Noname manuscript No.
(will be inserted by the editor)

Validating the Knuth-Morris-Pratt failure function, fast and online

Paweł Gawrychowski · Artur Jeż · Łukasz Jeż

Received: date / Accepted: date

Abstract Let π'_w denote the failure function of the Knuth-Morris-Pratt algorithm for a word w. In this paper we study the following problem: given an integer array $A'[1\,..\,n]$, is there a word w over an arbitrary alphabet Σ such that $A'[i] = \pi'_w[i]$ for all i? Moreover, what is the minimum cardinality of Σ required? We give an elementary and self-contained $\mathcal{O}(n \log n)$ time algorithm for this problem, thus improving the previously known solution [8], which had no polynomial time bound. Using both deeper combinatorial insight into the structure of π' and advanced algorithmic tools, we further improve the running time to $\mathcal{O}(n)$.

1 Introduction

1.1 Pattern recognition and failure functions

The Morris-Pratt algorithm [20], first linear time pattern matching algorithm, is well known for its beautiful concept. It simulates the minimal DFA recognizing Σ^*p (p denotes the pattern) by using a *failure function* π_p, known as the *border array*. The automaton's transitions are recovered, in amortized constant time, from the values of π_p for all prefixes of the pattern, to which the DFA's states correspond. The values of π_p are precomputed in a similar fashion, also in linear time.

The MP algorithm has many variants. For instance, the Knuth-Morris-Pratt algorithm [17] improves it by using an optimised failure function, namely the *strict border array* π' (or *strong failure function*). This was improved by Simon [22], and further improvements are known [13, 1]. We focus on the KMP failure function for two reasons. Unlike later algorithms, it is well-known and used in practice. Furthermore, the strong border array itself is of interest as, for instance, it captures all the information about periodicity of the word. Hence it is often used in word combinatorics and numerous text algorithms, see [4, 6]. On the other hand, even Simon's algorithm (i.e., the very first improvement) deals with periods of pattern

P. Gawrychowski, A. Jeż, Ł. Jeż
Institute of Computer Science, University of Wrocław
E-mail: gawry@cs.uni.wroc.pl, E-mail: aje@cs.uni.wroc.pl, E-mail: lje@cs.uni.wroc.pl

prefixes augmented by a single text symbol rather than pure periods of pattern prefixes.

1.2 Strict border array validation

Problem statement We investigate the following problem: given an integer array $A'[1\,..\,n]$, is there a word w over an arbitrary alphabet Σ such that $A'[i] = \pi'_w[i]$ for all i, where π'_w denotes the failure function of the Knuth-Morris-Pratt algorithm for w. If so, what is the minimum cardinality of the alphabet Σ over which such a word exists?

Pursuing these questions is motivated by the fact that in word combinatorics one is often interested only in values of π'_w rather than w itself. For instance, the logarithmic upper bound on delay of KMP follows from properties of the strict border array [17]. Thus it makes sense to ask if there is a word w admitting $\pi'_w = A'$ for a given array A'.

We are interested in an *online* algorithm, i.e., one that receives the input array values one by one, and is required to output the answer after reading each single value. For the Knuth-Morris-Pratt array validation problem it means that after reading $A'[i]$ the algorithm should answer, whether there exist a word w such that $A'[1\,..\,i] = \pi'_w[1\,..\,i]$ and what is the minimum size of the alphabet over which such a word w exists.

Previous results To our best knowledge, this problem was investigated only for a slightly different variant of π', namely a function g that can be expressed as $g[n] = \pi'[n-1]+1$, for which an offline validation algorithm due to Duval et al. [9] is known. Validation of border arrays is used by algorithms generating all valid border arrays [9, 11, 19].

Unfortunately, Duval et al. [8] provided no upper bound on the running time of their algorithm, but they did observe that on certain input arrays it runs in $\Omega(n^2)$ time.

Our results We give a simple $\mathcal{O}(n \log n)$ online algorithm Validate-π' for the strong border array validation, which uses the linear offline bijective transformation between π and π'. Validate-π' is also applicable to g validation with no changes, thus giving the first provably polynomial algorithm for the problem considered by Duval et al. [8]. Note that aforementioned bijection between π and π' cannot be applied directly to g, as it essentially uses the unavailable value $\pi[n] = \pi'[n]$, see Section 2.

Then we improve Validate-$\pi'$to an optimal linear online algorithm Linear-Validate-π'. The improved algorithm relies on both more sophisticated data structures, such as dynamic suffix trees supporting LCA queries, and deeper insight into the combinatorial properties of π' function.

Related results The study of validating arrays related to string algorithms and word combinatorics was started by Franěk et al. [11], who gave an offline linear algorithm for border array validation. This result was improved over time, in particular a simple linear online algorithm for π validation is known [9].

Table 1 Functions π and π' for a word *aabaabaaabaac*.

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$w[i]$	a	a	b	a	a	b	a	a	a	b	a	a	b	a	a	c
$\pi[i]$	0	1	0	1	2	3	4	5	2	3	4	5	6	7	9	0
$\pi[i]$	-1	1	-1	-1	1	-1	-1	5	1	-1	-1	1	-1	-1	8	0

The border array validation problem was also studied in the more general setting of the *parametrised border array* validation [14,15], where parametrised border array is a border array for text in which a permutation of letters of alphabet is allowed. A linear time algorithm for a restricted variant of this problem is known [14] and a $\mathcal{O}(n^{1.5})$ for the general case [15].

Recently a linear online algorithm for a closely related *prefix array* validation was given [2], as well as for *cover array* validation [5].

2 Preliminaries

For $w \in \Sigma^*$, we denote its length by $|w|$. For $v, w \in \Sigma^*$, by vw we denote the concatenation of v and w. We say that u is a *prefix* of w if there is $v \in \Sigma^*$ such that $w = uv$. Similarly, we call v a *suffix* of w if there is $u \in \Sigma^*$ such that $w = uv$. A word v that is both a prefix and a suffix of w is called a *border* of w. By $w[i]$ we denote the i-th letter of w and by $w[i \,..\, j]$ we denote the *subword* $w[i]w[i+1]\ldots w[j]$ of w. We call a prefix (respectively: suffix, border) v of the word w *proper* if $v \neq w$, i.e., it is shorter than w itself.

For a word w its *failure function* π_w is defined as follows: $\pi_w[i]$ is the length of the longest proper border of $w[1 \,..\, i]$ for $i = 1, 2 \ldots, n$. By $\pi_w^{(k)}$ we denote the k-fold composition of π_w with itself, i.e., $\pi_w^{(0)}[i] := i$ and $\pi_w^{(k+1)}[i] := \pi_w[\pi_w^{(k)}[i]]$. This convention applies to other functions as well. We omit the subscript w in π_w, whenever it is unambiguous. Note that every border of $w[1 \,..\, i]$ has length $\pi_w^{(k)}[i]$ for some integer $k \geq 0$.

Algorithm 1 COMPUTE-$\pi(w)$

$\pi[1] \leftarrow 0$
$k \leftarrow 0$
for $i \leftarrow 2$ to n **do**
  while $k > 0$ and $w[k+1] \neq w[i]$ **do**
    $k \leftarrow \pi[k]$
  end while
  if $w[k+1] = w[i]$ **then**
    $k \leftarrow k + 1$
  end if
  $\pi[i] \leftarrow k$
end for

The *strong failure function* π' is defined as follows: $\pi'_w[n] := \pi_w[n]$, and for $i < n$, $\pi'[i]$ is the largest k such that $w[1 \,..\, k]$ is a proper border of $w[1 \,..\, i]$ and $w[k+1] \neq w[i+1]$. If no such k exists, $\pi'[i] = -1$.

It is well-known that π_w and π'_w can be obtained from one another in linear time, using additional lookups in w to check whether $w[i] = w[j]$ for some i, j. What is perhaps less known, these lookups are not necessary, i.e., there is a constructive bijection between π_w and π'_w. For completeness, we supply both procedures. By standard argument it can be shown that they run in linear time. The correctness as well as the procedures themselves are a consequence of the following observation

$$w[i+1]=w[\pi[i]+1] \iff \pi[i+1]=\pi[i]+1 \iff \pi'[i]<\pi[i] \iff \pi'[i]=\pi'[\pi[i]] \quad . \tag{1}$$

Algorithm 2 π'-FROM-$\pi(\pi)$

```
π′[0] ← −1
for i ← 1 to n − 1 do
    if π[i + 1] = π[i] + 1 then
        π′[i] ← π′[π[i]]
    else
        π′[i] ← π[i]
    end if
end for
π′[n] ← π[n]
```

Algorithm 3 π-FROM-$\pi'(\pi')$

```
π[n] ← π′[n]
for i ← n − 1 downto 1 do
    π[i] ← max{π′[i], π[i + 1] − 1}
end for
```

Note that procedure π'-FROM-π explicitly uses the following recursive formula for $\pi'[j]$ for $j < n$, whose correctness follows from (1):

$$\pi'[j] = \begin{cases} \pi[j] & \text{if } \pi[j+1] < \pi[j]+1 \ , \\ \pi'[\pi[j]] & \text{if } \pi[j+1] = \pi[j]+1 \ . \end{cases} \tag{2}$$

2.1 Border array validation

Our algorithm uses an algorithm validating the input table as the border array. For completeness, we supply the code of one of the simplest such algorithms VALIDATE-π, due to Duval et al. [9]. This algorithm is online and also calculates the minimal size of the required alphabet.

Roughly speaking, given a valid border array $A[1 \mathinner{.\,.} n]$ VALIDATE-π computes all valid π-candidates for $A[n+1]$: given a valid border array $A[1 \mathinner{.\,.} n]$ the next element $A[n+1]$ is a *valid π-candidate* if $A[1 \mathinner{.\,.} n+1]$ is a valid border array as well. The exact formula for the set of valid candidates is not for us, though it should be noted that it depends only on $A[1 \mathinner{.\,.} n]$ and that 0 and $A[n]+1$ are always valid π-candidates.

Algorithm 4 VALIDATE-$\pi(A)$

```
if A[1] ≠ 0 then
    error A is not valid at 1
end if
cand[1] ← {0}, w[1] ← 1, Σ[1] ← 1
for i = 2 to n do
    if A[i] = 0 then
        cand[i] ← {0}
        Σ[i] ← Σ[A[i − 1] + 1] + 1
        MinΣ ← max(MinΣ, Σ[i])
        w[i] ← Σ[i]
    else
        cand[i] ← cand[A[i − 1] + 1]
        remove A[A[i − 1] + 1] from cand[i]
        add A[i − 1] + 1 to cand[i]
        if A[i] ∉ cand[i] then
            error A is not valid at i
        end if
        w[i] ← w[A[i]]
        Σ[i] ← Σ[A[i − 1] + 1]
    end if
end for
```

For future reference we list several properties that follow from VALIDATE-π:

(Val1) the valid candidates for $\pi[i]$ depend only on $\pi[1\mathinner{.\,.}i-1]$,
(Val2) $\pi[i-1]+1$ is always a valid candidate for $\pi[i]$,
(Val3) if the alphabet needed for $A[1\mathinner{.\,.}n]$ is strictly larger than the one needed for $A[1\mathinner{.\,.}n-1]$ then $A[n]=0$.

3 Overview of the algorithm

Since there is a bijection between valid border arrays and valid strict border arrays, it is natural to proceed as follows. Assume the input forms a valid strict border array, obtain the corresponding border array using π-FROM-$\pi'(A')$, and validate the result using VALIDATE-$\pi(A)$. Unfortunately, π-FROM-π' starts the calculations from the last entry of A', so it is not suitable for an online algorithm. Moreover, it assumes that $A'[n]=A[n]$, which may be not true for some intermediate value of i. As a consequence, there can be many border arrays consistent with $A'[1\mathinner{.\,.}i]$. We show that all these border arrays coincide on a certain prefix. VALIDATE-π' identifies this prefix and runs VALIDATE-π on it. Concerning the suffix, VALIDATE-π' identifies the border array which is maximal on it, in a sense explained below.

Definition 1 (Consistent functions) We say that $A[1\mathinner{.\,.}n{+}1]$ is *consistent* with $A'[1\mathinner{.\,.}n]$ if and only if there is a word $w[1\mathinner{.\,.}n+1]$ such that

(CF1) $A[1\mathinner{.\,.}n+1]=\pi_w[1\mathinner{.\,.}n+1]$,
(CF2) $A'[1\mathinner{.\,.}n]=\pi'_w[1\mathinner{.\,.}n]$.

Among functions consistent with A' there exists the maximal one:

Definition 2 (Maximal consistent function) A function $A[1\mathinner{.\,.}n{+}1]$ consistent with $A'[1\mathinner{.\,.}n]$ is *maximal* if

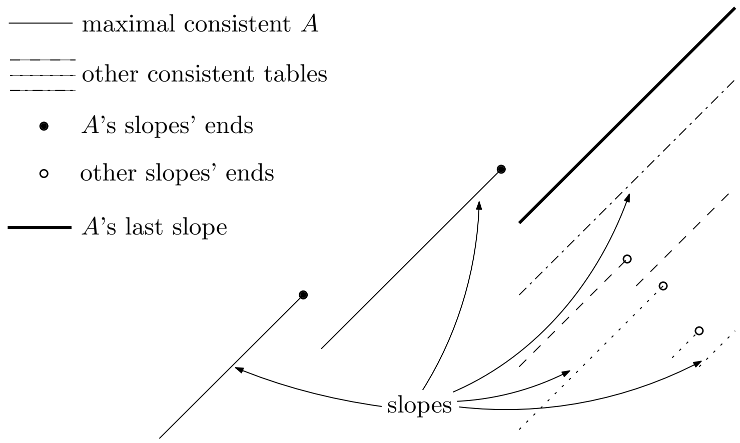

Fig. 1 Graphical illustration of slopes and maximal consistent function.

(CF3) every $B[1\,..\,n{+}1]$ consistent with $A'[1\,..\,n]$ satisfies $B[1\,..\,n{+}1] \le A[1\,..\,n{+}1]$,

where $A[1\,..\,m] \ge B[1\,..\,m]$ denotes that $A[j] \ge B[j]$ for $j = 1, \ldots, m$.

Our algorithms Validate-π' and Linear-Validate-π' maintain such a maximal A.

Slopes and their properties Imagine the array A' as the set of points $(i, A'[i])$ on the plane; we think of A in the similar way. Such a picture helps in understanding the idea behind the algorithm. In this setting we think of A as a collection of maximal *slopes*: a set of indices $i, i{+}1, \ldots, i{+}j$ is a slope if $A[i{+}k] = A[i]{+}k$ for $k = 1, \ldots, j$. Note that $A[i+j+1] \neq A[i+j]+1$ implies that $A[i+j] = A'[i+j]$, by (1). Let the *pin* be the first position on the last slope of A. Validate-π' calculates and stores the pin. It turns out that all functions consistent with A' differ from A only on the *last slope*, as shown later in Lemma 1.

When a new input value $A'[n]$ is read, the values of A and A' on the last slope should satisfy the following conditions:

$$A'[j] < A[j], \qquad \text{for each } j \in [i\,..\,n]\ , \tag{3}$$

$$A'[j] = A'[A[j]], \qquad \text{for each } j \in [i\,..\,n]\ . \tag{4}$$

Algorithm 5 Validate-$\pi'(A')$

```
A[1] ← 0
i ← 0
n ← 1
A'[0] ← −1
while TRUE do
    n ← n + 1
    if A'[n] ≠ A'[A[n]] then
        ADJUST-LAST-SLOPE
    end if
end while
```

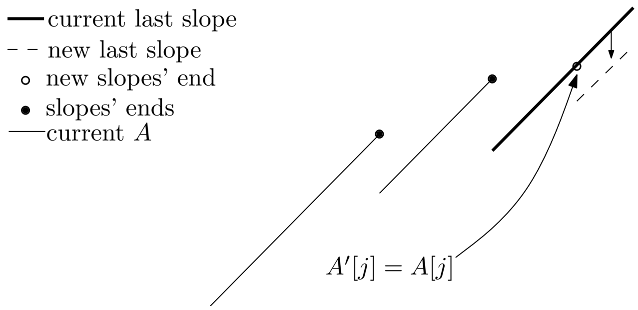

Fig. 2 Splitting the last slope.

The last slope is defined correctly if and only if (3) holds, while the values of A and A' on the last slope are consistent if and only if (4) holds. These conditions are checked by appropriate queries: (3) by the *pin value check* (denoted PIN-VALUE-CHECK), which returns any $j \in [i\,..\,n]$ such that $A'[j] > A[j]$ or, if there is no such j, the smallest $j \in [i\,..\,n]$ such that $A'[j] = A[j]$; and (4) by the *consistency check* (denoted CONSISTENCY-CHECK), which checks whether $A'[i\,..\,n] = A'[A[i]\,..\,A[i] + (n-i)]$.

If one of the conditions (3) or (4) does not hold, VALIDATE-π' adjusts the last slope of A, until both conditions hold or the input is reported as invalid.

Algorithm 6 ADJUST-LAST-SLOPE

while $j \leftarrow$ PIN-VALUE-CHECK is defined **do**
    if $A'[j] > A[i] + (j - i)$ **then**
        error A' is not valid at n
    end if
    for $m \leftarrow i$ to $j - 1$ **do**
        store $A[m] \leftarrow A[m-1] + 1$
        VALIDATE-$\pi(A)[m]$
        if $A'[m] \neq A'[A[m]]$ **then**
            error A' is not valid at n
        end if
    end for
    store $A[j] \leftarrow A[j-1] + 1$
    $i \leftarrow j + 1$
    $A[i] \leftarrow$ next candidate
end while
if not CONSISTENCY-CHECK **then**
    if $A[i] = 0$ **then**
        A' is not valid at n
    end if
    $A[i] \leftarrow$ next candidate
    goto 1:
end if

If the pin value check returns an index j such that $A'[j] > A[j]$, then we reject the input and report an error: since A is the maximal consistent function, for each consistent function A_1 it also holds that $A_1[j] < A'[j]$ and so none such A_1 exists and so A' is invalid.

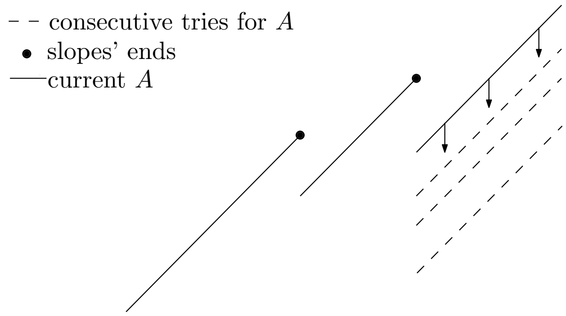

Fig. 3 Decreasing the $A[i]$.

If $A'[j] = A[j]$ we break the last slope in two: $[i \mathinner{..} j]$ and $[j+1 \mathinner{..} n]$, the new last slope, see Fig 2: since A is maximal consistent function, for every other A_1 consistent with A' on the one hand $A_1[j] \leq A[j] = A'[j]$, and on the other $A_1[j] \geq A'[j] \geq A[j]$. We also check whether

$$A'[i \mathinner{..} j-1] = A'[A[i] \mathinner{..} A[i] + (j-i-1)]$$

holds. If not, we reject: every other consistent table $A_1[j] = A[j] = A'[j]$ and therefore they have to be equal on all preceding values as well. Next we set i to $j+1$ and $A[i]$ to the largest valid candidate value for $\pi[i]$.

If Consistency-Check check fails, then we set the value of $A[i]$ to the next valid candidate value for $\pi[i]$, see Fig. 3 and propagate the change along the whole slope. If this happens for $A[i] = 0$, then there is no further candidate value, and A' is rejected. The idea is that some adjustment is needed and since pin value check does not return an index, we cannot break the slope into two and so the only possibility is to decrement A on the whole last slope.

Unfortunately, this simple combinatorial idea alone fails to produce a linear algorithm. The problem is caused by the second condition: large segments of A' should be compared in amortised constant time. While LCA queries on suffix trees seem ideal for this task, available solutions are imperfect: the online suffix tree construction algorithms [18, 23] are linear for constant size-alphabets only, while the only linear-time algorithm for non-constant alphabets [10] is inherently offline. To overcome this obstacle we specialise the data structures used, building the suffix tree for compressed encoding of A' and multiple suffix trees for short texts over polylogarithmic alphabet. The details are presented in Section 8.

4 Details and correctness

In this section we present technical details of the algorithm, provide a proof of its correctness and proofs of used combinatorial properties. We start with showing that all the consistent tables coincide on indices smaller than pin.

Lemma 1 *Let $A[1 \mathinner{..} n+1] \geq B[1 \mathinner{..} n+1]$ be both consistent with $A'[1 \mathinner{..} n]$. Let i be the pin (for A). Then $A[1 \mathinner{..} i-1] = B[1 \mathinner{..} i-1]$.*

Proof The claim holds vacuously when there is only one slope, i.e., $i = 1$. If there are more, let i be the pin and consider $i-1$. Since it is the end of a slope, by (1) $A'[i-1] = A[i-1]$. On the other hand, consider $B[1\,..\,n+1]$ as in the statement of the lemma. By assumption of the lemma, $A[i-1] \geq B[i-1]$. Thus

$$A'[i-1] \leq B[i-1] \leq A[i-1] = A'[i-1] \ ,$$

hence $B[i-1] = A[i-1]$. Let $B[1\,..\,n+1] = \pi_{w'}[1\,..\,n+1]$ and $A[1\,..\,n+1] = \pi_w[1\,..\,n+1]$. Using π-FROM-π' we can uniquely recover $\pi_{w'}[1\,..\,i-1]$ from $\pi'_{w'}[1\,..\,i-1]$ and $\pi_{w'}[i-1]$, as well as $\pi_w[1\,..\,i-1]$ from $\pi'_w[1\,..\,i-1]$ and $\pi_w[i-1]$. But since those pairs of values are the same,

$$A[1\,..\,i-1] = \pi_w[1\,..\,i-1] = \pi_{w'}[1\,..\,i-1] = B[1\,..\,i-1] \ ,$$

which shows the claim of the lemma. □

Data maintained VALIDATE-π'stores:

- n, the number of values read so far,
- $A'[1\,..\,n]$, the input read so far,
- i, the current pin
- $A[1\,..\,n+1]$, the maximal function consistent with $A'[1\,..\,n]$:
  - $A[1\,..\,i-1]$, the fixed prefix,
  - $A[i]$, the candidate value that may change.

Note that $A[j]$ for $j > i$ are not stored. These values are implicit, given by $A[j] = A[i]+(j-i)$. In particular this means that decrementing $A[i]$ results in decrementing the whole last slope.

Sets of valid π candidates and validating A VALIDATE-π' creates a border array A, which is always valid by the construction. Nevertheless, it runs VALIDATE-$\pi(A[1\,..\,i-1])$. This way the set of valid candidates for $\pi[i]$ is computed, as well as a word w over Σ that admits A.

In the remainder of this section it is shown that CF1–CF3 are preserved by VALIDATE-π'.

Lemma 2 *If $A'[n] = A'[A[n]]$ and no changes are done, the CF1–CF3 hold.*

Proof Whenever a new symbol is read, VALIDATE-π'checks (4) for $j = n$, i.e., whether $A'[n] = A'[A[n]]$. If it holds, then no changes are needed because:

CF1 holds trivially: the implicit $A[n+1] = A[n]+1$ is always a valid value for $\pi[n+1]$, see Val2.

CF2 holds: as $A'[n] < A[n]$ by (1) it is enough to check that $A'[n] = A'[A[n]]$, which holds by (4).

CF3 holds: consider any $B[1\,..\,n+1]$ consistent with $A'[1\,..\,n]$. By induction assumption CF3 holds for $A[1\,..\,n]$, hence $B[n] \leq A[n]$. Therefore

$$B[n+1] \leq B[n]+1 \leq A[n]+1 = A[n+1] \ ,$$

which shows the last claim and thus completes the proof. □

Thus it is left to show that CF1–CF3 are preserved by Adjust-Last-Slope. We show that during the adjusting inside Adjust-Last-Slope CF1 and CF3. To be more specific, CF1 alone means that A is always a valid border, while CF3 means that it is greater than any border table consistent with A' (this is assumed to hold vacuously if no consistent table exists). What is missing is that A is in fact consistent with A'. We show that this holds when Adjust-Last-Slope ends.

For the completeness of the proof, we need also show that if at any point A' was reported to be invalid, it in fact is invalid.

Lemma 3 *If* Pin-Value-Check *returns an index j such that A' is rejected in line 3 then A' is invalid. If* Pin-Value-Check *returns an index j, and A is adjusted, then afterwards CF1 and CF3 are satisfied.*

Proof Let $A_1[1\,..\,n+1]$ be any table consistent with $A'[1\,..\,n]$. Suppose that Pin-Value-Check returns j such that $A[j] < A'[j]$. Then, since CF3 is satisfied, $A_1[j] \leq A[j] < A'[j]$, i.e., A_1 is not a valid π table. So no A_1 is consistent with A', which means that A' is invalid, as reported by Validate-π'.

Suppose that Pin-Value-Check returns j such that $A[j] = A'[j]$. We first show that CF1 is satisfied: $A[j]$ is explicitly set to a valid π candidate while for $p > j$ the $A[p]$ is set $A[p] = A[p-1]+1$, which is always a valid π candidate, by Val2.

It is left to show that if A' is declared invalid in line 9 then it is invalid; and otherwise CF3 is satisfied after the adjustments. Firstly we prove that j is an end of slope for A_1. By CF3, $A_1[j] \leq A[j] = A'[j]$ but as A_1 is a valid π table, $A_1[j] \geq A'[j]$. So $A'[j] = A_1[j]$ and therefore, by (2), it is an end of a slope for A_1. As a consequence, by Lemma 1, $A[i\,..\,j] = A_1[i\,..\,j]$. In particular, if A_1 exists, $A[i\,..\,j]$ is consistent with $A'[i\,..\,j-1]$. Note that for $p \in [i\,..\,j-1]$ it holds that $A[p] > A'[p]$: otherwise Pin-Value-Check would have returned such p instead of j. Thus, by (1), $A[p]$ and $A'[p]$ should satisfy equation $A'[p] = A'[A[p]]$, and this condition is verified by Validate-π' in line 8. If this equation is not satisfied by some p then clearly $A'[i\,..\,j-1]$ is not consistent with $A[i\,..\,j]$, which holds assuming that A_1 exists. So no such A_1 exists and A' is invalid.

Suppose that A' was not rejected. It is left to show that CF3 is satisfied. Since $A_1[i]$ is a valid π value and $A[i]$ is the maximal valid π value, $A_1[i] \leq A[i]$. The implicit values $A[j]$ for $j \in [i+1\,..\,n]$ satisfy $A[j] = A[i] + (j-i)$. Since A_1 is a valid π table $A_1[p] \leq A[p]$ for $p = i, \ldots, j$ and thus:

$$A_1[p] \leq A_1[i] + (p-i) \leq A[i] + (p-i) = A[p] \ ,$$

and as A_1 was chosen arbitrarily, CF3 is satisfied. □

Lemma 4 *If* Consistency-Check *returns* false *and $A[i] = 0$ then A' is invalid. Otherwise after adjusting in line 20, CF1 and CF3 hold.*

Proof Since Validate-π' decreases the value of $A[i]$, we want to show that $A[i] > A_1[i]$ for any A_1 consistent with A'. In such case $A[i]$ is assigned a valid π candidate and therefore after the adjustment $A[i] \geq A_1[i]$. Then for $p > i$ the values $A[p]$ are implicit and therefore $A[p] = A[i] + (p-i)$. On the other hand A_1 is a valid border array and thus $A_1[p] \leq A[i] + (p-i)$. Thus

$$A_1[p] \leq A_1[i] + (p-i) \leq A[i] + (p-i) = A[p] \ .$$

So CF3 holds for A after the adjustment.

Assume, for the sake of contradiction, that $A_1[i] = A[i]$. Let j be such that $A'[j] \neq A'[A[j]]$. It is first shown that $A_1[j] < A[j]$ and from this it is inferred that $A[i] > A_1[i]$, obtaining a contradiction. Note, that by CF3 it is enough to show that $A_1[j] \neq A[j]$.

By (1) either $A_1[j] = A'[j]$ or $A_1[j] > A'[j]$; we show that in each of these cases $A_1[j] < A[j]$.

- if $A_1[j] = A'[j]$, then Consistency-Check is called only when Pin-Value-Check returns no index, in particular for j it holds that $A[j] > A'[j]$; and so $A[j] > A_1[j]$
- if $A_1[j] > A'[j]$, then $A'[j] = A'[A_1[j]]$ by (1). But j satisfies $A'[j] \neq A'[A[j]]$, and so $A_1[j] \neq A[j]$.

Since $A_1[i] = A[i]$ and $A_1[j] \neq A[j]$ there exists the smallest $j' < j$ such that $A_1[j'] = A[j']$ and $A_1[j'+1] \neq A[j'+1]$. Then $A_1[j'+1] < A_1[j']+1$, as:

$$A_1[j'+1] < A[j'+1] = A[j'] + 1 = A_1[j'] + 1 \ .$$

By (1) this implies $A'[j'] = A_1[j']$ and as $A_1[j'] = A[j]$, we conclude that $A'[j'] = A[j']$. Contradiction, as j was returned by Pin-Value-Check as the smallest index satisfying $A'[j] = A[j]$.

So $A_1[i] < A[i]$. In particular, if $A[i] = 0$, there is no such A_1 and hence A' is invalid. Otherwise Validate-π' sets $A[i]$ to next largest valid candidate for $\pi[i]$.

It is left to show that CF1 holds, i.e., that $A[i \mathinner{.\,.} n+1]$ were all assigned valid candidates for π at their respective positions. This was addressed explicitly for $A[i]$, while for $p > i$ the assigned values are $A[p-1]+1$, which are always valid by Val2. ◻

The last lemma shows that when Adjust-Last-Slope finishes, CF2 is satisfied as well.

Lemma 5 *When* Adjust-Last-Slope *finishes, CF2 is satisfied.*

Proof We already know the recursive formula for π', it is given in (2). Note, that the first case corresponds to j being the last element on the slope and the second case corresponds to other j's.

If $A[j]$ is an explicit value and j is not an end of a slope, this formula is verified, when $A[j]$ is stored. If $A[j]$ is explicit and j is an end of the slope then the formula trivially holds.

If $A[j]$ is an implicit value, i.e., such that j is on the last slope of A, Pin-Value-Check guarantees that $A[j] > A'[j]$ and so the second case of this formula should hold. This is verified by Consistency-Check. Hence CF2 holds when all adjustments are finished. ◻

The above four lemmata together show the correctness of Validate-π'.

Theorem 1 Validate-π' *answers if A' is a valid strict border array. If so, it supplies the maximal function A consistent with A'.*

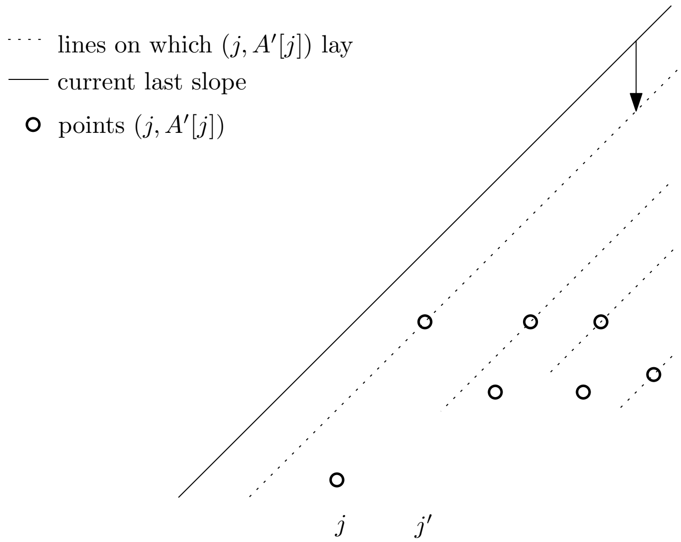

Fig. 4 Answering pin value check.

Proof We proceed by induction on n. If $n = 0$, then clearly $A[1] = 0$ and CF1–CF3 trivially hold, and A' is a valid (empty) π' array. If $n > 0$ and no adjustments were done, CF1–CF3 hold by Lemma 2. So we consider the case, when ADJUST-LAST-SLOPE was invoked.

By Lemma 3 and Lemma 4 if the $A'[1\,..\,n]$ is rejected, it is invalid. So assume that $A'[1\,..\,n]$ was not rejected. We show that it is valid. As it was not rejected, by Lemma 3 and Lemma 4 the constructed table $A[1\,..\,n+1]$ together with $A'[1\,..\,n]$ satisfy CF1 and CF3. Moreover, by Lemma 5 they satisfy also CF2. Thus $A[1\,..\,n+1]$ is a valid border array for some word $w[1\,..\,n+1]$ and $A'[1\,..\,n]$ is a valid strong border array for the same word $w[1\,..\,n]$. □

5 Performing pin value checks

Consider the PIN-VALUE-CHECK and any two indices $j < j'$ such that

$$A'[j'] - j' > A'[j] - j \ .$$

We denote this relation by $j \prec j'$ and say that j' *dominates* j. We show that if $j' \succ j$ and j is an answer to PIN-VALUE-CHECK, so is j', consult Fig. 4. This observation allows to keep a collection $j_1 < j_2 < \cdots < j_\ell$ of indices such that to perform the pin value check, it is enough to see whether $A[j_1] < A'[j_1]$. In particular, the answer can be given in constant time. Updates of this collection are done by either removal of j_1, when i becomes $j_1 + 1$, or by consecutive removals from the end of the list, when a new $A'[n]$ is read.

Note that since CF1 is satisfied inside ADJUST-LAST-SLOPE, by Lemma 3 and Lemma 4; i.e, A is a valid border array, in particular $A[i] \le A[j] + (i - j)$ whenever $j < i$.

Properties of $\prec$ As $\prec$ is an intersection of two transitive relations (order on indices and order on T, defined as $T[j] = A[j] - j$), it is transitive.

Observe that if $j \prec j'$, then $A'[j] \geq A[j]$ implies $A[j'] < A'[j']$:

$$\begin{aligned} A[j'] &\leq A[j] + (j' - j) \\ &\leq A'[j] + (j' - j) \\ &< A'[j] + (A'[j'] - A'[j]) \\ &= A'[j'] \ . \end{aligned} \tag{5}$$

Therefore if j is an answer to pin value check, so is j'. Hence we need not keep track of j as a potential answer to the Pin-Value-Check.

Data stored Validate-π' stores a list of positions $j_1 < j_2 < \cdots < j_k$ such that (for the sake of simplicity, let $j_0 = i$):

$$j_{\ell'} \not\prec j_\ell \qquad \text{for all } 0 < \ell' < \ell \ , \tag{6}$$

$$j_\ell \succ j \qquad \text{for all } 0 < \ell \leq k \text{ and } j \in [j_{\ell-1} + 1 \mathinner{.\,.} j_\ell - 1] \ . \tag{7}$$

Answering Pin-Value-Check When Pin-Value-Check is asked, we check if $A[j_1] \leq A'[j_1]$. This way the Pin-Value-Check is answered in constant time. We show that evaluating this expression for other values of j is not needed. Suppose that $A'[j] \geq A[j]$ for some $j \in [j_{\ell-1} + 1 \mathinner{.\,.} j_\ell - 1]$. Since j_ℓ dominates j it holds that $A'[j_\ell] > A[j_\ell]$, by (5). Suppose now that $A'[j_\ell] \geq A[j_\ell]$ for $j_\ell > j_1$. Since $j_1 < j_\ell$ and j_ℓ does not dominate j_1,

$$A'[j_\ell] - A'[j_1] \leq j_\ell - j_1 \ .$$

As j_1 and j_ℓ are on the last slope,

$$A[j_\ell] = A[j_1] + (j_\ell - j_1) \ ,$$

and hence

$$\begin{aligned} A[j_1] &= A[j_\ell] - (j_\ell - j_1) \\ &\leq A[j_\ell] - (A'[j_\ell] - A'[j_1]) \\ &= A'[j_1] + (A[j_\ell] - A'[j_\ell]) \\ &\leq A'[j_1] \ , \end{aligned}$$

so j_1 is a proper answer to the Pin-Value-Check. Similarly, $A'[j_\ell] > A[j_\ell]$ implies $A'[j_1] > A[j_1]$.

Update We demonstrate that all updates of the list $j_1, \ldots, j_k$ can be done in $\mathcal{O}(n)$ time. When new position n is read, we update the list by successively removing j_ℓ's dominated by n from the end of the queue. By routine calculations, if $n \succ j_\ell$, then $n \succ j_{\ell+1}$ as well:

$$A[n] - n > A[j_\ell] - j_\ell \ ,$$
$$A[j_\ell] - j_\ell \geq A[j_{\ell+1}] - j_{\ell+1} \ ,$$

with the latter being (6). Therefore

$$A[n] - A[j_{\ell+1}] > n - j_{\ell+1} \ .$$

So we simply have to remove some tail from the list of j's. Suppose that $j_\ell, \ldots, j_k$ were removed. It is left to show that after the removal (6) and (7) are preserved. Consider first (6), i.e., any $j \in [j_{\ell-1} \mathinner{.\,.} n-1]$. Then there is some $j_{\ell'}$ such that $j \in [j_{\ell'-1} \mathinner{.\,.} j_{\ell'} - 1]$. By (7), $j_{\ell'} \succ j$. Since by assumption $n \succ j_{\ell'}$, by transitivity of $\succ$, also $n \succ j$. Consider now (7). It holds by the assumption, as if since $j_{\ell-1}$, it holds that $j_{\ell-1} \not\prec n$, as desired.

There is another possible update: when Pin-Value-Check return j_1 then $i \leftarrow j_1 + 1$ and so $j_1 + 1$ becomes the new pin. In such case we remove j_1 from the list.

As each position enters and leaves the list at most once, the time of update is linear.

6 Performing consistency checks: slow but easy

We need to efficiently perform two operations: appending a letter to the current text $A'[1 \mathinner{.\,.} n]$ and checking if two fragments of the prefix read so far are the same. First we show how to implement both of them using randomisation so that the expected running time is $\mathcal{O}(\log n)$. In the next section we improve the running time to deterministic $\mathcal{O}(1)$.

We use the standard labeling technique [16], assigning unique small names to all fragments of lengths that are powers of two. More formally, let $name[i][j]$ be an integer from $\{1, \ldots, n\}$ such that $name[i][j] = name[i'][j]$ if and only if $A'[i..i + 2^j - 1] = A'[i' \mathinner{.\,.} i' + 2^j - 1]$. Then checking if any two fragments of A' are the same is easy: we only need to cover both of them with fragments which are of the same length 2^j, where 2^j is the largest power of two not exceeding their length. Then we check if the corresponding fragments of length 2^j are the same in constant time using the previously assigned names.

Appending a new letter $A'[n+1]$ is more difficult, as we need to compute $name[n - 2^j + 2][j]$ for all $j = 1, \ldots, \log n$. We set $name[n+1][0]$ to $A'[n+1]$. Computing other names is more complicated: we need to check if a given fragment of text $A'[n - 2^j + 2 \mathinner{.\,.} n+1]$ occurs at some earlier position, and if so, choose the same name. To locate the previous occurrences, for all $j > 0$ we keep a dictionary $M(j)$ mapping pair $(name[i][j-1], name[i + 2^{j-1}][j-1])$ to $name[i][j]$. To check if a given fragment $A'[n - 2^j + 2 \mathinner{.\,.} n+1]$ occurs previously in the text, we look up the pair $(name[n - 2^j + 2][j-1], name[n - 2^{j-1} + 2][j-1])$ in $M(j)$. If there is such an element in $M(j)$, we set $name[n - 2^j + 2][j]$ equal to the corresponding name. Otherwise we set $name[n - 2^j + 2][j]$ equal to the size of $M(j)$ plus 1. Note that the new name is the smallest integer which we have not assigned as a name of fragment of length 2^j yet. Then we update the dictionary accordingly: insert the mapping from $(name[n-2^j+2][j-1], name[n-2^{j-1}+2][j-1])$ to the newly added element.

To implement the dictionaries $M(j)$, we use dynamic hashing with a worst-case constant time lookup and amortized expected constant time for updates (see [7] or a simpler variant with the same performance bounds [21]). Then the running time of the whole algorithm becomes expected $\mathcal{O}(n \log n)$, as there are $\log n$ dictionaries, each running in expected linear time. The expectation is taken over the random choices of the algorithm.

7 Size of the alphabet

It is known that VALIDATE-π not only answers whether the input table is a valid border array, but also returns the minimum size of the needed alphabet. We show, that this is in fact true also for VALIDATE-π'. Roughly speaking, VALIDATE-π' runs VALIDATE-π and can just return its answers. To this end we shwo that the minimum alphabet size required by the fixed prefix of A matches the minimum alphabet size required by A'.

Lemma 6 *Let $A'[1\,.\,.\,n]$ be a valid π' function, $A[1\,.\,.\,n+1]$ be the maximal function consistent with $A'[1\,.\,.\,n]$, and i be the pin. The minimum alphabet size required by $A'[1\,.\,.\,n]$ equals the minimum alphabet size required by $A[1\,.\,.\,i-1]$ if $A[i] > 0$, and by $A[1\,.\,.\,i]$ if $A[i] = 0$.*

Proof Suppose first that $A[i] > 0$. Thus VALIDATE-π run on $A[1\,.\,.\,n]$ returns the same size of required alphabet as run on $A[1\,.\,.\,i-1]$ since new letters are needed only when $A[j] = 0$ at some position, see Val3, and $A[j] > 0$ for j on the last slope. Consider any $B[1\,.\,.\,n+1]$ consistent with $A'[1\,.\,.\,n]$. Then by Lemma 1 $B[1\,.\,.\,i-1] = A[1\,.\,.\,i-1]$. Clearly VALIDATE-$\pi(B[1\,.\,.\,i-1])$ reports the size of required alphabet no larger than VALIDATE-$\pi(B[1\,.\,.\,n])$, as the needed alphabet only increases with n. Thus A does not require an alphabet larger than B.

Suppose now that $A[i] = 0$. Then, for any $B[1\,.\,.\,n+1]$ consistent with $A'[1\,.\,.\,n]$,

$$0 \leq B[i] \leq A[i] = 0$$

holds by CF3, i.e., $A[1\,.\,.\,i] = B[1\,.\,.\,i]$. Since $A[j] > 0$ for $j > i$, the same argument as previously works. $\square$

Note, that VALIDATE-π' runs VALIDATE-π on $A[1\,.\,.\,i-1]$ only. But running it on $A[i]$ is needed only when $A[i] = 0$ and this can be safely performed, as such $A[i]$ cannot be altered later (it cannot further decrease), so VALIDATE-π can be run on $A[i]$ as soon as it reaches 0.

We further note that Lemma 6 implies that the minimum size of the alphabet required for a valid strict border array is at most as large as the one required for border array. The latter is known to be $\mathcal{O}(\log n)$ [19, Th. 3.3a]. These two observations imply the following.

Corollary 1 *The minimum size of the alphabet required for a valid strict border array is $\mathcal{O}(\log n)$.*

8 Improving the running time to linear

To improve the running time we only need to show how to perform consistency checks more efficiently. A natural approach is as follows: construct a suffix tree [10, 18, 23] for the input table $A'[1\..n]$, together with a data structure for answering LCA queries [3]. The best known algorithm for constructing the suffix tree runs in linear time, regardless of the size of the alphabet [10]. Unfortunately, this algorithm, and all other linear time solutions we are aware of, are inherently off-line, and as such invalid for our purposes. The online suffix tree constructions of [18, 23] have a slightly bigger running time of $\mathcal{O}(n \log |\Sigma|)$, where Σ is the used alphabet. As A' is a text over an alphabet $\{-1, 0, \ldots, n-1\}$, i.e., of size $n+1$, the known online suffix tree constructions would take $\mathcal{O}(n \log n)$ time.

To get a linear time algorithm we exploit both the structure of the π' array and the relationship between subsequent consistency checks. In more detail, firstly we demonstrate how to improve Ukkonen's algorithm [23] so that it runs in time $\mathcal{O}(n)$ for alphabets of polylogarithmic size. This alone is still not enough, since A' is over an alphabet of linear size. To overcome this obstacle we use the combinatorial properties of A' to compress it. The compressed table uses alphabet of polylogarithmic size, which makes the improved version of the Ukkonen's algorithm applicable. New problems arise, as the compressed table is a little harder to read and further conditions need to be verified to answer the consistency checks.

8.1 Suffix trees for polylogarithmic alphabet

In this section we present a construction of an online dictionary with constant time access and insertion, for $t = \log n$ elements. When used in Ukkonen's algorithm [23], it guarantees the following construction of suffix treees.

Lemma 7 *For any constant c, the suffix tree for a text of length n over an alphabet of size* $\log^c n$ *can be constructed on-line in* $\mathcal{O}(n)$ *time. Given a vertex in the resulting tree, its child labeled by a specified letter can be retrieved in constant time.*

The only reason Ukkonen's algorithm [23] does not work in linear time is that given a vertex it needs to efficiently retrieve its child labeled with a specified letter. If we are able to perform such retrieval in constant time, the Ukkonen's algorithm runs in linear time.

For that we can use the atomic heaps of Fredman and Willard [12], which allow constant time search and insert operations on a collection of $\mathcal{O}(\sqrt{\log n})$-elements sets. This results in a fairly complicated structure, which can be greatly simplified since in our case not only are the sets small, but the size of the universe is bounded as well.

Simplifying assumptions We assume that the value of $\lfloor \log n \rfloor$ is known. Since n is not known in the advance when we read elements of A' one-by-one, as soon as the value of n doubles, we repeat the whole computation with a new value of $\lfloor \log n \rfloor$. This changes the running time only by a constant factor.

It is enough to give the construction for the alphabet of size $\log n$ as for alphabets of size $\log^c n$ we can encode each letter in c characters chosen from an alphabet of a logarithmic size.

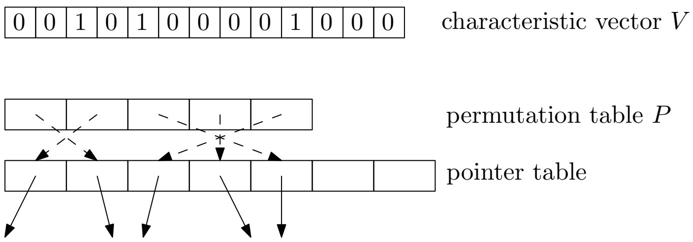

Fig. 5 Basic structure for succinct suffix tree.

First step: dictionary for small number of elements We implement an online dictionary for an universe of size $\log n$. Both access and insert time are constant and the memory usage is at most linear in the number of elements stored. The first step of the construction is a simpler case of t keys, for $t \leq \sqrt{\log n}$. Then this construction is folded twice to obtain the general case of $t = \Theta(\log n)$.

The indices of items currently present in the dictionary are encoded in one machine word, called the *characteristic vector* V, in which the bit $V[i] = 1$ if and only if dictionary contains key i.

We store pointer to the keys in the dictionary in a dynamically resized *pointer table*, in order of their arrival times: whenever we insert a new item, its pointer is put right after the previously added one. Additionally, we keep a *permutation table* P that encodes the order in which currently stored elements have been inserted. In other words, $P[i]$ stores the position in the pointer table of the pointer to i. Since $t \leq \sqrt{\log n}$, all successive values of such permutation can be stored in just one machine word.

Accessing the information for small number of elements If we want to find the pointer to the element number k, we first check if $V[k] = 1$. Then we find the index of k, i.e., $j = \#\{k' \leq k \ : \ V[k'] = 1\}$. To do this, we mask out all the bits on positions larger than k, obtaining vector V'. Then $j = \#\{k' \ : \ V'[k'] = 1\}$. Computing j can be done comparing V' with the precomputed table. Then we look at position j in the permutation table — $P[j]$ gives address in the pointer table under which the pointer to k is stored. This gives us the desired key.

The precomputed tables can be obtained using standard techniques as well as deamortised in a standard way.

Updating the information for small number of elements When a new key k arrives, it is stored in the memory and a pointer to it is put in the dictionary: firstly we set $V[k] = 1$ and insert the pointer on the last position at the pointer table. We also need to update the permutation table. To do this, we calculate $j = \#\{k' < k \ : \ V[k'] = 1\}$ and $n = \#\{k' \ : \ V[k'] = 1\}$, this is done in the same way as when accessing the stored pointer. Then we change the permutation table: we move all the numbers on positions greater than j one position higher and write n on position j. Since the whole permutation table fits in one code-word, this can be done in constant time: let P' be the table P with all positions larger than $j - 1$ masked out and P'' the table with all position smaller than j masked out. Then we shift P'' by one position higher and set $P \leftarrow P'|P''$. Then we set $P[j] = n$.

Larger number of elements When the number of items becomes bigger, we fold the above construction twice (somehow resembling the B-tree of order $t = \sqrt{\log n}$): choose a subset of keys $k_1 < k_2 < \cdots < k_\ell$ such that between k_j and k_{j+1} there are at least t and at most $2t$ other keys. Observe that $k_1 < k_2 < \cdots < k_\ell$ can be kept in the above structure, with constant update and access time, we refer to it as the *top* structure. Moreover, for each i the keys between k_i and k_{i+1} also can be kept in such a structure. We refer to those structures as the *bottom* structures.

Access for large number of elements To access information associated with a given key k, we first look up the largest chosen key smaller than k in the top structure and then look up k in the corresponding bottom structure. The second operation is already known to have constant amortised time. The first operation can be done in $\mathcal{O}(1)$ time by first masking out the bits on positions larger than k in top characteristic vector and then extracting the position of the largest bit. Again this can be done using standard techniques.

Update for large number of elements When we insert new item k, firstly we find i such that $k_{i-1} \leq k < k_i$, where k_{i-1} and k_i are elements of the top structure. This is done in the same way as when information on k is accessed. Then k is inserted into proper bottom structure.

It can happen that after such insertion the bottom structure has too many, that is $2t + 1$, elements. In such a case we choose its middle element, insert it into the top structure and split the keys into two parts consisting of t elements and create two new bottom structures out of them. This requires $\mathcal{O}(t)$ time but a simple analysis shows that the amortised insertion time is $\mathcal{O}(1)$: the size of the bottom structure is t after the split and $2t$ before the next split, so we can charge the cost to the new t keys inserted into the tree before the splits.

8.2 Compressing A'

Lemma 7 does not apply to A' directly, as it may hold too many different values. To overcome this obstacle we compress A' into *Compress*(A'), so that the resulting text is over a polylogarithmic alphabet and checking equality of two fragments of A' can be performed by looking at the corresponding fragments of *Compress*(A'). To compress A', we scan it from left to right. If $A'[i] = A'[i-j]$ for some $1 \leq j \leq \log^2 n$ we output $\#_0 j$. If $A'[i] \leq \log^2 n$ we output $\#_1 A'[i]$. Otherwise we output the binary encoding of $A'[i]$ enclosed by $\#_2$ and $\#_3$. For each i we store the position of its encoding in *Compress*(A') in *Start*$[i]$.

The above compression outputs non-constant number of elements only in the last case, i.e., when $A'[i] > \log^2 n$ and $A'[i]$ does not occur in $A'[i - \log^2 n \mathinner{.\,.} i-1]$. We show that the number of different large values of π' is small, which allows bounding the size of the mentioned problematic cases by $\mathcal{O}(n)$; therefore the size of the whole table *Compress*(A') can be also bounded by by $\mathcal{O}(n)$.

Lemma 8 *Let $k \geq 0$ and consider a segment of 2^k consecutive entries in the π' array. At most 48 different values from the interval $[2^k, 2^{k+1})$ occur in such a segment.*

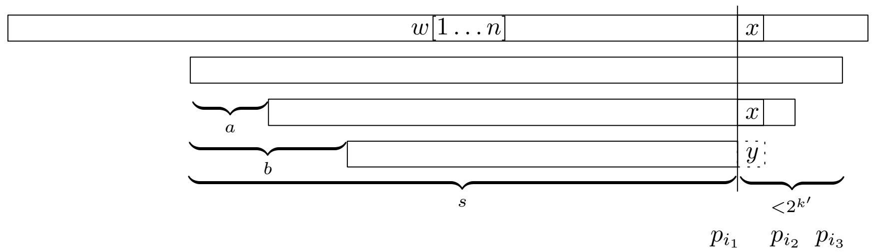

Fig. 6 Proof of Lemma 8, decreasing sequence.

Proof First note that each i such that $\pi'[i] > 0$ corresponds to a non-extensible occurrence of the border $w[1\,..\,\pi'[i]]$, i.e., $\pi'[i]$ is the maximum j such that $w[1\,..\,j]$ is a suffix of $w[1\,..\,i]$ and $w[j+1] \neq w[i+1]$.

If $k < 2$ then the claim is trivial. So let $k' = k - 2 \geq 0$ and assume that there are more than 48 different values from $[4 \cdot 2^{k'}, 8 \cdot 2^{k'})$ occurring in some segment of length 2^k. Then more than 12 different values from $[4 \cdot 2^{k'}, 8 \cdot 2^{k'})$ occur in a segment of length $2^{k'}$. Split the range $[4 \cdot 2^{k'}, 8 \cdot 2^{k'})$ into three subranges $[4 \cdot 2^{k'}, 5 \cdot 2^{k'})$, $[5 \cdot 2^{k'}, 6 \cdot 2^{k'})$ and $[6 \cdot 2^{k'}, 8 \cdot 2^{k'})$. Then at least 5 different values from one of these subranges occur in the segment; let $[\ell, r)$ be that subrange. Note that (no matter which one it is),

$$r - \ell \leq \frac{1}{2}\ell - 2^{k'} \ .$$

Let these 5 different values occur at positions $p_1 < \cdots < p_5$. Consider the sequence $p_i - \pi'[p_i] + 1$ for $i = 1, \ldots, 5$: these are the beginnings of the corresponding non-extensible borders. In particular p_i's are pairwise different. Each sequence of length 5 contains a monotone subsequence of length 3. We consider the cases of decreasing and increasing sequence separately:

1. There exist $p_{i_1} < p_{i_2} < p_{i_3}$ in this segment such that

$$p_{i_1} - \pi'[p_{i_1}] + 1 > p_{i_2} - \pi'[p_{i_2}] + 1 > p_{i_3} - \pi'[p_{i_3}] + 1 \ .$$

Define $x = w[p_{i_1} + 1]$ and $y = w[\pi'[p_{i_1}] + 1]$, see Fig. 6. Then by the definition of $\pi'[p_{i_1}]$, $x \neq y$. We derive a contradiction by showing that $x = y$. To this end we use the periodicity of the word w. Define

$$\begin{aligned} a &= (p_{i_2} - \pi'[p_{i_2}] + 1) - (p_{i_3} - \pi'[p_{i_3}] + 1) \ , \\ b &= (p_{i_1} - \pi'[p_{i_1}] + 1) - (p_{i_3} - \pi'[p_{i_3}] + 1) \ , \\ s &= \pi'[p_{i_1}] + b \ , \end{aligned}$$

see Fig. 6. Define $s = \pi'[p_{i_1}] + b$, see Fig. 6; then both a, b are periods of $w[1\,..\,s]$, see Fig. 6. We show that $a, b \leq \frac{s}{2}$ and so periodicity lemma can be applied to them and word $w[1\,..\,s]$.

$$\begin{aligned} a < b &= (p_{i_1} - \pi'[p_{i_1}]) - (p_{i_3} - \pi'[p_{i_3}]) \\ &< \pi'[p_{i_3}] - \pi'[p_{i_1}] \\ &\leq r - \ell \\ &\leq \frac{\ell}{2} \ . \end{aligned}$$

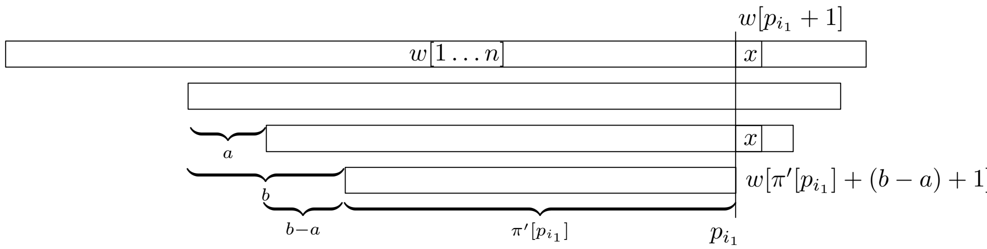

Fig. 7 An illustration of equality $w[p_{i_1}+1] = w[\pi'[p_{i_1}]+(b-a)+1]$.

Since $s = \pi'[p_{i_1}] + b$ and $\pi'[p_{i_1}] \in [\ell, r)$, $s \geq \ell$. Thus

$$a < b \leq \frac{s}{2} \ .$$

By periodicity lemma $b - a$ is also a period of $w[1\,..\,s]$. As position $p_{i_1}+1$ is covered by the non-extensible border ending at p_{i_2}:

$$x = w[p_{i_1}+1] = w[\pi'[p_{i_1}] + 1 + (b-a)] \ ,$$

see Fig. 7. Note that

$$\pi'[p_{i_1}] + 1 + (b-a) \leq \pi'[p_{i_1}] + b = s$$

and so $w[\pi'[p_{i_1}] + 1 + (b-a)]$ is a letter from word $w[1\,..\,s]$, which has a period $b-a$. Hence

$$x = w[\pi'[p_{i_1}] + 1 + (b-a)] = w[\pi'[p_{i_1}] + 1] = y \ ,$$

contradiction.

2. There exist $p_{i_1} < p_{i_2} < p_{i_3}$ in this segment such that

$$p_{i_1} - \pi'[p_{i_1}] + 1 < p_{i_2} - \pi'[p_{i_2}] + 1 < p_{i_3} - \pi'[p_{i_3}] + 1 \ ,$$

see Fig. 8.
By assumption $\pi'[p_{i_1}], \pi'[p_{i_2}] \geq \ell$. We identify the periods of the corresponding subwords $w[1\,..\,p_{i_1}]$ and $w[1\,..\,p_{i_2}]$, respectively :

$$\begin{aligned}
a &= (p_{i_2} - \pi'[p_{i_2}] + 1) - (p_{i_1} - \pi'[p_{i_1}] + 1) \ , \\
b &= (p_{i_3} - \pi'[p_{i_3}] + 1) - (p_{i_2} - \pi'[p_{i_2}] + 1) \ ,
\end{aligned}$$

as depicted on Fig. 8. We estimate their sum:

$$\begin{aligned}
a + b &= (p_{i_2} - \pi'[p_{i_2}]) - (p_{i_1} - \pi'[p_{i_1}]) + (p_{i_3} - \pi'[p_{i_3}]) - (p_{i_2} - \pi'[p_{i_2}]) \\
&= (p_{i_3} - \pi'[p_{i_3}]) - (p_{i_1} - \pi'[p_{i_1}]) \\
&= (\pi'[p_{i_1}] - \pi'[p_{i_3}]) + (p_{i_3} - p_{i_1}) \\
&\leq (r - \ell) + 2^{k'} \\
&\leq (\frac{1}{2}\ell - 2^{k'}) + 2^{k'} \\
&= \frac{\ell}{2}
\end{aligned}$$

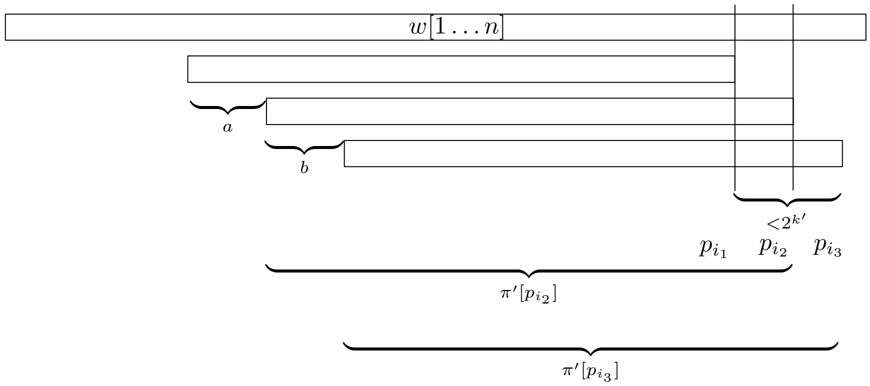

Fig. 8 Proof of Lemma 8, increasing sequence.

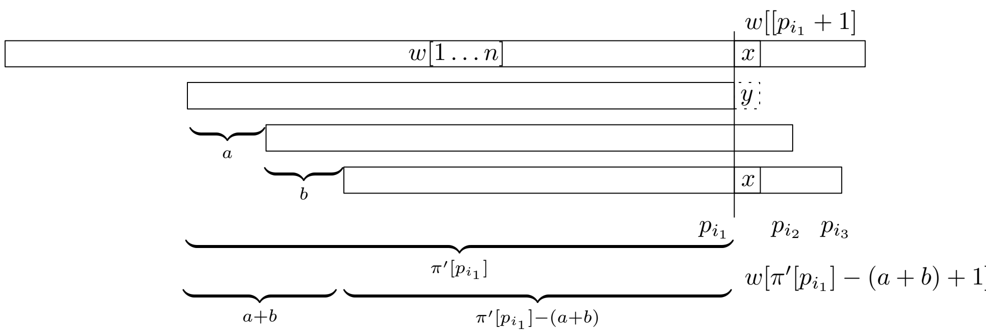

Fig. 9 Illustration of $w[(\pi'[p_{i_1}]+1)-(a+b)] = w[p_{i_1}+1]$.

Since $\ell \leq \pi'[p_{i_1}], \pi'[p_{i_2}]$, we obtain that

$$a + b \leq \frac{\pi'[p_{i_1}]}{2}, \frac{\pi'[p_{i_2}]}{2} \quad . \tag{8}$$

There are two subcases, depending on whether $\pi'[p_{i_1}] < \pi'[p_{i_2}]$ or $\pi'[p_{i_1}] > \pi'[p_{i_2}]$:

(a) $\pi'[p_{i_1}] < \pi'[p_{i_2}]$: Define $x = w[p_{i_1}+1]$ and $y = w[\pi'[p_{i_1}]+1]$, see Fig. 9. Then by definition of $\pi'[p_{i_1}]$, $x \neq y$. We obtain a contradiction by showing that $x = y$. Since the non-extensible border ending at p_{i_3} spans over position $p_{i_1}+1$, it holds that

$$x = w[(\pi'[p_{i_1}]+1)-(a+b)] \quad . \tag{9}$$

Comparing the non-extensible borders ending at p_{i_2} and p_{i_3} we deduce that b is a period of $w[1\,..\,\pi'[p_{i_2}]]$ and as $\pi'[p_{i_1}]+1 \leq \pi'[p_{i_2}]$,

$$y = w[\pi'[p_{i_1}]+1] = w[\pi'[p_{i_1}]+1-b] \quad .$$

Similarly by comparing the non-extensible prefixes ending at p_{i_1} and p_{i_2} we deduce that a is a period of $w[1\,..\,\pi'[p_{i_1}]]$. Thus

$$y = w[\pi'[p_{i_1}]+1-b] = w[\pi'[p_{i_1}]+1-b-a] \tag{10}$$

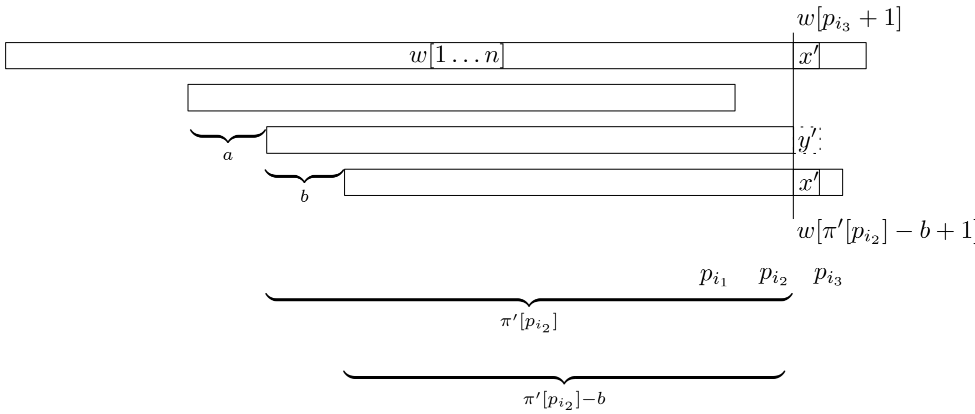

Fig. 10 Illustration of $w[(\pi'[p_{i_2}]+1)-b] = w[p_{i_3}+1]$.

and therefore by (9) and (10) $x = y$. Contradiction.

(b) $\pi'[p_{i_1}] > \pi'[p_{i_2}]$: Let $x' = w[p_{i_2}+1]$ and $y' = w[\pi'[p_{i_2}]+1]$. Then $x' \neq y'$ by the definition of $\pi'[p_{i_2}]$, see Fig. 10. We show that $x' = y'$ and hence obtain a contradiction. Since non-extensible border ending at p_{i_3} spans over position $p_{i_2}+1$, we obtain that

$$x' = w[\pi'[p_{i_2}] - b + 1] \ , \tag{11}$$

see Fig. 10. By comparing non-extensible prefixes ending at p_{i_1} and p_{i_2} we deduce that a is a period of $w[1\,..\,\pi'[p_{i_1}]]$. As $\pi'[p_{i_2}]+1 \leq \pi'[p_{i_1}]$,

$$y' = w[\pi'[p_{i_2}]+1] = w[\pi'[p_{i_2}]+1-a] \ .$$

By comparing the non-extensible prefixes ending at p_{i_2} and p_{i_3} we deduce that b is a period of $w[1\,..\,\pi'[p_{i_2}]]$ and $a + b \leq \frac{\pi'[p_{i_2}]}{2}$ by (8), it holds that

$$y' = w[\pi'[p_{i_2}]+1-a] = w[\pi'[p_{i_2}]+1-a-b] \ .$$

As a is a period of $w[1\,..\,\pi'[p_{i_2}]+1]$,

$$y' = w[\pi'[p_{i_2}]+1-a-b] = w[\pi'[p_{i_2}]+1-b] \ . \tag{12}$$

So by (11) and (12) $x' = y'$, contradiction. □

Corollary 2 *Compress(A') consists of $\mathcal{O}(n)$ symbols over an alphabet of $\mathcal{O}(\log^2 n)$ size.*

Proof To calculate the total length of the resulting text, observe that the only case resulting in a non-constant number of characters being output for a single index i is when $A'[i] > \log^2 n$ and the value of $A'[i]$ does not occur at any of $\log^2 n$ previous indices. By Lemma 8, as long as $2^k \geq \log^2 n$, any segment of consecutive $\log^2 n$ indices contains at most 48 different values from $[2^k, 2^{k+1})$. For a single k there are $\frac{n}{\log^2 n}$ such segments of length $\log^2 n$ end encoding one value of A' takes

$\log n$ characters in $\mathit{Compress}(A')$. As k takes values from $\log(\log^2(n))$ to $\log n$ the total number of characters used to describe all those values of $A'[i]$ is at most

$$\sum_{k=2\log\log n}^{\log n} 48\frac{n}{\log^2 n}\log n \in \mathcal{O}(n) \enspace ,$$

so $|\mathit{Compress}(A')| = \mathcal{O}(n)$. ◻

As the alphabet of $\mathit{Compress}(A')$ is of polylogarithmic size, the suffix tree for $\mathit{Compress}(A')$ can be constructed in linear time by Lemma 7.

8.3 Performing consistency checks on the $\mathit{Compress}(A')$

Subchecks Consider consistency check: is $A'[j\mathinner{..}j{+}k] = A'[i\mathinner{..}i{+}k]$, where $j = A[i]$? We first establish equivalence of this equality with equality of proper fragments of $\mathit{Compress}(A')$. Note, that $A'[\ell] = A'[\ell']$ does not imply the equality of two corresponding fragments of $\mathit{Compress}(A')$, as they may refer to previous values of A'. Still, such references can be only $\log^2 n$ elements backwards. This observation is formalised as:

Lemma 9 *Let $j = A[i]$. Then*

$$A'[j\mathinner{..}j+k] = A'[i\mathinner{..}i+k] \tag{13}$$

if and only if

$$\mathit{Compress}(A')[\mathit{Start}[j+\log^2 n]\mathinner{..}\mathit{Start}[j+k+1]-1] = \tag{14}$$
$$\mathit{Compress}(A')[\mathit{Start}[i+\log^2 n]\mathinner{..}\mathit{Start}[i+k+1]-1]$$
$$\textit{and } A'[j\mathinner{..}j+\min(k,\log^2 n)] = A'[i\mathinner{..}i+\min(k,\log^2 n)] \tag{15}$$

Proof If $k \le \log^2 n$, the claim holds trivially, as (13) and (15) are exactly the same and (14) holds vacuously.

So suppose that $k > \log^2 n$.

⇒ Suppose first that $A'[j\mathinner{..}j{+}k] = A'[i\mathinner{..}i{+}k]$. Then of course $A'[j\mathinner{..}j{+}\log^2 n] = A'[i\mathinner{..}i+\log^2 n]$, as $k > \log^2 n$ by case assumption. Thus (15) holds.

Note that $\mathit{Compress}(A')[\mathit{Start}[j+\log^2 n]\mathinner{..}\mathit{Start}[j+k+1]-1]$ is created using only $A[j\mathinner{..}j+k]$: when creating an entry corresponding to $A'[\ell]$ we can refer to $A'[\ell]$ and to at most $\log^2 n$ elements before it. Similarly, $\mathit{Compress}(A')[\mathit{Start}[i+\log^2 n]\mathinner{..}\mathit{Start}[i{+}k{+}1]{-}1]$ is created using $A'[i\mathinner{..}i{+}k]$ exclusively. Since $A[j\mathinner{..}j{+}k] = A[i\mathinner{..}i+k]$, both fragments of $\mathit{Compress}(A')$ are used using the same input, and so they are equal. Thus (14) holds, which ends the proof in this direction.

⇐ Assume that (14) and (15) hold. We show by a simple induction on ℓ, that that $A'[i+\ell] = A'[j+\ell]$. For $\ell \le \log^2 n$ the claim is trivial, as it is explicitly stated in (15). So let $\ell \ge \log^2 n$. Consider $\mathit{Compress}(A')[\mathit{Start}[i+\ell]\mathinner{..}\mathit{Start}[i+\ell+1]-1]$ and $\mathit{Compress}(A')[\mathit{Start}[j+\ell]\mathinner{..}\mathit{Start}[j+\ell+1]-1]$, they are known to be equal by the assumption.

- If they are both equal to $\#_0 m$ then $A'[i+\ell] = A'[i+\ell-m]$ and $A'[j+\ell] = A'[j+\ell-m]$; by the inductive assumption $A'[i+\ell-m] = A'[j+\ell-m]$, which ends the case.

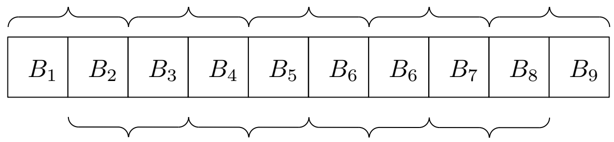

Fig. 11 Scheme of ranges for suffix trees.

– If they are both equal to $\#_1 m$ then $A'[i+\ell] = A'[j+\ell] = m$.
– If they are equal to $\#_2 m_1 \dots m_z \#_3$ then $m_1 \dots m_z$ encode m in binary and $A'[i+\ell] = A'[j+\ell] = m$, which ends the last case. $\square$

Similarly as in the Section 8.1, we assume that $\lfloor \log n \rfloor$ is known. In the same way we repeat the whole computation from the scratch as soon as it value changes. This increases the running time by a constant factor.

We call the checks of the form (14) the *compressed consistency checks*, checks of the form (15) the *short consistency checks* and the *near short consistency checks* when moreover $|i-j| < \log^2 n$.

The compressed consistency checks can be answered in amortised constant time using LCA query [3] on the suffix tree built for *Compress*(A'). What is left is to show how to perform short consistency checks in amortised constant time as well.

Performing near short consistency checks To do near short consistency checks efficiently, we split A' into blocks of $\log^2 n$ consecutive letters: $A' = B_1 B_2 \dots B_k$, see Fig 11. Then we build suffix trees for each pair of consecutive blocks, i.e., $B_1B_2, B_2B_3, \dots, B_{k-1}B_k$. Each block contains at most $\log^2 n$ values smaller than $\log^2 n$, and at most $48 \log n$ larger values, by Lemma 8, so all the suffix trees can be built in linear time by Lemma 7. For each tree we also build a data structure supporting constant-time LCA queries [3]. Then, any near short consistency check reduces to an LCA query in one of these suffix trees. Note, that such a query gives also gives the actual length of the longest prefix of the two compared strings; this is used in performing short consistency checks.

Performing short consistency checks short consistency checks are answered by near short consistency checks and naive letter-to-letter comparisons. To obtain linear total time, the results of previous short consistency checks are reused as follows. We store the value j_{best} for which the length L of the common prefix of $A'[j \mathinner{.\,.} j+\log^2 n]$ and $A'[i \mathinner{.\,.} i+\log^2 n]$ is relatively long as well as L itself. To be precise, the values j_{best} and L satisfy the following invariants:

$$L \le k \tag{16}$$

$$j \le j_{best} \le j + \log^2 n \tag{17}$$

$$A'[j_{best} \mathinner{.\,.} j_{best} + L - 1] = A'[i \mathinner{.\,.} i + L - 1] \tag{18}$$

$$\text{if } j \ne j_{best} \text{ then } A'[j \mathinner{.\,.} j + L - 1] \ne A'[i \mathinner{.\,.} i + L - 1] \tag{19}$$

When another short consistency check is done we first compute the common prefix of $A'[j \mathinner{.\,.} j+\log^2 n]$ and $A'[j_{best} \mathinner{.\,.} j_{best}+\log^2 n]$ and compare it with L. If it is smaller, then clearly the common prefix of $A'[j \mathinner{.\,.} j+\log^2 n]$ and $A'[i \mathinner{.\,.} i+\log^2 n]$ is smaller than L; if it equals L, then we naively check if $A'[j + L + k] = A'[i + L + k]$ for consecutive k.

Consecutive short consistency check Before we state how short consistency check is performed, we investigate the relation between two consecutive short consistency checks. To simplify the presentation, we assume that the adjusting of the last slope is done in a slightly different way, than written in the code of VALIDATE-π': if the pin is assigned value i', firstly $A[i']$ is set to $A[i]+(i'-i)$, i.e., to its current implicit value, then it is verified if $A'[i' \mathinner{..} n] = A'[A[i'] \mathinner{..} A[i'] + (n - i')]$ (and the result ignored) and only after that $A[i']$ is assigned valid value for $\pi[i']$. Such a change can only increases the running time of the algorithm.

Consider two consecutive short consistency checks. We refer to the previous short consistency check as $A'[j \mathinner{..} j+k] = A'[i \mathinner{..} i+k]$ and to the current as $A'[j' \mathinner{..} j'+k'] = A'[i' \mathinner{..} i' + k']$. The new short consistency check can be asked only when consistency check is performed in the following situations:

(Short1) this is a first iteration of ADJUST-LAST-SLOPE and PIN-VALUE-CHECK did not return any index in this iteration. This means that, comparing to the previous short consistency check, the values of i and j did not change: we did not move the pin, since PIN-VALUE-CHECK did not return an index; and the value of $A[i]$ was not modified yet. Thus $i' = i$, $j = j'$, though $k' \geq k$ and LINEAR-VALIDATE-π' read $k' - k$ new values of A'.

(Short2) this is not a first iteration of ADJUST-LAST-SLOPE and PIN-VALUE-CHECK did not return any index in this iteration. In this case the previous modification is assigning $A[i]$ the next candidate, so $i' = i$, $k' = k$, but $j' < j$.

(Short3) PIN-VALUE-CHECK did return an index in this iteration. In this case the short consistency check is run for the same slope, but starting from greater index. Thus $i' > i$ and $i' - i = j' - j$. Also $k' \leq k$, still $k' \geq k - (j' - j)$.

Performing short consistency checks We now present the actions of the short consistency check in one round:

Algorithm 7 SHORT-CONSISTENCY-CHECK

1: read i', j', k'
2: **if** $i' = i$ **and** $j' = j$ **and** $k' = k$ **then**
3: **return** previous answer
4: **end if**
5: **if** $k' > k$ **then** ▷ Only for Short1
6: **if** $L < k$ **then**
7: **return** NO
8: **end if**
9: **while** $A'[j_{best} + L] = A'[i' + L]$ **and** $L < k'$ **do**
10: $L \leftarrow L + 1$
11: **end while**
12: **if** $L = k'$ **then**
13: **return** YES
14: **else**
15: **return** NO
16: **end if**
17: **end if** ▷ Actions for Short1 end here
18: **if** $j' < j$ **and** $j' + \log^2 n < j_{best}$ **then** ▷ Only for Short2
19: $j_{best} \leftarrow j'$, $L \leftarrow 0$
20: **end if**
21: **if** $j' > j$ **then** ▷ Only for Short3
22: $j_{best} \leftarrow j_{best} + (j' - j)$
23: $L \leftarrow L - (j' - j)$
24: **if** $L < 0$ **then**
25: $j_{best} \leftarrow j'$, $L \leftarrow 0$
26: **end if**
27: **end if**
28: $\ell \leftarrow$ NEAR-SHORT-CONSISTENCY-CHECK$(A'[j_{best} \mathinner{..} j_{best} + k'], A'[j' \mathinner{..} j' + k'])$
29: **if** $\ell < L$ **then**
30: **return** NO
31: **else**
32: $j_{best} \leftarrow j'$
33: **end if**
34: **while** $A'[j' + L] = A'[i' + L]$ **and** $L \leq k'$ **do**
35: $L \leftarrow L + 1$
36: **end while**
37: **if** $L = k'$ **then**
38: **return** YES
39: **else**
40: **return** NO
41: **end if**
42: $i \leftarrow i'$, $j \leftarrow j'$, $k \leftarrow k'$

We show by induction that the (16)–(19) are preserved by SHORT-CONSISTENCY-CHECK. The claim is shown for groups of actions of the SHORT-CONSISTENCY-CHECK. As in the algorithm, let x' describe the value from the current round and x from the previous round.

Lemma 10 *The algorithm* SHORT-CONSISTENCY-CHECK *preserves* (16)–(19).

Proof We consider each of the conditions separately.

(16) Note first, that $L = 0$ always satisfies (16), so we do not deal with it. Let us inspect the code inside lines 5–17. By the induction assumption, $L < k$ and by case asssumption $k < k'$. So we need only to consider the increases of L by 1 in line 10. But it is performed only when $L < k'$.

In lines 18–20 $k' = k$ and therefore the claim holds by the induction assumption. In lines 21–27 k' is decreased, but $k' \geq k - (j' - j)$, as noted in Short3, and $L \leftarrow L - (j' - j)$. So the claim holds by the induction assumption.
L can increased by 1 in line 35, but this is performed only when $L < k'$.

(17) There are two parts of the inequality. The first states that $j' \leq j_{best}$. If j_{best} is set to j' or $j' < j$ this is preserved.
In lines 21–27 we consider the cases when $j' > j$. But there $j_{best} \leftarrow j_{bset} + (j' - j)$, so the inequality is preserved by the induction assumption.
Let us look at the right inequality of (17). In lines 18–20 it is checked and restored. In lines 21–27 $j_{best} \leftarrow j_{best} + (j' - j)$, so the inequality holds by the induction assumption. Then possibly j_{best} is set to j in some lines, which is also fine.

(18) We need to inspect the steps, in which L, j_{best}, k or i are changed.
In lines 5–17 $i = i'$, as noted in Short1 and so be the inductive assumption $A'[j_{best} \mathinner{..} j_{best} + L - 1] = A'[i' \mathinner{..} i' + L - 1]$. Then L is increased (by 1) only when $A'[j_{best} + L] = A'[i' + L]$.
In lines 18–20 L is assigned 0, which is always a proper value.
In lines 21–27 $j'_{best} = j_{best} - (i' - i)$ or $j'_{best} = j'$. The latter enforces $L' = 0$, which trivially satisfies (18). So suppose that $j'_{best} = j_{best} - (i' - i)$. Then $A[i \mathinner{..} i + L - 1] = A[j_{best} \mathinner{..} j_{best} + L - 1]$ holds by the case assumption Then $A[i' \mathinner{..} i' + L' - 1] = A[i' \mathinner{..} i + L - 1]$ and $A[j' \mathinner{..} j' + L' - 1] = A[jbest' \mathinner{..} jbest + L - 1]$, therefore $A[i' \mathinner{..} i' + L' - 1] = A[j'_{best} \mathinner{..} j'_{best} + L' - 1]$ trivially holds.
If in line 32 j_{best} is changed, it means that the common prefix of $A[j' \mathinner{..} j' + k']$ and $A[j'_{best} \mathinner{..} j'_{best} + k']$ are of length L. But we already have shown in the proof that $A'[j_{best} \mathinner{..} j_{best} + L - 1] = A'[i' \mathinner{..} i' + L - 1]$. Since $k' \geq L$, $A[j' \mathinner{..} j' + k'] = A'[i' \mathinner{..} i' + L - 1]$, as needed.
In line 35 we already know from the proof that $A'[j_{best} \mathinner{..} j_{best} + L - 1] = A'[i \mathinner{..} i + L - 1]$. Then L is increased by 1 only when $A'[j_{best} + L] = A'[i + L]$ and so (18) is preserved.

(19) We show the contraposition: if

$$A'[i' \mathinner{..} i' + L - 1] = A'[j' \mathinner{..} j' + L - 1] \tag{20}$$

then $j' = j_{best}$. We look at the place at which the algorithm returns an answer. Assume that the round ended before line 17. This means that $i' = i$ and $j' = j$, as noted in Short1. Also, L can only increase before line 17. Thus if (20) holds, it held also in the previous round. Thus by induction assumption for (19), $j_{best} = j = j'$.
Assume that the round ended in line 30 and that (20) holds. Since (18) holds:

$$A'[j'_{best} \mathinner{..} j'_{best} + L - 1] = A'[i' \mathinner{..} i' + L - 1]$$

thus

$$A'[j'_{best} \mathinner{..} j'_{best} + L - 1] = A'[j' \mathinner{..} j' + L - 1] \ ,$$

contradiction, as this means that $\ell = L$ and line 30 is not executed.
Otherwise line 32 is executed and $j_{best} = j'$. □

Lemma 11 *Answering short consistency checks can be done in $\mathcal{O}(n)$ time.*

Proof Let Δx denote the change of the value of x in a single step of the algorithm.

The proof uses amortised analysis. For every increase Δn, we get $2\Delta n$ units of credit. For every change of j we get $3|\Delta j|$ units of credit. We define a potential of the configuration as

$$p = k - L + (j_{best} - j) \ .$$

We use the credit to pay for near short consistency checks, letter-by-letter comparisons, and for the change of the potential. In the following we consider each possible action of the algorithm. Let c be the amount of released credit and s the cost of comparisons. We show that $c - s - \Delta p \geq 0$.

First note, that among three blocks in the algorithm: 5–17, 18–20 and 21–27, exactly one is executed, as they correspond to different cases among Short1–Short3. Moreover, 5–17 is not followed by any execution whatsoever. We show that after 5–17 the change of credit is non-negative, while execution of any other two blocks results in a left over of some credit. This credit is then used in the rest of the algorithm.

Consider first lines 5–17. Since $k' > k$, it means that a new value $A'[n']$ was read and so $n' = n + \Delta k$. So $2\Delta k$ units of credit are released. The comparisons $A'[j_{best} + L]$ with $A'[i' + L]$ cost at most ΔL, which is at most Δk, as they are performed only if $L = k$ and L is not increased above k'. The increase of the potential is:

$$\begin{aligned}\Delta p &= \Delta k - \Delta L + \Delta j_{best} - \Delta j \\ &\leq \Delta k - \Delta k + 0 - 0 \\ &= 0 \ .\end{aligned}$$

So the released credit is enough to pay for the comparisons.

Note, that in the rest of the algorithm k can only decrease, which decreases the potential. So we may assume that it does not change.

Consider lines 18–20. We estimate the change of the potential. If the line 19 was executed, then

$$\begin{aligned}\Delta p &= -\Delta L + \Delta j_{best} - \Delta j \\ &\leq -(-L) - \log^2 n - \Delta j \\ &\leq \log^2 n - \log^2 n - \Delta j \\ &= -\Delta j \ .\end{aligned}$$

If it was not then

$$\begin{aligned}\Delta p &= -\Delta L + \Delta j_{best} - \Delta j \\ &= -0 + 0 - \Delta j \\ &= -\Delta j \ .\end{aligned}$$

On the other hand there is $-3\Delta j$ credit released, so we are left with $-2\Delta j > 0$ credit.

The last block is lines 21–27. et us estimate the change of the potential:

$$\Delta p = -\Delta L + \Delta j_{best} - \Delta j \ .$$

We deal with each value separately. L is decreased by Δj and then perhaps increased (to 0), so $-\Delta L \leq \Delta j$. j_{best} is increased by Δj and then perhaps decreased, so $\Delta j_{best} \leq \Delta j$. Therefore

$$\begin{aligned}\Delta p &= -\Delta L + \Delta j_{best} - \Delta j \\ &\leq \Delta j + \Delta j - \Delta j \\ &= \Delta j \ .\end{aligned}$$

There is at least $3\Delta j$ credit released, so we are left with at least $2\Delta j > 0$ credit.

In lines 28–33 we pay 1 unit for asking the near short consistency check. The potential may only decrease, if j_{best} decreases. So the total cost is at most 1.

In lines 34–41 the cost of letter-by-letter comparison is most $\Delta L + 1$. Moreover, as there are no changes to j_{best} and j, the potential changes by

$$\Delta p = -\Delta L \ .$$

So the cost and potential change is $(\Delta L + 1) + (-\Delta L) = 1$.

Since lines 28–41 have to be preceeded by execution of either 18–20 or 21–27, we are always able to pay. The only thing left to show is the sum of $|\Delta j|$ is $\mathcal{O}(n)$.

To this end note that j increases only when the pin i is updated as $j = A[i]$. Moreover, it increases by at most Δi:

$$\Delta j = j' - j = A[i'] - A[i] \leq (A[i] + (i' - i)) - A[i] = \Delta i \ .$$

Since $i \leq n$ and i only increases, its sum of increments is linear. So the total sum of increments of j is linear. Hence the sum of decrements of j is linear as well. $\square$

Running time VALIDATE-π'runs in $\mathcal{O}(n)$ time: construction of the suffix trees and doing consistency checks, as well as doing pin value checks all take $\mathcal{O}(n)$ time.

9 Remarks and open problems

While VALIDATE-π produces the word w over the minimum alphabet such that $\pi_w = A$ on-line, this is not the case with VALIDATE-π' and LINEAR-VALIDATE-π'. At each time-step both these algorithms can output a word over minimum alphabet such that $\pi'_w = A'$, but the letters assigned to positions on the last slope may yet change as further entries of A' are read.

Since VALIDATE-π' and LINEAR-VALIDATE-π' keep the function $\pi[1\,..\,n{+}1]$ after reading $A'[1\,..\,n]$, virtually no changes are required to adapt them to g validation, where $g[i] = \pi'[i-1] + 1$ is the function considered by Duval et al. [8], because $A'[1\,..\,n-1]$ can be obtained from $g[1\,..\,n]$. Running VALIDATE-π' or LINEAR-VALIDATE-π' on such A' gives $A[1\,..\,n]$ that is consistent with $A'[1\,..\,n-1]$ and $g[1\,..\,n]$. Similar proof shows that $A[1\,..\,n]$ and $g[1\,..\,n]$ require the same minimum size of the alphabet.

Two interesting questions remain: is it possible to remove the suffix trees and LCA queries from our algorithm without hindering its time complexity? We believe that deeper combinatorial insight might result in a positive answer.

Acknowledgments

This work was partially supported by Polish Ministry of Science and Higher Education under grants N N206 1723 33, 2007–2010; Łukasz Jeż was also supported under grant number N N206 4906 38 2010–2011. Artur Jeż was additionally supported under personal grant FNP START of Foundation for Polish Science.

References

1. Breslauer, D., Colussi, L., Toniolo, L.: On the comparison complexity of the string prefix-matching problem. J. Algorithms **29**(1), 18–67 (1998)
2. Clément, J., Crochemore, M., Rindone, G.: Reverse engineering prefix tables. In: Proceedings of 26th STACS, pp. 289–300 (2009)
3. Cole, R., Hariharan, R.: Dynamic lca queries on trees. In: Proceedings of SODA '99, pp. 235–244. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA (1999)
4. Crochemore, M., Hancart, C., Lecroq, T.: Algorithms on Strings. Cambridge University Press (2007)
5. Crochemore, M., Iliopoulos, C., Pissis, S., Tischler, G.: Cover array string reconstruction. In: CPM 2010, LNCS, to appear. Springer (2010)
6. Crochemore, M., Rytter, W.: Jewels of Stringology. World Scientific Publishing Company (2002)
7. Dietzfelbinger, M., Karlin, A.R., Mehlhorn, K., auf der Heide, F.M., Rohnert, H., Tarjan, R.E.: Dynamic perfect hashing: Upper and lower bounds. SIAM J. Comput. **23**(4), 738–761 (1994)
8. Duval, J.P., Lecroq, T., Lefebvre, A.: Efficient validation and construction of Knuth–Morris–Pratt arrays. Conference in honor of Donald E. Knuth (2007)
9. Duval, J.P., Lecroq, T., Lefebvre, A.: Efficient validation and construction of border arrays and validation of string matching automata. ITA **43**(2), 281–297 (2009)
10. Farach, M.: Optimal suffix tree construction with large alphabets. In: Proceedings of FOCS '97, pp. 137–143. IEEE Computer Society, Washington, DC, USA (1997)
11. Franěk, F., Gao, S., Lu, W., Ryan, P.J., Smyth, W.F., Sun, Y., Yang, L.: Verifying a border array in linear time. J. Comb. Math. Comb. Comput. **42**, 223–236 (2002)
12. Fredman, M.L., Willard, D.E.: Trans-dichotomous algorithms for minimum spanning trees and shortest paths. J. Comput. Syst. Sci. **48**(3), 533–551 (1994). DOI http://dx.doi.org/10.1016/S0022-0000(05)80064-9
13. Hancart, C.: On Simon's string searching algorithm. Information Processing Letters **47**(2), 95–99 (1993)
14. I, T., Inenaga, S., Bannai, H., Takeda, M.: Counting parameterized border arrays for a binary alphabet. In: Proc. of the 3rd LATA, pp. 422–433 (2009)
15. I, T., Inenaga, S., Bannai, H., Takeda, M.: Verifying a parameterized border array in $\mathcal{O}(n^{1.5})$ time. In: CPM 2010, LNCS, to appear. Springer (2010)
16. Karp, R.M., Miller, R.E., Rosenberg, A.L.: Rapid identification of repeated patterns in strings, trees and arrays. In: STOC '72: Proceedings of the fourth annual ACM symposium on Theory of computing, pp. 125–136. ACM, New York, NY, USA (1972). DOI http://doi.acm.org/10.1145/800152.804905
17. Knuth, D.E., Morris Jr., J.H., Pratt, V.R.: Fast pattern matching in strings. SIAM J. Comput. **6**(2), 323–350 (1977)
18. McCreight, E.M.: A space-economical suffix tree construction algorithm. J. ACM **23**(2), 262–272 (1976). DOI http://doi.acm.org/10.1145/321941.321946
19. Moore, D., Smyth, W.F., Miller, D.: Counting distinct strings. Algorithmica **23**(1), 1–13 (1999)
20. Morris Jr., J.H., Pratt, V.R.: A linear pattern-matching algorithm. Tech. Rep. 40, University of California, Berkeley (1970)
21. Pagh, R., Rodler, F.F.: Cuckoo hashing. J. Algorithms **51**(2), 122–144 (2004)
22. Simon, I.: String matching algorithms and automata. In: Results and Trends in Theoretical Computer Science, *LNCS*, vol. 812, pp. 386–395. Springer (1994)
23. Ukkonen, E.: On-line construction of suffix trees. Algorithmica **14**(3), 249–260 (1995). URL `http://citeseer.ist.psu.edu/ukkonen95line.html`