil$ and the depth $d$ of $PH(T)$ is bounded by $2h(T)$.

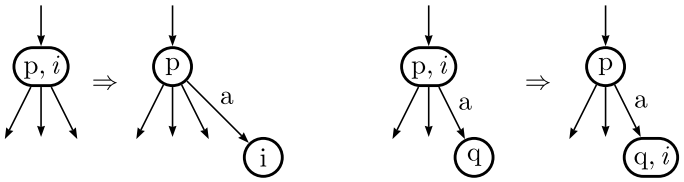



**Fig. 3.** Updating secondary position $i$ when transforming $PH(T[1..k])$ into $PH(T[1..k+1])$: *first case* (left) and *second case* (right).

(iv) If a substring $w$ occurs in $T$ at least $|w|$ times, then there exist successive occurrences of $w[1..1]$, $w[1..2]$, $w[1..3]$, $\ldots$, $w[1..|w|]$, and, by Property (i), $w$ is represented in $PH(T)$. Assume now that $w$ is not represented in $PH(T)$ and $w'$ is the longest prefix of $w$ which is represented. Assume further that $a$ is the letter that follows prefix $w'$ in $w$. Observe that $w'a$ occurs at most $|w'|$ times, as the contrary would mean that $w'a$ is represented too, which contradicts the choice of $w'$. Therefore, $w'a$ occurs at most $|w'|$ times and so does $w$. □

Properties (iii) and (iv) show that the position heap of a string "adapts" to the frequencies of its substrings. In particular, if a string is "frequent" (occurs as many times as it is long), then it is necessarily represented in the position heap. On the other hand, if it is not represented, it has less occurrences than its length. The latter property is crucial for obtaining a linear-time string matching algorithm of [8].

## 4. On-line construction algorithm

Let us have a closer look at the properties of double nodes of a position heap $PH(T)$. Each such node stores two positions $i, j$ of $T$. Assume $i < j$, then positions $i$ and $j$ will be called the *primary* and the *secondary* positions respectively.

**Lemma 2.** *Let $T = T[1..n]$. If $j < n$ is the secondary position of some node of $PH(T)$, then so is $j+1$.*

**Proof.** Consider $PH(T)$ for some string $T[1..n]$. Assume $i$, $j$, $i < j$, are respectively primary and secondary positions of some node. This means that by the time the suffix $T[j..n]$ is inserted into $PH(T)$ during its construction, node $\overline{T[j..n]}$ already exists. By Theorem 1(ii), node $\overline{T[j+1..n]}$ exists too. *A fortiori*, node $\overline{T[j+1..n]}$ exists when $T[j+1..n]$ is inserted into $PH(T)$. Therefore, $j+1$ becomes the secondary position of that node after the insertion of suffix $T[j+1..n]$. □

Lemma 2 implies that all positions of $T[1..n]$ are split into two intervals: primary positions $[1..s-1]$, for some position $s$, and secondary positions $[s..n]$. Position $s$ will be called *active secondary position*, or *active position* for short.

Assume we have constructed the position heap $PH(T[1..k])$ for some prefix $T[1..k]$ of the input string $T[1..n]$. Let us analyze the differences between $PH(T[1..k])$ and $PH(T[1..k+1])$ and the modifications that need to be made to transform the former into the latter.

Let $s$ be the active position of $T[1..k]$. First observe that for suffixes $1, \ldots, s-1$, no changes need to be made. Inserting each suffix $T[i..k]$ for $1 \leqslant i \leqslant s-1$ into $PH(T[1..k])$ led to the creation of a new node. This means that by the time this suffix was inserted into $PH(T[1..k])$, some prefix $T[i..\ell]$ of $T[i..k]$, $\ell \leqslant k$, was not represented in the position heap, which led to the creation of a new node $\overline{T[i..\ell]}$ with the minimal such $\ell$. This shows that inserting suffixes $1, \ldots, s-1$ involve completely identical steps in the construction of both $PH(T[1..k])$ and $PH(T[1..k+1])$.

The situation is different for the secondary positions $s, \ldots, k$. Each suffix $T[i..k]$ for $s \leqslant i \leqslant k$ was already represented in $PH(T[1..k])$ at the moment of its insertion, and then resulted in the addition of the secondary position $i$ to the node $\overline{T[i..k]}$. When inserting the corresponding suffix $T[i..k+1]$ into the position heap $PH(T[1..k+1])$, two cases arise. In the *first case*, inserting the suffix $T[i..k+1]$ leads to the creation of the new node $\overline{T[i..k+1]}$ if this node does not exist yet. Position $i$ then becomes the primary position of this new node. Observe that this only occurs when $PH(T[1..k])$ does not contain an $T[k+1]$-edge outgoing from the node $\overline{T[i..k]}$. It is easily seen that such an edge cannot appear by the time of insertion of $T[i..k+1]$ into $PH(T[1..k+1])$ if it was not already present in $PH(T[1..k])$. In the *second case*, node $T[i..k]$ has an outgoing $T[k+1]$-edge in $PH(T[1..k])$, and in the construction of $PH(T[1..k+1])$, the secondary position $i$ stored in this node should be "moved" to the child node $\overline{T[i..k+1]}$. It becomes then the secondary position of this node. The two cases are illustrated in Fig. 3.

Observe now that if for a secondary position $i$, the corresponding node $\overline{T[i..k]}$ has an outgoing $T[k+1]$-edge, then so does the node $\overline{T[i+1..k]}$ storing the secondary position $i+1$. This can again be seen from the factorial property of the position heap (Theorem 1(ii)). This shows that the above two cases split the interval of secondary positions $[s..k]$ into two subintervals $[s..t-1]$ and $[t..k]$, such that node $\overline{T[i..k]}$ does not have an outgoing $T[k+1]$-edge for $i \in [s..t-1]$ and does have such an edge for $i \in [t..k]$.

The above discussion is summarized in the following lemma specifying the changes that have to be made to transform $PH(T[1..k])$ into $PH(T[1..k+1])$.

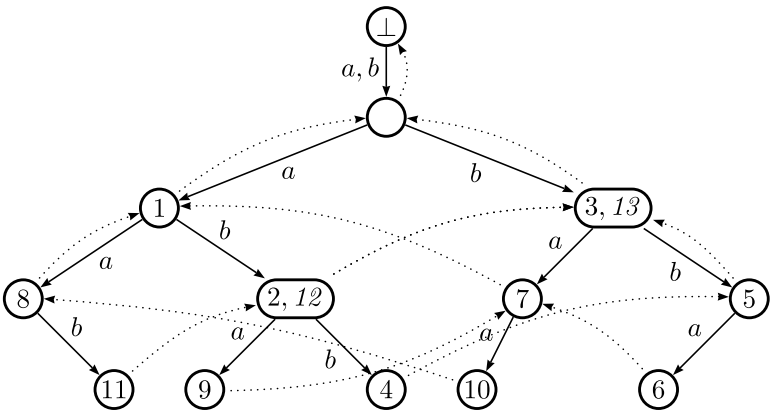



**Fig. 4.** Position heap for string *aababbbaabaab* with suffix pointers (dotted arrows). Secondary positions are shown in italic.

**Lemma 3.** *Given $T[1..n]$, consider $PH(T[1..k])$ for $k < n$. Let $s$ be the active secondary position, stored in the node $\overline{T[s..k]}$. Let $t \geqslant s$ be the smallest position such that node $\overline{T[t..k]}$ has an outgoing $T[k+1]$-transition. To obtain $PH(T[1..k+1])$, $PH(T[1..k])$ should be modified in the following way*:

(i) *for every node $\overline{T[i..k]}$, $s \leqslant i \leqslant t-1$, create a new child linked to $\overline{T[i..k]}$ by a $T[k+1]$-edge. Delete secondary position $i$ from the node $\overline{T[i..k]}$ and assign it as a primary position to the new node $\overline{T[i..k+1]}$,*
(ii) *for every node $\overline{T[i..k]}$, $i \geqslant t$, move the secondary position $i$ from node $\overline{T[i..k]}$ to node $\overline{T[i..k+1]}$.*

We describe now the algorithm implementing the changes specified by Lemma 3. We augment $PH(T)$ with *suffix pointers* $f$ defined in the usual way:

**Definition 1.** *For each node $\overline{T[i..j]}$ of $PH(T)$, a* suffix pointer *is defined by $f(\overline{T[i..j]}) = \overline{T[i+1..j]}$.*

Note that the definition is sound, as the node $\overline{T[i+1..j]}$ exists whenever the node $\overline{T[i..j]}$ exists, according to Theorem 1(ii). For the root node, it will be convenient for us to define $f(root) = \perp$, where $\perp$ is a special node such that there is an $a$-edge between $\perp$ and *root* for every $a \in A$ (similar to Ukkonen's algorithm [17]). Fig. 4 shows the position heap of Fig. 2 supplemented by suffix pointers.

We now begin to describe the on-line construction algorithm for $PH(T)$, given a text $T[1..n]$. Consider the node $\overline{T[s..k]}$ of $PH(T[1..k])$ storing the active secondary position $s$, that we call the *active node*. If the active secondary position does not exist (i.e. there is no secondary positions at all), then the active node is *root* and the active position is set to $k+1$. Observe that the nodes storing the other secondary positions $s+1, s+2, \ldots, n$ can be reached, in order, by following the chain of suffix pointers $f(\overline{T[s..n]})$, $f(f(\overline{T[s..n]}))$, ... until the root node is reached. On the example of Fig. 4, the active secondary position is *12*, and the chain of suffix pointer outgoing from the active node leads to the node storing position *13* followed by the root.

This brings us to the main trick of our construction: *we will not store secondary positions at all, but only memorize the active secondary position and the active node*. The secondary positions can be easily recovered by traversing the chain of suffix pointers starting from the active node and incrementing the position counter after traversing each edge. Note also that if the input string $T$ is ended by a unique sentinel symbol, the resulting position heap does not contain any secondary nodes and there is no need to recover them.

Keeping in mind that the secondary positions are not stored explicitly, the transformation of $PH(T[1..k])$ into $PH(T[1..k+1])$ specified by Lemma 3 reduces to processing case (i) only, as case (ii) does not imply any modification anymore. Case (i) is implemented by the following simple procedure. Starting from the active node, the algorithm traverses the chain of suffix pointers as long as the current node does not have an outgoing $T[k+1]$-edge. For each such node, a new node is created linked by a $T[k+1]$-edge to the current node. A suffix pointer to this new node is set from the previously created new node. Once the first node with an outgoing $T[k+1]$-edge is encountered, the algorithm moves to the node this edge leads to, sets the suffix pointer to this node, and assigns this node to be the active node for the following iteration. The correctness of the last assignment is stated in the following lemma.

**Lemma 4.** *Consider $PH(T[1..k])$ and let $s$ be the active position, and $t \geqslant s$ be the smallest position such that node $\overline{T[t..k]}$ has an outgoing $T[k+1]$-edge. Then node $\overline{T[t..k+1]}$ is the active node of $PH(T[1..k+1])$.*

**Proof.** As it follows from Lemma 3, $t$ is the largest secondary position of $T[1..k+1]$. □

Algorithm 1 provides a pseudo-code of the algorithm.

**Algorithm 1** On-line construction of the position heap $PH(T[1..n])$.

```
1: create states root and ⊥
2: f(root) ← ⊥
3: for all a ∈ A do
4:    set an a-edge from ⊥ to root
5: end for
6: currentnode ← root
7: currentsuffix ← 1
8: for i = 1 to n do
9:    lastcreatednode ← undefined
10:   while currentnode does not have an outgoing T[i]-edge do
11:      create a new node newnode pointing to currentsuffix
12:      set a T[i]-edge from currentnode to newnode
13:      if lastcreatednode ≠ undefined then
14:         f(lastcreatednode) ← newnode
15:      end if
16:      lastcreatednode ← newnode
17:      currentnode ← f(currentnode)
18:      currentsuffix ← currentsuffix + 1
19:   end while
20:   move currentnode to the target node of the outgoing T[i]-edge
21:   if lastcreatednode ≠ undefined then
22:      f(lastcreatednode) ← currentnode
23:   end if
24: end for
```

The correctness of Algorithm 1 follows from Lemmas 3, 4 and the discussion above. It is instructive, in addition, to observe the following:

- It is easily seen that the suffix pointers of $PH(T[1..k+1])$ are correctly set. Indeed, the algorithm assigns to $\overline{T[i..k+1]}$ a suffix pointer to $\overline{T[i+1..k+1]}$ which is obviously correct. Note that for the active position $s$ of $T[1..k]$, the created node $\overline{T[s..k+1]}$ does not get pointed to by any suffix pointer, which is correct, as $\overline{T[s-1..k+1]}$ is not represented in $PH(T[1..k+1])$: the position $s-1$ is primary in $T[1..k]$ and therefore the node $\overline{T[s-1..k]}$, if it exists in $PH(T[1..k])$, does not get extended by a $T[k+1]$-edge (cf. Lemma 3).
- Since the depth of $\overline{T[s..k]}$ ($s$ is the active position) in $PH(T[1..k])$ is $k+1-s$ and a traversal of a suffix link decrements the depth by 1 and increments the current position by 1, it follows that if the traversal of the suffix chain reaches the root node, the active position value becomes $k+1$, which is exactly what we need to start processing the next letter $T[k+1]$. This shows why Algorithm 1 correctly maintains *currentsuffix* and never needs to reset it at the beginning of the **for**-loop iteration.

It is easy to see that the running time of Algorithm 1 is linear in the length $n$ of the input string. Since each iteration of the **while**-loop creates a node, this loop iterates exactly $n$ times over the whole run of the algorithm. Trivially, the **for**-loop iterates $n$ times too, and all the involved operations are constant time. Thus, the whole algorithm takes $O(n)$ time. The following theorem concludes the construction.

**Theorem 2.** *For an input string $T[1..n]$, Algorithm 1 correctly constructs $PH(T)$ on-line in time $O(n)$.*

## 5. Augmented position heap

Assume we have a text $T[1..n]$ for which we constructed the position heap $PH(T)$. We don't assume that $T$ is ended by a unique letter, and therefore some nodes of $PH(T)$ are double nodes and store two positions of $T$, one primary and one secondary. Here we assume that the secondary positions *are* actually stored (or can be retrieved in constant time for each node). As explained in Section 4, even if the secondary positions are not stored during the construction of $PH(T)$, they can be easily recovered once the construction is completed.

Ehrenfeucht et al. [8] proposed a linear-time string matching algorithm using $PH(T[1..n])$, i.e. an algorithm that computes all occurrences of a pattern string in $T$ in time $O(m+occ)$, where $m$ is the pattern length and $occ$ the number of occurrences. Describing this elegant algorithm is beyond the scope of this paper, we refer the reader to [8] for its description. We only note that the algorithm itself applies without changes to our definition of position heap, as it does not depend in any way on the order that the suffixes of $T$ are inserted.

However, the algorithm of [8] runs on $PH(T)$ enriched with some additional information. Let $\dot{i}$ denote the node of $PH(T)$ storing position $i$, $1 \leqslant i \leqslant n$. The extended data structure, called the *augmented position heap*, should allow the following queries to be answered in constant time:

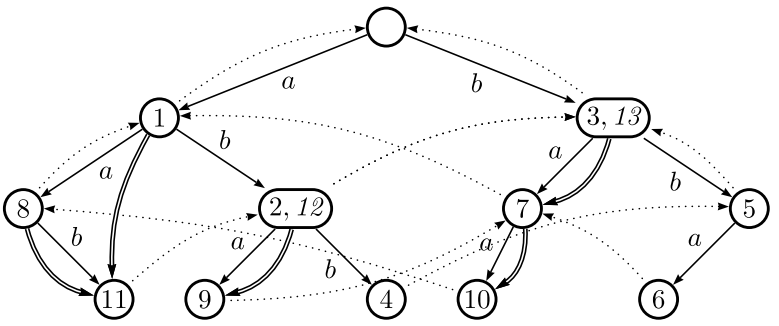



**Fig. 5.** Position heap for string *aababbbaabaab* with suffix pointers and maximal-reach pointers *mrp* (double arrows). Only values for which $mrp(i) \neq \dot{i}$ are shown, namely $mrp(1) = \overline{11}$, $mrp(8) = \overline{11}$, $mrp(2) = \overline{9}$, $mrp(3) = \overline{7}$, $mrp(7) = \overline{10}$. Note that maximal reach pointers outgoing from double nodes are unambiguous as for all secondary positions $i$, we have $mrp(i) = \dot{i}$.

- given a position $i$, retrieve the node $\dot{i}$,
- given two nodes $\dot{i}$ and $\dot{j}$, is $\dot{i}$ a (not necessarily immediate) ancestor of $\dot{j}$?
- given a position $i$ of $T$, retrieve the node $\overline{T[i..i+\ell]}$, where $T[i..i+\ell]$ is the longest substring of $T$ starting at position $i$ and represented in $PH(T)$.

To answer the first query, [8] simply introduces an auxiliary array storing, for each position $i$, a pointer to the node $\dot{i}$. Maintaining this array during the construction of $PH(T)$ by Algorithm 1 is trivial: once a position is assigned to a newly created node (line 11 of Algorithm 1), a new entry of the array is set. If $T$ is not ended by a unique symbol and then the final $PH(T)$ has secondary positions, those are easily recovered by traversing the chain of suffix pointers at the very end of the construction.

The second query can be also easily answered in constant time after a linear-time preprocessing of $PH(T)$. A solution proposed in [8] consists in traversing $PH(T)$ depth-first and storing, for each node, its discovery and finishing times [4]. Then node $\dot{i}$ is an ancestor of node $\dot{j}$ if and only if the discovery and finishing time of $\dot{i}$ is respectively smaller and greater than the discovery and finishing time of $\dot{j}$.

A more space-efficient solution would be to use a balanced parenthesis representation of the tree topology of $PH(T)$, taking $2n$ bits, and link each node to the corresponding opening parenthesis. Then the corresponding closing parenthesis can be retrieved in constant time by the method of [13] using $o(n)$ auxiliary bits. This allows ancestor queries to be answered in constant time.

The third type of queries is answered by an additional mapping called *maximal-reach pointer* [8]: for a position $i$ of $T[1..n]$, define $mrp(i)$ to be the node $\overline{T[i..i+\ell]}$, where $T[i..i+\ell]$ is the longest prefix of $T[i..n]$ represented in $PH(T)$. Observe first that if $i$ is a secondary position, then $mrp(i) = \dot{i}$. This is because a secondary position $i$ is stored in node $\overline{T[i..n]}$, which trivially corresponds to the longest prefix starting at $i$. Therefore, as it is done in [8], *mrp* can be represented by pointers from node $\dot{i}$ to node $mrp(i)$ whenever these nodes are different. In our case, we have then to keep in mind that a maximal-reach pointer from a double node applies to the primary position of this node. Fig. 5 provides an illustration.

In [8], maximal-reach pointers are computed by an extra traversal of $PH(T)$, using an auxiliary *dual heap* structure on top of it (see Introduction). Here we show that maximal-reach pointers can be easily computed using suffix pointers instead of the dual heap. Thus, we completely get rid of the dual heap for constructing the augmented position heap, replacing it with suffix pointers.

After $PH(T)$ is constructed, we compute $mrp(i)$ iteratively for $i = 1, 2, \ldots, s-1$, where $s$ is the active secondary position of $T[1..n]$. Assume we have computed $mrp(i)$ for some $i$ and have to compute $mrp(i+1)$. Assume $mrp(i) = \overline{T[i..i+\ell]}$. It is easily seen that $T[i+1..i+\ell]$ is a prefix of the string represented by $mrp(i+1)$. To compute $mrp(k+1)$, we follow the suffix link $f(mrp(k))$ to reach $\overline{T[i+1..i+\ell]}$ and then keep extending the prefix $T[i+1..i+\ell]$ as long as it is represented in $PH(T)$. The resulting pseudo-code is given in Algorithm 2.

**Algorithm 2** Linear-time computation of maximal-reach pointers $mrp(i)$.

```
1: currentnode ← root
2: readhead ← 1
3: for i = 1 to n do
4:    while currentnode has an outgoing T[readhead]-edge and readhead ⩽ n do
5:       move currentnode to the target node of the outgoing T[readhead]-edge
6:       readhead ← readhead + 1
7:    end while
8:    mrp(i) ← currentnode
9:    currentnode ← f(currentnode)
10: end for
```

It is easy to see that Algorithm 2 works in time $O(n)$: the **while**-loop makes exactly $n$ iterations overall, as each iteration increments the *readhead* counter.

The following property of Algorithm 2 is useful to observe: as soon as *readhead* gets the value $n+1$ (line 6), the node *currentnode* gets assigned to the active node of $PH(T[1..n])$ (line 9); at the subsequent iterations, the algorithm simply traverses the chain of suffix links and sets the maximal-reach pointer for each secondary position to be the node storing this position (lines 8–9).

Maximal-reach pointers constitute an additional data structure on top of the tree structure of the position heap. However, it is interesting to note that this structure can be represented compactly in $O(n)$ bits so that $mrp(i)$ can be computed in constant time. Here is how it can be done.

As observed earlier, $mrp(i) \geqslant mrp(i-1)-1$ for all $i \in [2..n]$. Define $\delta_i = mrp(i) - mrp(i-1) + 1$ and observe that $mrp(1) + \sum_{i=2}^{n} \delta_i = n$. Represent the vector $(mrp(1), \delta_2, \ldots, \delta_n)$ as a binary vector $\mathbb{B}_{mrp}$ by representing all values in unary followed by a 0. For example, vector $(1,2,0,3,0,1,0)$ is then represented as 10110011100100. Note that the length of $\mathbb{B}_{mrp}$ is $2n$. To $\mathbb{B}_{mrp}$, we will be applying rank and select operations. Recall that for a binary vector $\mathbb{B}$, $\mathsf{rank}_1(\mathbb{B}, i)$ returns the number of 1's occurring in $\mathbb{B}[1..i]$, and $\mathsf{select}_1(\mathbb{B}, \ell)$ returns the position of the $\ell$-th occurrence of 1 in $\mathbb{B}$ (counting from left). $\mathsf{rank}_0$ and $\mathsf{select}_0$ are defined similarly. It is known that the input binary vector of length $n$ can be pre-processed using $o(n)$ additional memory bits, so that rank and select queries can be answered in time $O(1)$ [12,2]. Observe now that $mrp(i) = \mathsf{rank}_1(\mathsf{select}_0(i)) - i + 1$. Therefore, $mrp(i)$ can be computed in constant time. We summarize this in the following Lemma.

**Lemma 5.** *For the position heap $PH(T)$ of any text $T[1..n]$, the maximal-reach pointers $mrp(i)$, $i \in [1..n]$, can be stored in $2n + o(n)$ bits so that each $mrp(i)$ can be recovered in time $O(1)$.*

## 6. Concluding remarks

We proposed a construction algorithm of a position heap of a string, under a modified definition of position heap compared to [8]. In contrast with the algorithm of [8] that processes the sequence right-to-left, our algorithm reads the string left-to-right and has the on-line property. Drawing a parallel to suffix trees, our algorithm can be compared to Ukkonen's on-line algorithm [17], while the algorithm of [8] can be compared to Weiner's algorithm [18]. The similarity of our algorithm to Ukkonen's algorithm goes beyond this parallel, as the execution of both algorithms (e.g. the way of traversing the tree under construction, or updating the active node) are clearly analogous.

Position heap is a smaller data structure than suffix tree: it contains exactly $n+1$ nodes whereas the suffix tree has $n$ leaves and then up to $2n$ nodes. Still, the position heap allows for a linear-time string matching. The position heap is a new data structure and many questions about it are open.

The $O(n)$ complexity bounds of both Algorithm 1 (Theorem 2) and Algorithm 2 are stated for a constant-size alphabet, otherwise a correcting factor $\log|A|$ should be introduced, similarly to the suffix tree construction. One may ask if there exists a linear-time algorithm (not necessarily on-line) which constructs position heaps within a time independent of the alphabet size, as Farach's algorithm does for suffix trees [9].

An interesting direction to study is whether the position heap can be compacted. The theory of compact data structures has become a major subfield of string processing, and has accumulated a number of interesting and powerful techniques [15]. In this paper, we showed that some components of the position heap can be effectively compacted, however the compaction of its main part (the trie) is still to be studied.

It would be interesting to study combinatorial properties of position heaps. In combinatorial terminology, a tree with $n$ nodes labeled by distinct integers from $\{1,\ldots,n\}$ and such that the label of any node is smaller than the label of any of its descendant is called an *increasing tree*. It is known, for example, that there are $n!$ ordered binary increasing trees [16] ("ordered" means that left and right children are distinguished), which implies that there are $(n+1)!$ "potential position heaps" over binary alphabet. Obviously, there are only $2^n$ different position heaps over the binary alphabet. It would be interesting to establish combinatorial properties that distinguish arbitrary increasing trees from position heaps.

The authors of [8] proposed algorithms for updating the position heap when the input string undergoes modifications (character insertions/deletions). We believe that these algorithms can be easily applied to our definition of position heap. Recently, the authors of [14] showed that the position heap can be generalized to a set of strings stored in a trie such that the construction and pattern matching remain linear-time. Other interesting applications of position heap are still to be discovered.

From a more practical perspective, it would be also interesting to exploit the "adaptiveness" of position heaps to substring frequencies, mentioned in Section 3.

## References

[1] A. Blumer, J. Blumer, D. Haussler, A. Ehrenfeucht, M.T. Chen, J. Seiferas, The smallest automaton recognizing the subwords of a text, Theoretical Computer Science 40 (1985) 31–55.

[2] D. Clark, J. Munro, Efficient suffix trees on secondary storage (extended abstract), in: E. Tardos (Ed.), Proceedings of the Seventh Annual ACM–SIAM Symposium on Discrete Algorithms (SODA), Atlanta, Georgia, 28–30 January 1996, ACM/SIAM, 1996, pp. 383–391.

[3] E. Coffman, J. Eve, File structures using hash functions, Communications of the ACM 13 (1970) 427–432.
[4] T. Cormen, C. Leiserson, R. Rivest, Introduction to Algorithms, MIT Press, 1999.
[5] M. Crochemore, Transducers and repetitions, Theoretical Computer Science 45 (1986) 63–86.
[6] M. Crochemore, String matching with constraints, in: Proceedings of the International Symposium on Mathematical Foundations of Computer Science, in: Lecture Notes in Computer Science, vol. 324, Springer-Verlag, 1988, pp. 44–58.
[7] M. Crochemore, W. Rytter, Text Algorithms, Oxford University Press, 1994.
[8] A. Ehrenfeucht, R. McConnell, N. Osheim, S.-W. Woo, Position heaps: A simple and dynamic text indexing data structure, Journal of Discrete Algorithms 9 (1) (2011) 100–121, preliminary version in: Proc. 20th Anniversary Edition of the Annual Symposium on Combinatorial Pattern Matching (CPM 2009).
[9] M. Farach, Optimal suffix tree construction with large alphabets, in: Proc. 38th Annual Symposium on Foundations of Computer Science, FOCS'97, Miami Beach, Florida, USA, October 19–22, 1997, IEEE Computer Society, 1997, pp. 137–143.
[10] E. Fredkin, Trie memory, Communications of the ACM 3 (9) (1960) 490–499.
[11] D. Gusfield, Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology, Cambridge University Press, ISBN 0-521-58519-8, 1997.
[12] G. Jacobson, Space-efficient static trees and graphs, in: Proc. 30th Annual Symposium on Foundations of Computer Science (FOCS), Research Triangle Park, North Carolina, USA, 30 October–1 November 1989, IEEE Computer Society, 1989, pp. 549–554.
[13] J. Munro, V. Raman, Succinct representation of balanced parentheses and static trees, SIAM J. Comput. 31 (3) (2001) 762–776.
[14] Y. Nakashima, T. I, S. Inenaga, H. Bannai, M. Takeda, The position heap of a trie, in: Proc. of the 19th Symposium on String Processing and Information Retrieval (SPIRE'12), Cartagena, Colombia, October 21–25, 2012, in: Lecture Notes in Computer Science, Springer-Verlag, 2012, in press.
[15] G. Navarro, V. Mäkinen, Compressed full-text indexes, ACM Comput. Surv. 39 (1) (2007), http://dx.doi.org/10.1145/1216370.1216372.
[16] R. Stanley, Enumerative Combinatorics, Cambridge Studies in Advanced Mathematics, vol. 1, Cambridge University Press, 1999.
[17] E. Ukkonen, On-line construction of suffix-trees, Algorithmica 14 (3) (1995) 249–260.
[18] P. Weiner, Linear pattern matching algorithm, in: 14th Annual IEEE Symposium on Switching and Automata Theory, 1973, pp. 1–11.