We propose a new indexing structure for parameterized strings, called parameterized position heap. Parameterized position heap is applicable for parameterized pattern matching problem, where the pattern matches a substring of the text if there exists a bijective mapping from the symbols of the pattern to the symbols of the substring. We propose an online construction algorithm of parameterized position heap of a text and show that our algorithm runs in linear time with respect to the text size. We also show that by using parameterized position heap, we can find all occurrences of a pattern in the text in linear time with respect to the product of the pattern size and the alphabet size.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases string matching, indexing structure, parameterized pattern matching, position heap

Digital Object Identifier 10.4230/LIPIcs.CPM.2017.8

1 Introduction

String matching problem is to find occurrences of a pattern string in a text string. Formally, given a text string $t$ and a pattern string $p$ over an alphabet $\Sigma$, output all positions at which $p$ occurs in $t$. Suffix tree and suffix array are most widely used data structures and provide many applications for various string matchings (see e.g. [11, 6]).

Ehrenfeucht et al. [8] proposed an indexing structure for string matching, called a position heap. Position heap uses less memory than suffix tree does, and provides efficient search of patterns by preprocessing the text string, similarly to suffix tree and suffix array. A position heap for a string $t$ is a sequence hash tree [4] for the ordered set of all suffixes of $t$. In [8], the suffixes are ordered in the ascending order of length, and the proposed construction algorithm processes the text from right to left. Later, Kucherov [13] considered the ordered set of suffixes in the descending order of length and proposed a linear-time
online construction algorithm based on the Ukkonen’s algorithm [16]. Nakashima et al. [14] proposed an algorithm to construct a position heap for a set of strings, where the input is given as a trie of the set. Gagie et al. [10] proposed a position heap with limited height and showed some relations between position heap and suffix array.

The parameterized pattern matching that focuses on a structure of strings is introduced by Baker [2]. Let $\Sigma$ and $\Pi$ be two disjoint sets of symbols. A string over $\Sigma \cup \Pi$ is called a parameterized string (p-string for short). In the parameterized pattern matching problem, given p-strings $t$ and $p$, find positions of substrings of $t$ that can be transformed into $p$ by applying one-to-one function that renames symbols in $\Pi$. The parameterized pattern matching is motivated by applying to the software maintenance [1, 2, 3], the plagiarism detection [9], the analysis of gene structure [15], and so on. Similar to the basic string matching problem, some indexing structures that support the parameterized pattern matching are proposed, such as parameterized suffix tree [2], structural suffix tree [15], and parameterized suffix array [7, 12].

In this paper, we propose a new indexing structure called parameterized position heap for the parameterized pattern matching. The parameterized position heap is a sequence hash tree for the ordered set of prev-encoded [2] suffixes of a parameterized string. We give an online construction algorithm of a parameterized position heap based on Kucherov’s algorithm [13] that runs in $O(n \log (|\Sigma| + |\Pi|))$ time and an algorithm that runs in $O(m \log (|\Sigma| + |\Pi|) + m|\Pi| + \text{occ})$ time to find the occurrences of a pattern in the text, where $n$ is the length of the text, $m$ is the length of the pattern, $|\Sigma|$ is the number of constant symbols, $|\Sigma|$ is the number of parameter symbols, and $\text{occ}$ is the number of occurrences of the pattern in the text.

## 2 Notation

Let $\Sigma$ and $\Pi$ be two disjoint sets of symbols. $\Sigma$ is a set of constant symbols and $\Pi$ is a set of parameter symbols. An element of $\Sigma^*$ is called a string, and an element of $(\Sigma \cup \Pi)^*$ is called a parameterized string, or p-string for short. For a p-string $w = xyz$, $x$, $y$, and $z$ are called prefix, substring, and suffix of $w$, respectively. $|w|$ denotes the length of $w$, and $w[i]$ denotes the $i$-th symbol of $w$ for $1 \leq i \leq |w|$. The substring of $w$ that begins at position $i$ and ends at position $j$ is denoted by $w[i : j]$ for $1 \leq i \leq j \leq |w|$. Moreover, let $w[i : i] = w[1 : i]$ and $w[i : j] = w[i : |w|]$ for $1 \leq i \leq |w|$. The empty p-string is denoted by $\varepsilon$, that is $|\varepsilon| = 0$. For convenience, let $w[i : j]$ if $i > j$. Let $\mathcal{N}$ denote the set of all non-negative integers.

Given two p-strings $w_1$ and $w_2$, $w_1$ and $w_2$ are a parameterized match or p-match, denoted by $w_1 \approx w_2$, if there exists a bijection $f$ from the symbols of $w_1$ to the symbols of $w_2$, such that $f$ is identity on the constant symbols [2]. We can determine whether $w_1 \approx w_2$ or not by using an encoding called prev-encoding defined as follows.

**Definition 1 (Prev-encoding [2]).** For a p-string $w$ over $\Sigma \cup \Pi$, the prev-encoding for $w$, denoted by $\text{prev}(w)$, is a string $x$ of length $|w|$ over $\Sigma \cup \mathcal{N}$ defined by

$$x[i] = \begin{cases} w[i] & \text{if } w[i] \in \Sigma, \\ 0 & \text{if } w[i] \in \Pi \text{ and } w[i] \neq w[j] \text{ for } 1 \leq j < i, \\ i - \max\{j \mid w[j] = w[i] \text{ and } 1 \leq j < i\} & \text{otherwise.} \end{cases}$$

For any p-strings $w_1$ and $w_2$, $w_1 \approx w_2$ if and only if $\text{prev}(w_1) = \text{prev}(w_2)$. For example, given $\Sigma = \{a, b\}$ and $\Pi = \{u, v, x, y\}$, $s_1 = uvuvaarb$ and $s_2 = xyxyaxxyb$ are p-matches where $\text{prev}(s_1) = \text{prev}(s_2) = 0022a314b$.

The parameterized pattern matching is a problem to find occurrences of a p-string pattern in a p-string text defined as follows.
Figure 1 (a) A sequence hash tree for (aab, ab, bba, baa, aaba, baba). (b) A position heap for a string abbaabaabaab. (c) An augmented position heap for a string abbaabaabaab. Maximal-reach pointers for $mrp(i) \neq i$ are illustrated by doublet arrows.

Definition 2 (Parameterized pattern matching [2]). Given two $p$-strings, text $t$ and pattern $p$, find all positions $i$ in $t$ such that $t[i : i + |p| - 1] \approx p$.

For example, let us consider a text $t = urabaarb$ and a pattern $p = xayby$ over $\Sigma = \{a, b\}$ and $\Pi = \{u, v, x, y\}$. Because $p \approx t[2 : 6]$ and $p \approx t[6 : 10]$, we should output 2 and 6.

Throughout this paper, let $t$ be a text of length $n$ and $p$ be a pattern of length $m$.

## 3 Position Heap

In this section, we briefly review the position heap for strings. First we introduce the sequence hash tree that is a trie for hashing proposed by Coffman and Eve [4]. Each edge of the trie is labeled by a symbol and each node can be identified with the string obtained by concatenating all labels found on the path from root to the node.

Definition 3 (Sequence Hash Tree). Let $W = (w_1, \ldots, w_n)$ be an ordered set of strings over $\Sigma$ and $W_i = (w_1, \ldots, w_i)$ for $1 \leq i \leq n$. A sequence hash tree $SHT(W) = (V_n, E_n)$ for $W$ is a trie over $\Sigma$ defined recursively as follows. Let $SHT(W_i) = (V_i, E_i)$. Then,

$SHT(W_i) = \begin{cases} \{\{\varepsilon\}, \emptyset\} & \text{(if } i = 0\text{)} , \\ \{V_{i-1} \cup \{p_i\}, E_{i-1} \cup \{(q_i, c, p_i)\}\} & \text{(if } 1 \leq i \leq n\text{)} . \end{cases}$

where $p_i$ is the shortest prefix of $w_i$ such that $p_i \not\in V_{i-1}$, and $q_i = w_i[1 : |p_i| - 1]$, $c = w_i[|p_i|]$. If no such $p_i$ exists, then $V_i = V_{i-1}$ and $E_i = E_{i-1}$.

Each node in a sequence hash tree stores one or several indices of strings in the input set. An example of a sequence hash tree is shown in Figure 1 (a).

The position heap proposed by Ehrenfeucht et al. [8] is a sequence hash tree for the ordered set of all suffixes of a string. Two types of position heap are known. The first one is proposed by Ehrenfeucht et al. [8], that constructed by the ordered set of suffixes in ascending order of length and the second one is proposed by Kucherov [13], which constructed in descending order. We adopt the Kucherov [13] type and his online construction algorithm for constructing position heaps for parameterized strings in Section 4. Here we recall the definition of the position heap by Kucherov.

Definition 4 (Position Heap [13]). Given a string $t \in \Sigma^*$, let $S_t = (t[1:], t[2:], \ldots, t[n:])$ be the ordered set of all suffixes of $t$ except $\varepsilon$ in descending order of length. The position heap $PH(t)$ for $t$ is $SHT(S_t)$. 

\[\text{CPM 2017}\]
Position Heaps for Parameterized Strings

Each node except the root in a position heap stores either one or two integers those are beginning positions of corresponding suffixes. We call them regular node and double node respectively. Assume that \( i \) and \( j \) are positions stored by a double node \( v \) in \( PH(t) \) where \( i < j \), \( i \) and \( j \) are called the primary position and the secondary position respectively. Figure 1 (b) shows an example of a position heap.

In order to find occurrences of the pattern in \( O(m + \text{occ}) \) time, Ehrenfeucht et al. [8] and Kucherov [13] added additional pointer called maximal-reach pointer to the position heap and called this extended data structure as augmented position heap. An example of an augmented position heap is showed in Figure 1 (c).

4 Parameterized Position Heap

In this section, we propose a new indexing structure called parameterized position heap. It is based on the position heap proposed by Kucherov [13].

4.1 Definition and Property of Parameterized Position Heap

The parameterized position heap is a sequence hash tree [4] for the ordered set of prev-encoded suffixes in the descending order of length.

Definition 5. (Parameterized Position Heap). Given a p-string \( t \in (\Sigma \cup \Pi)^n \), let \( S_t = \{\text{prev}(t[1:]), \text{prev}(t[2:]), \ldots, \text{prev}(t[n:])\} \) be the ordered set of all prev-encoded suffixes of the p-string \( t \) except \( \varepsilon \) in descending order of length. The parameterized position heap \( PPH(t) \) for \( t \) is \( SHT(S_t) \).

Figure 2 (a) shows an example of a parameterized position heap. A parameterized position heap \( PPH(t) \) for a p-string \( t \) of length \( n \) consists of the root and nodes that corresponds to \( \text{prev}(t[1:]) \), \( \text{prev}(t[2:]) \), \ldots, \( \text{prev}(t[n:]) \), so \( PPH(t) \) has at most \( n + 1 \) nodes. Each node in \( PPH(t) \) holds either one or two of beginning positions of corresponding p-suffixes similar to the standard position heaps. We can specify each node in \( PPH(t) \) by its primary position, its secondary position, or the string obtained by concatenating labels found on the path from the root to the node.

Different from standard position heap, \( \text{prev}(t[i:]) = \text{prev}(t)[i:] \) does not necessarily hold for some cases. For example, for \( t = xxxyxxyyaxyx, \) \( \text{prev}([3:]) = 0022221a4322 \) while \( \text{prev}([3:]) = 022221a4322 \). Therefore, the construction and matching algorithms for the
standard position heaps cannot be directly applied for the parameterized position heaps. However, we can similar properties to construct parameterized position heaps efficiently.

**Lemma 6.** For $i$ and $j$, where $1 \leq i \leq j \leq n$, if $\text{prev}(t[i:j])$ is represented in $PPH(t)$, then a prev-encoded string for any substring of $t[i:j]$ is also represented in $PPH(t)$.

**Proof.** First we will show that prev-encoding of any prefix of $t[i:j]$ is represented in $PPH(t)$. From the definition of prev-encoding, $\text{prev}(t[i:j])[1:i-j] = \text{prev}(t[i:j-1])$. In other words, $\text{prev}(t[i:j-1])$ is a prefix of $\text{prev}(t[i:j])$. From the definition of $PPH(t)$, prefixes of $\text{prev}(t[i:j])$ are represented in $PPH(t)$. Therefore, $\text{prev}(t[i:j-1])$ is represented in $PPH(t)$. Similarly, $\text{prev}(t[i:j-2]), \ldots, \text{prev}(t[i:i])$ are represented in $PPH(t)$.

Next, we will show that prev-encoding of any suffix of $t[i:j]$ is represented in $PPH(t)$. From the above discussion, there are positions $b_0 < b_1 < \cdots < b_{j-i} = i$ in $t$ such that $\text{prev}(t[b_k : b_{k+1}]) = \text{prev}(t[i : i+k])$. From the definition of parameterized position heap, $\text{prev}(t[b_k + 1 : b_{k+1}])$ is represented in $PPH(t)$. Since $\text{prev}(t[b_k + 1 : b_{k+1}])$ is a prefix of $PPH(t)$ then $\text{prev}(t[b_k + 1 : b_{k+1} + k + 1])$ is also represented in $PPH(t)$ recursively. Therefore, $\text{prev}(t[b_{j-i} + 1 : b_j - i]) = \text{prev}(t[i + 1 : j])$ is represented in $PPH(t)$. Similarly, $\text{prev}(t[i+2:j]), \ldots, \text{prev}(t[j:j])$ are represented in $PPH(t)$.

Since any prefix and suffix of $\text{prev}(t[i:j])$ is represented in $PPH(t)$, we can say that any substring of $\text{prev}(t[i:j])$ is represented in $PPH(t)$ by induction.

**4.2 Online Construction Algorithm of Parameterized Position Heap**

In this section, we propose an online algorithm that constructs parameterized position heaps. Our algorithm is based on Kucherov’s algorithm, although it cannot be applied easily. The algorithm updates $PH(t[1:k])$ to $PH(t[1:k+1])$ when $t[k+1]$ is read, where $1 \leq k \leq n-1$. Updating of the position heap begins from a special node, called the active node. A position specified by the active node is called the active position. At first, we show that there exists a position similar to the active position in the parameterized position heap.

**Lemma 7.** If $j$ is a secondary position of a double node in a parameterized position heap, then $j+1$ is also a secondary position.

**Proof.** Let $i$ be the primary position and $j$ be the secondary position of node $v$, where $i < j$. This means there is a position $h$ such that $\text{prev}(t[i : h]) = \text{prev}(t[j : i])$. By Lemma 6, there is a node that represents $\text{prev}(t[i+1:h])$. Since $\text{prev}(t[j+1 : i]) = \text{prev}(t[l+1 : h])$, then $j+1$ will be the secondary positions of node $\text{prev}(t[i+1 : h])$.

Lemma 7 means that there exists a position $s$ which splits all positions in $t[1:n]$ into two intervals, similar to the active position in [13]. Positions in $[1:s-1]$ and $[s:n]$ are called primary and secondary positions, respectively. We also call the position $s$ as active position.

Assume we have constructed $PPH(t[1:k])$ and we want to construct $PPH(t[1:k+1])$ from $PPH(t[1:k])$. The primary positions $1, \ldots, s-1$ in $PPH(t[1:k])$ become primary positions also in $PPH(t[1:k+1])$, because $\text{prev}(t[i : k]) = \text{prev}(t[i : k+1])[1 : k-1+1]$ holds for $1 \leq i \leq s-1$. Therefore, we do not need to update the primary positions.

On the other hand, the secondary positions $s, \ldots, k$ require some modifications. When inserting a new symbol, two cases can occur. The first case is that $\text{prev}(t[i : k+1])$ is not represented in $PPH(t[1:k])$. In this case, a new node $\text{prev}(t[i : k+1])$ is created as a child node of $\text{prev}(t[i : k])$ and position $i$ becomes the primary position of the new node. The second case is that $\text{prev}(t[i : k+1])$ was already represented in $PPH(t[1:k])$. In this case,
the secondary position $i$ that is stored in $\text{prev}(t[i : k])$ currently should be moved to the child
node $\text{prev}(t[i : k + 1])$, and position $i$ becomes the secondary position of this node.

From Lemma 6, if the node $\text{prev}(t[i : k])$ has an edge to the node $\text{prev}(t[i : k + 1])$, $\text{prev}(t[i + 1 : k])$ also has an edge to $\text{prev}(t[i + 1 : k + 1])$. Therefore, there exists $r$, with $1 \leq s \leq r \leq k$, that splits the interval $[s : k]$ into two subintervals $[s : r - 1]$ and $[r : k]$, such that the node $\text{prev}(t[i : k])$ does not have an edge to $\text{prev}(t[i : k + 1])$ for $s \leq i \leq r - 1$, and does have such an edge for $r \leq i \leq k$.

The above analysis leads to the following lemma that specifies the modifications from
$\text{PPH}(t[1 : k])$ to $\text{PPH}(t[1 : k + 1])$.

$\blacktriangleright$ Lemma 8. Given $t \in (\Sigma \cup \Pi)^n$, consider $\text{PPH}(t[1 : k])$ for $k < n$. Let $s$ be the active
position, stored in the node $\text{prev}(t[s : k])$. Let $r \geq s$ be the smallest position such that node
$\text{prev}(t[r : k])$ has an outgoing edge labeled with $\text{prev}(t[r : k + 1][k - r + 2])$. $\text{PPH}(t[1 : k + 1])$
can be obtained by modifying $\text{PPH}(t[1 : k])$ in the following way:
1. For each node $\text{prev}(t[i : k])$, $s \leq i < r$, create a new child $\text{prev}(t[i : k + 1])$ linked by an
edge labeled $\text{prev}(t[i : k + 1][k - i + 2])$. Delete the secondary position $i$ from the node
$\text{prev}(t[i : k])$ and assign it as the primary position of the new node $\text{prev}(t[i : k + 1])$.
2. For each node $\text{prev}(t[i : k])$, $r \leq i \leq k$, move the secondary position $i$ from the node
$\text{prev}(t[i : k])$ to the node $\text{prev}(t[i : k + 1])$.

Moreover, $r$ will be the active position in $\text{PPH}(t[1 : k + 1])$.

Proof. Consider the first case that $i$ be a secondary position in $\text{PPH}(t[1 : k])$ and $s \leq i < r$.
From the definition of $r$, there is no node $\text{prev}(t[i : k + 1])$ in $\text{PPH}(t[1 : k])$. Therefore, $i$ will be a primary position of the node $\text{prev}(t[i : k + 1])$ in $\text{PPH}(t[1 : k + 1])$. We can update the
position heap from $\text{PPH}(t[1 : k])$ to $\text{PPH}(t[1 : k + 1])$ by delete $i$ from secondary position
of the node $\text{prev}(t[i : k])$ and create a new node $\text{prev}(t[i : k + 1])$ and assign $i$ to its primary
position for the case $s \leq i < r$.

Next case, $i$ be a secondary position in $\text{PPH}(t[1 : k])$ and $r \leq i \leq k$. In this case, there is a node $\text{prev}(t[i : k + 1])$ in $\text{PPH}(t[1 : k])$ and the node $\text{prev}(t[i : k + 1])$ is also represented in
$\text{PPH}(t[i : k + 1])$. Therefore, $i$ will be a secondary position of the node $\text{prev}(t[i : k + 1])$ in

![Figure 3](image-url) An example of updating a parameterized position heap, from (a) $\text{PPH}(aaxyyyxyz)$ to
(b) $\text{PPH}(aaxyyyzyxx)$. The updated positions are colored red. The secondary positions 6 and 7 in
$\text{PPH}(aaxyyyxyz)$ are become primary positions in $\text{PPH}(aaxyyyzyxx)$, while the secondary position
8 in $\text{PPH}(aaxyyyxyz)$ is become a secondary position of another node in $\text{PPH}(aaxyyyzyxx)$. The
active position is updated from 6 to 8.
We can update the position heap from $PPH(t[1 : k])$ to $PPH(t[1 : k + 1])$ by delete $i$ from secondary position of the node $prev(t[i : k])$ and assign $i$ as secondary position of the node $prev(t[i : k + 1])$ for the case $r ≤ i ≤ k$.

Since position $i$ for $1 ≤ i < r$ be a primary position in $PPH(t[1 : k + 1])$ and position $i$ for $r ≤ i ≤ k + 1$ be a secondary position in $PPH(t[1 : k + 1])$, $r$ will be the active position in $PPH(t[1 : k + 1])$.

Figure 3 show an example of updating a parameterized position heap. The modifications specified by Lemma 8 need to be applied to all secondary positions. In order to perform these modifications efficiently, we use parameterized suffix pointers.

**Definition 9 (Parameterized Suffix Pointer).** For each node $prev(t[i : j])$ of $PPH(t)$, the parameterized suffix pointer of $prev(t[i : j])$ is defined by $psp(prev(t[i : j])) = prev(t[i + 1 : j])$.

By Lemma 6, whenever the node $prev(t[i : j])$ exists, the node $prev(t[i + 1 : j])$ exists too. This means that $psp(prev(t[i : j]))$ always exists. During the construction of the parameterized position heap, let $\perp$ be the auxiliary node that works as the parent of root and is connected to root with an edge labeled with any symbol $c ∈ \Sigma \cup \emptyset$. We define $psp(root) = \perp$.

When $s$ is the active position in $PPH(t[1 : k])$, we call $prev(t[s : k])$ the active node. If no node holds a secondary position, root becomes the active node and the active position is set to $k + 1$. The nodes for the secondary positions $s, s + 1, \ldots, k$ can be visited by traversing with the suffix pointers from the active node. Thus, the algorithm only has to memorize the active position and the active node in order to visit any other secondary positions.

Updating $PPH(t[1 : k])$ to $PPH(t[1 : k + 1])$ specified by Lemma 8 is processed as the following procedures. The algorithm traverses with the suffix pointers from the active node till the node that has the outgoing edge labeled with $prev(t[i : k + 1])[k − i + 2]$ is found, which is $i = r$. For each traversed node, a new node is created and linked by an edge labeled with $prev(t[i : k + 1])[k − i + 2]$ to each node. A suffix pointer to this new node is set from the previously created node. When the node that has the outgoing edge labeled with $prev(t[i : k + 1])[k − i + 2]$ is traversed, the algorithm moves to the node that is led to by this edge, and a suffix pointer to this node is set from the last created node, then the algorithm assigns this node to be the active node.

A pseudocode of our proposed construction algorithm is given as Algorithm 1. $prim(v)$ and $sec(v)$ denotes primary and secondary positions of $v$, respectively. From the property of pre-encoding, $prev(t[i + 1 : k + 1])[k − i + 1] = prev(t[i : k + 1])[k − i + 2]$ if $prev(t[i : k + 1])[k − i + 2] ∈ \Sigma$ or $prev(t[i : k + 1])[k − i + 2] ≤ k − i$ and $prev(t[i + 1 : k])[k − i + 1] = 0$ otherwise. Therefore, we use a function $normalize(c, j)$ that returns $c$ if $c ∈ \Sigma$ or $c ≤ j$ and returns 0 otherwise.

The construction algorithm consists of $n$ iterations. In the $i$-th iteration, the algorithm reads $t[i]$ and make $PPH(t[1 : i])$. In the $i$-th iteration, the traversal of the suffix pointers as explained above is done. Since the depth of the current node decreases by traversing a suffix pointer, the number of the nodes that can be visited by traversal is $O(n)$. For each traversed node, all the operations such as creating a node, an edge and updating position can be done in $O(log(|\Sigma| + |\Pi|))$. Therefore, the total time for the traversals is $O(n log (|\Sigma| + |\Pi|))$.

From the above discussion, the following theorem is obtained.

**Theorem 10.** Given $t \in (\Sigma \cup \Pi)^n$, Algorithm 1 constructs $PPH(t)$ in $O(n log (|\Sigma| + |\Pi|))$ time and space.
Algorithm 1: Parameterized position heap online construction algorithm

Input: A p-string \( t \in (\Sigma \cup \Pi)^n \)
Output: A parameterized position heap \( \text{PPH}(t) \)
1. create root and \( \perp \) nodes;
2. \( \text{psp}(\text{root}) = \perp \);
3. \( \text{child}(\perp, c) = \text{root} \) for \( c \in \Sigma \cup \{0\} \);
4. \( \text{currentNode} = \text{root} \);
5. \( s = 1 \);
6. for \( i = 1 \) to \( n \) do
   7. \( c = \text{normalize}(\text{prev}(t)[i]), \text{depth}(	ext{currentNode})) \);
   8. lastCreateNode = undefined;
   9. while \( \text{child}(	ext{currentNode}, c) = \text{null} \) do
      10. create newnode;
      11. \( \text{prim}(\text{newnode}) = s \);
      12. \( \text{child}(	ext{currentNode}, c) = \text{newnode} \);
      13. if lastCreateNode \( \neq \) undefined then \( \text{psp}(\text{lastCreateNode}) = \text{newnode} \);
      14. lastCreateNode = \( \text{newnode} \);
      15. currentNode = \( \text{psp}(\text{currentNode}) \);
      16. \( c = \text{normalize}(\text{prev}(t)[i]), \text{depth}(	ext{currentNode})) \);
      17. \( s = s + 1 \);
      18. currentNode = \( \text{child}(	ext{currentNode}, c) \);
      19. if lastCreateNode \( \neq \) undefined then \( \text{psp}(\text{lastCreateNode}) = \text{currentNode} \);
   20. while \( s \leq n \) do
      21. \( \text{sec}(	ext{currentNode}) = s \);
      22. currentNode = \( \text{psp}(	ext{currentNode}) \);
      23. \( s = s + 1 \);

4.3 Augmented Parameterized Position Heaps

We will describe augmented parameterized position heaps, the parameterized position heaps with an additional data structure called the parameterized maximal-reach pointers similar to the maximal-reach pointers for the position heap [8]. The augmented parameterized position heap gives an efficient algorithm for parameterized pattern matching.

\textbf{Definition 11 (Parameterized Maximal-Reach Pointer).} For a position \( i \) on \( t \), a parameterized maximal-reach pointer of \( \text{pmrp}(i) \) is a pointer from node \( i \) to the deepest node whose path label is a prefix of \( \text{prev}(t[i :) \) .

Obviously, if \( i \) is a secondary position, then \( \text{pmrp}(i) \) is node \( i \) itself. We assume that the parameterized maximal-reach pointer for a double node applies to the primary position of this node. Figure 2 (b) shows an example of an augmented parameterized position heap. Given a prev-encoded p-string \( \text{prev}(w) \) represented in an augmented parameterized position heap \( \text{APPH}(t) \) and a position \( 1 \leq i \leq n \), we can determine whether \( \text{prev}(w) \) is a prefix of \( \text{prev}(t[i :) \) or not in \( O(1) \) time by checking whether \( \text{pmrp}(i) \) is a descendant of \( \text{prev}(w) \) or not. It can be done in \( O(1) \) time by appropriately preprocessing \( \text{APPH}(t) \) [5].

Parameterized maximal-reach pointers can be computed by using parameterized suffix pointers, similar to [13]. Algorithm 2 shows an algorithm to compute parameterized maximal-reach pointers. \( \text{pmrp}(i) \) is computed iteratively for \( i = 1, 2, \cdots, n \). Assume that we have computed \( \text{pmrp}(i) \) for some \( i \). Let \( \text{pmrp}(i) = \text{prev}(t[i : l]) \). Obviously, \( \text{prev}(t[i + 1 : l]) \) is a
prefix of the string represented by pmrp$(i+1)$. Thus, in order to compute pmrp$(i+1)$, we should extend the prefix prev$(t[i+1:l]) = \text{psp}(\text{prev}(t[i:l]))$ in PPH$(t)$ until we found $l'$ such that node prev$(t[i+1:l'])$ does not have outgoing edge labeled with prev$(t[i+1:])(l'-i+1)$ and set pmrp$(i+1) = \text{prev}(t[i+1:l'])$. In this time, we need re-compute prev$(t[i+1:]j)$ by replacing prev$(t[i+1:]j)$ with 0 if we found that prev$(t[i+1:]j) \geq j$. The total number of extending prev$(t[i+1:l])$ in the algorithm is at most $n$ because both $i$ and $l$ always increase in each iteration. In each iteration, operations such as traversing a child node can be done in $O(\log((|\Sigma|+|\Pi|)))$. Therefore, we can get the following theorem.

- **Theorem 12.** Parameterized maximal-reach pointers for PPH$(t)$ can be computed in $O(n \log((|\Sigma|+|\Pi|)))$ time.

### 4.4 Parameterized Pattern Matching with Augmented Parameterized Position Heaps

Ehrenfeucht et al. [8] and Kucherov [13] split a pattern $p$ into segments $q_1, q_2, \ldots, q_k$, then compute occurrences of $q_1q_2\cdots q_k$ iteratively for $j = 1, \ldots, k$. The correctness depends on a simple fact that for strings $x = t[i : i + |x| - 1]$ and $y = t[i + |x| : i + |x| + |y| - 1]$ implies $xy = t[i : i + |xy| - 1]$. However, when $x$, $y$, and $t$ are p-strings, prev$(x) = \text{prev}(t[i : i + |x| - 1])$ and prev$(y) = \text{prev}(t[i + |x| : i + |x| + |y| - 1])$ does not necessarily implies prev$(xy) = \text{prev}(t[i : i + |xy| - 1])$. Therefore, we need to modify the matching algorithm for parameterized strings.

Let $x$, $y$ and $w$ be p-strings such that $|w| = |xy|$, prev$(x) = \text{prev}(w[:|x|])$ and prev$(y) = \text{prev}(w[|x| + 1:])$. Let us consider the case that prev$(xy) \neq \text{prev}(w)$. From prev$(x) = \text{prev}(w[:|x|])$ and prev$(y) = \text{prev}(w[|x| + 1:])$, $x$ and $y$ have the same structure of $w[:|x|]$ and $w[|x| + 1:]$, respectively. However, the parameter symbols those are prev-encoded into 0 in prev$(y)$ and prev$(w[|x| + 1:])$, might be encoded differently in prev$(xy)$ and prev$(w)$, respectively. Therefore, we need to check whether prev$(xy)[|x|+i] = \text{prev}(w)[|x|+i]$ if prev$(y)[i] = 0$. Given prev$(xy)$ and the set of positions of 0 in prev$(y)$, $Z = \{i \mid 1 \leq i \leq |y|\}$ such that prev$(y)[i] = 0$. We need to verify whether prev$(xy)[|x|+i] = \text{prev}(w)[|x|+i]$ or not for $i \in Z$. Since the size of $Z$ is at most $|\Pi|$, this computation can be done in $O(|\Pi|)$ time.
Proof. Let $u$ be a node represents $prev(p)$. Assume $p$ occurs at position $i$ in $t$ and represented in $APPH(t)$ as $prev(t[i : k])$. Since either $prev(t[i : k])$ is a prefix of $prev(p)$ or $prev(p)$ is a prefix of $prev(t[i : k])$, then $i$ is either an ancestor or descendant of $u$. For both cases $pmrp(i)$ is a descendant of $u$, because $p$ occurs at position $i$.
Next let $i$ be a node such that $pmrp(i)$ is a descendant of $u$ and represents $\text{prev}(t[i : k])$. In this case, $\text{prev}(p)$ is a prefix of $\text{prev}(t[i : k])$. Therefore $p$ occurs at $i$.

**Lemma 14.** Assume $\text{prev}(p)$ is not represented in $\text{APPH}(t)$. We can split $p$ into $q_1, q_2, \ldots, q_k$ such that $q_j$ is the longest prefix of $\text{prev}(p[q_1 \cdots q_{j-1} + 1 :])$ that is represented in $\text{APPH}(t)$. If $p$ occurs at position $i$ in $t$, then $pmrp(i + |q_1 \cdots q_{j-1}|)$ is the node $\text{prev}(q_j)$ for $1 \leq j < k$ and $pmrp(i + |q_1 \cdots q_{k-1}|)$ is the node $\text{prev}(q_k)$ or its descendant.

**Proof.** Let $p = q_1 q_2 \cdots q_k$ occur at position $i$ in $t$. Since $\text{prev}(q_1)$ is a prefix of $\text{prev}(p)$, then $pmrp(i)$ is the node that represents $\text{prev}(q_1)$ or its descendant. However, if $pmrp(i)$ is a descendant of node $\text{prev}(q_1)$, then we can extend $q_1$ which contradicts with the definition of $q_1$. Therefore, $pmrp(i)$ is the node represents $\text{prev}(q_1)$.

Similarly for $1 < j < k$, $\text{prev}(q_j)$ is a prefix of $\text{prev}(p[q_1 \cdots q_{j-1} + 1 :])$ and occurs at position $i + |q_1 \cdots q_{j-1}|$ in $t$. Therefore, $pmrp(i + |q_1 \cdots q_{j-1}|)$ is the node represents $\text{prev}(q_j)$.

Last, since $q_k$ is a suffix of $p$, then $pmrp(i + |q_1 \cdots q_{k-1}|)$ can be the node $\text{prev}(q_k)$ or its descendant.

Algorithm 3 utilizes Lemmas 13 and 14 to find occurrences of $p$ in $t$ by using $\text{APPH}(t)$. First, if $\text{prev}(p)$ is represented in $\text{APPH}(t)$ then the algorithm will output all position $i$ such that $pmrp(i)$ is a node $\text{prev}(p)$ or its descendant. Otherwise, it will split $p$ into $q_1 q_2 \cdots q_k$ and find their occurrences as described in Lemma 14. The algorithm also checks whether $\text{prev}(q_1 \cdots q_j)$ occurs in $t$ or not in each iteration as described the above.

Examples of parameterized pattern matching by using an augmented position heap are given in Figure 4. Let $t = xaxyxyxyxyxyxy$ be the text. In Figure 4 (a) we want to find the occurrence positions of a pattern $p_1 = xyxy$ in $t$. In this case, since $\text{prev}(p_1) = 0022$ is represented in $\text{PPH}(t)$, The algorithm outputs all positions $i$ such that $pmrp(i)$ is the node 0022 or its descendants, those are 3, 4, 5, and 11. On the other hand, Figure 4 (b) shows how to find the occurrence positions of a pattern $p_2 = axyx$ in $t$. In this case, $\text{prev}(p_2) = a002$ is not represented in $\text{PPH}(t)$. Therefore, The algorithm finds the longest prefix of $\text{prev}(p_2)$ that is represented in $\text{PPH}(i)$, which is $\text{prev}(p_2)[1 : 2] = a0$. We can see that $pmrp(2) = pmrp(10) = a0$, then we save positions 2 and 10 as candidates to ans. Next, The algorithm finds the node that represents the longest prefix of $\text{prev}(p_2[3 :]) = 00$ which is $\text{prev}(p_2[3 :]) = 00$ itself. Since both of $pmrp(2 + |p_2[1 : 2]|) = pmrp(4)$ and $pmrp(10 + |p_2[1 : 2]|) = pmrp(12)$ is descendants of the node 00, $\text{prev}(t[2 : 5][3]) = \text{prev}(t[10 : 13][3]) = \text{prev}(p_2)[3] = 0$, and $\text{prev}(t[2 : 5][4]) = \text{prev}(t[10 : 13][4]) = \text{prev}(p_2)[4] = 2$, then the algorithm outputs 2 and 10.
The time complexity of the matching algorithm is as follow.

**Theorem 15.** Algorithm 3 runs in $O(m \log (|\Sigma| + |\Pi|) + m|\Pi| + occ)$ time.

**Proof.** It is easily seen that we can compute line 4 to 7 in $O(m \log (|\Sigma| + |\Pi|) + occ)$ time. Assume that $p$ can be decomposed into $q_1, q_2, \ldots, q_k$ such that $q_1$ is the longest prefix of $p$ and $q_i$ is the longest prefix of $\text{prev}(p[|q_1 \cdots q_{i-1}| + 1 :])$ represented in $APPH(t)$. The loop for line 15 consists of $k - 1$ iterations. In the loop line 18 in $j$-th iteration, $q_j + 1$ is extended up to reach $|q_j| + 1$ length. This can be computed in $O(|q_j + 1| \log (|\Sigma| + |\Pi|))$ time. After $k - 1$ iterations, the total number of extending of $q_j + 1$ does not exceed $m$, because $\Sigma_{j=2}^k |q_j| < m$. In the loop for line 25, the algorithm verifies elements of $\text{ans}$. In $j$-th iteration, the size of $\text{ans}$ is at most $|q_j|$. Thus, after $k - 1$ iterations, the total number of elements verified in line 25 does not exceed $m$ by the same reason for that of line 18. In each verification in line 25, the number of checks for line 27 and 29 is at most $|q_j|$. Therefore, it can be computed from line 25 to 32 in $O(m|\Pi|)$ time.

## 5 Conclusion and Future Work

For the parameterized pattern matching problem, we proposed an indexing structure called a parameterized position heap. Given a $p$-string $t$ of length $n$ over a constant size alphabet, the parameterized position heap for $t$ can be constructed in $O(n \log (|\Sigma| + |\Pi|))$ time by our construction algorithm. We also proposed an algorithm for the parameterized pattern matching problem. It can be computed in $O(m \log (|\Sigma| + |\Pi|) + m|\Pi| + occ)$ time using parameterized position heaps with parameterized maximal-reach pointers. Gagie et al. [10] showed an interesting relationship between position heap and suffix array of a string. We will examine this relation for parameterized position heap and parameterized suffix array [7, 12] as a future work.

## References


