Theory Comput Syst (2014) 54:337–372
DOI 10.1007/s00224-013-9522-8

Validating the Knuth-Morris-Pratt Failure Function, Fast and Online

Paweł Gawrychowski · Artur Jeż · Łukasz Jeż

Published online: 6 December 2013

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 (Duval et al. in Conference in honor of Donald E. Knuth, 2007), 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 pattern matching algorithms attracted much attention since the dawn of computer science. It was particularly interesting, whether a linear-time algorithm for this

P. Gawrychowski · A. Jeż
Max Planck Institute for Computer Science, Saarbrücken, Germany

P. Gawrychowski · A. Jeż (✉) · Ł. Jeż
Institute of Computer Science, University of Wrocław, Wrocław, Poland
e-mail: aje@cs.uni.wroc.pl

P. Gawrychowski
e-mail: gawry@cs.uni.wroc.pl

Ł. Jeż
e-mail: lje@cs.uni.wroc.pl

Ł. Jeż
Blavatnik School of Computer Science, Tel Aviv University, Tel Aviv, Israel

problem exists. First results were obtained by Matiyasevich for a fixed pattern in the Turing Machine model [18]. However, the first fully linear time pattern matching algorithm is the Morris-Pratt algorithm [21], which is designed for the RAM machine model, and is well known for its beautiful concept. It simulates the minimal DFA recognizing $\Sigma^* p$ (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 [23], and further improvements are known [1, 13]. 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, 5]. On the other hand, even Simon's algorithm (i.e., the very first improvement) deals with periods of pattern 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. [8] is known. Validation of border arrays is used by algorithms generating all valid border arrays [9, 11, 20].

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.

Table 1 Functions π and π' for a word *aabaabaaabaabaac*

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	8	0
$\pi[i]$	−1	1	−1	−1	1	−1	−1	5	1	−1	−1	1	−1	−1	8	0

Algorithm 1 COMPUTE-$\pi(w)$

```
1:  $\pi[1] \leftarrow 0, k \leftarrow 0$
2:  **for** $i \leftarrow 2$ to $n$ **do**
3:      **while** $k > 0$ and $w[k+1] \neq w[i]$ **do**
4:          $k \leftarrow \pi[k]$
5:      **end while**
6:      **if** $w[k+1] = w[i]$ **then**
7:          $k \leftarrow k+1$
8:      **end if**
9:      $\pi[i] \leftarrow k$
10: **end for**
```

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 Sect. 2.

Then we improve VALIDATE-π' 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].

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 [6].

Algorithm 2 π'-FROM-$\pi(\pi)$

```
1: $\pi'[0] \leftarrow -1$
2: **for** $i \leftarrow 1$ **to** $n-1$ **do**
3:     **if** $\pi[i+1] = \pi[i]+1$ **then**
4:         $\pi'[i] \leftarrow \pi'[\pi[i]]$
5:     **else**
6:         $\pi'[i] \leftarrow \pi[i]$
7:     **end if**
8: **end for**
9: $\pi'[n] \leftarrow \pi[n]$
```

Algorithm 3 π-FROM-$\pi'(\pi')$

```
1: $\pi[n] \leftarrow \pi'[n]$
2: **for** $i \leftarrow n-1$ **downto** 1 **do**
3:     $\pi[i] \leftarrow \max\{\pi'[i], \pi[i+1]-1\}$
4: **end for**
```

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* $w[i]w[i+1]\dots w[j]$ 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 $i = 1, 2, \dots, n$, see Table 1. It is known that π_w table can be computed in linear-time, see Algorithm 1.

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 $k \geq 0$.

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, see Algorithm 2 and Algorithm 3. 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

Algorithm 4 VALIDATE-$\pi(A)$

```
if $A[1] \neq 0$ then
    error $A$ is not valid at 1
end if
$cand[1] \leftarrow \{0\}$, $w[1] \leftarrow 1$, $\Sigma[1] \leftarrow 1$
for $i = 2$ to $n$ do
    if $A[i] = 0$ then
        $cand[i] \leftarrow \{0\}$
        $\Sigma[i] \leftarrow \Sigma[A[i-1]+1]+1$
        $Min\Sigma \leftarrow \max(Min\Sigma, \Sigma[i])$
        $w[i] \leftarrow \Sigma[i]$
    else
        $cand[i] \leftarrow cand[A[i-1]+1]$
        remove $A[A[i-1]+1]$ from $cand[i]$
        add $A[i-1]+1$ to $cand[i]$
        if $A[i] \notin cand[i]$ then
            error $A$ is not valid at $i$
        end if
        $w[i] \leftarrow w[A[i]]$
        $\Sigma[i] \leftarrow \Sigma[A[i-1]+1]$
    end if
end for
```

$$\begin{aligned} &w[i+1] = w\big[\pi[i]+1\big] \\ &\quad\iff \pi[i+1] = \pi[i]+1 \iff \pi'[i] < \pi[i] \iff \pi'[i] = \pi'\big[\pi[i]\big]. \end{aligned} \tag{1}$$

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

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

For two arrays of numbers A and B, we write $A[i_a \mathinner{..} i_a+k] \geq B[i_b \mathinner{..} i_b+k]$ when $A[i_a+j] \geq B[i_b+j]$ for $j = 0, \dots, k$.

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, see Algorithm 4, 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 useful for us, though it

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

```
1: $A[1] \leftarrow 0, i \leftarrow 0, n \leftarrow 1, A'[0] \leftarrow -1$
2: while TRUE do
3:     $n \leftarrow n + 1$
4:     if $A'[n] \neq A'[A[n]]$ then                ▷ This includes $A'[n] = A[n]$
5:         ADJUST-LAST-SLOPE
6:     end if
7: end while
```

should be noted that it depends only on $A[1\,..\,n]$ and that 0 and $A[n] + 1$ are always valid π-candidates.

The key idea needed to understand the algorithm is that $w[i]$ depends only on the letters of w at positions $A^k[i-1]+1$ for $k = 1, 2, \ldots$. Thus the algorithm stores $\Sigma[i]$, the alphabet size required for such sequence of indices starting at i, for all i. The minimum size of the alphabet required for the whole array A is the maximum over all those values.

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

(Val1) the valid candidates for $\pi[i]$ depend only on $\pi[1\,..\,i-1]$,
(Val2) $\pi[i-1] + 1$ is always a valid candidate for $\pi[i]$,
(Val3) if the alphabet needed for $A[1\,..\,n]$ is strictly larger than the one needed for $A[1\,..\,n-1]$ 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, compute 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 values of i. Removing this condition invalidates the bijection and, as a consequence, for intermediate values of i there can be many border arrays consistent with $A'[1\,..\,i]$, each of them corresponding to a different value of $A[i+1]$. We show that all these border arrays coincide on a certain prefix. VALIDATE-π', demonstrated in Algorithm 5, identifies this prefix and runs VALIDATE-π on it. Concerning the remaining suffix, VALIDATE-π' identifies the border array which is *maximal* on it, in a sense explained below.

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

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

A function $A[1\,..\,n+1]$ consistent with $A'[1\,..\,n]$ is *maximal* if

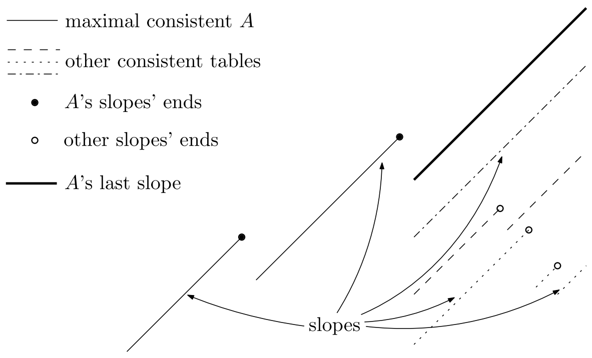

Fig. 1 Graphical illustration of slopes and maximal consistent function

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

Note that it is crucial that A is defined also on $n+1$.

Our algorithm VALIDATE-π' (and its improved variant LINEAR-VALIDATE-π') maintains 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 $i, i+1, \dots, i+j$ is a *slope* if $A[i+k] = A[i] + k$ for $k = 1, \dots, j$. From here on whenever we refer to slope, we implicitly mean a maximal one, i.e., extending as far as possible in both directions. Note that $n+1$ is part of the last slope, which may consist only of $n+1$. It is even better to imagine a slope a collection of points $(i, A[i])$ which together span one interval on the plain, see Fig. 1. Observe also that $A[i+j+1] \neq A[i+j]+1$ implies $A[i+j] = A'[i+j]$, by (1), i.e., the last index of a (maximal) slope is the unique one on which $A[i+j] = A'[i+j]$. Let the *pin* be the first position on the last slope of A (in some extreme cases it might be that $n+1$ is the pin). 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 $[i\,..\,n+1]$ should satisfy the following conditions:

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

$$A'[j] = A'\big[A[j]\big], \quad \text{for each } j \in [i\,..\,n]. \tag{3b}$$

The last slope is defined correctly if and only if (3a) holds (otherwise the slope should end earlier), while the values of A and A' on the last slope are consistent if and only if (3b) holds. These conditions are checked by appropriate queries: (3a) 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 (3b) by the *consistency check* (denoted CONSISTENCY-CHECK), which checks whether $A'[i\,..\,n] = A'[A[i]\,..\,A[i] + (n-i)]$.

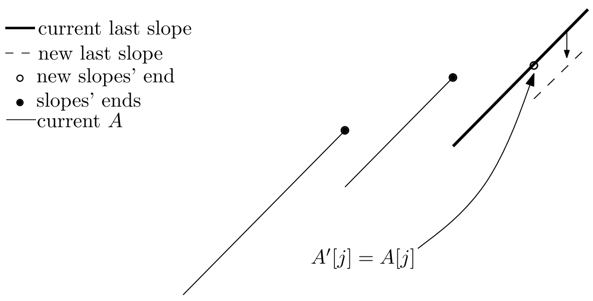

Fig. 2 Splitting the last slope: we move the whole last slope down until a point $(j, A'[j])$ is found on it. This point divides the slope into two. The left new slope stays in place and the right new one is moved further down

Algorithm 6 ADJUST-LAST-SLOPE

```
1: while $j \leftarrow$ PIN-VALUE-CHECK is defined do
2:     if $A'[j] > A[i] + (j - i)$ then
3:         error $A'$ is not valid at $n$
4:     end if
5:     for $m \leftarrow i$ to $j - 1$ do
6:         store $A[m] \leftarrow A[m-1] + 1$        ▷ Needed to calculate the set of candidates
7:         VALIDATE-$\pi(A)[m]$
8:         if $A'[m] \neq A'[A[m]]$ then
9:             error $A'$ is not valid at $n$
10:        end if
11:    end for
12:    store $A[j] \leftarrow A[j-1] + 1$        ▷ Maximal possible candidate
13:    $i \leftarrow j + 1$
14:    $A[i] \leftarrow$ next candidate        ▷ Known due to VALIDATE-$pi(A)$
15: end while
16: if not CONSISTENCY-CHECK then
17:     if $A[i] = 0$ then
18:         $A'$ is not valid at $n$
19:     end if
20:     $A[i] \leftarrow$ next candidate
21:     goto 1:
22: end if
```

If one of the conditions (3a), (3b) does not hold, VALIDATE-π' adjusts the last slope of A, until both conditions hold or the input is reported as invalid. These actions are given in detail in Algorithm 6.

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.

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: for every A_1 consistent with A' it holds that $A_1[j] \geq A'[j] \geq A[j]$, but as A is maximal consistent with A', it also holds that $A_1[j] \leq A[j] = A'[j]$, and

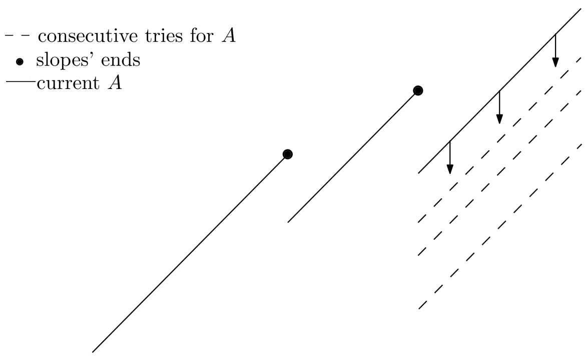

Fig. 3 Decreasing the $A[i]$. The values on the whole last slope are decreased by the same value

hence $A_1[j] = A[j]$. We also check whether

$$A'[i\,..\,j-1] = A'\bigl[A[i]\,..\,A[i]+(j-i-1)\bigr]$$

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

If Consistency-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-time 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 [19, 24] are linear only for alphabets of constant size, while the only linear-time algorithm for larger 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 Sect. 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 do not address the running time and the way the data structures are organised. We start with showing that all the consistent tables coincide on indices smaller than pin.

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

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 a minimal-size alphabet Σ such that $\pi_w[1\,..\,i-1] = A[1\,..\,i-1]$.

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

Lemma 2 *If* $A'[n] = A'[A[n]]$, *then no changes are done by* VALIDATE-π' *and the* CF1–CF3 *are preserved.*

Proof Whenever a new symbol is read, VALIDATE-π' checks (3b) 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 (3b).
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 hold. To be more specific, CF1 alone means that A is always a valid border array, 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). Finally, we show that CF3 holds when ADJUST-LAST-SLOPE ends adjusting the last slope, i.e., that then A is in fact consistent with A'.

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

Lemma 3 *After each iteration of the loop in line 1 of* ADJUST-LAST-SLOPE *the* CF1 *and* CF3 *are preserved. Furthermore, if* ADJUST-LAST-SLOPE *rejects A' in line 3 or 9, then A' is invalid.*

Proof We show both claims by induction. In the following, let $A_1[1\,..\,n+1]$ be any table consistent with $A'[1\,..\,n]$.

For the induction base note that $A[1\,..\,n]$ and $A'[1\,..\,n-1]$ satisfy CF1–CF3. To see that CF1 is satisfied by $A[1\,..\,n+1]$ note that the assigned value $A[n+1] = A[n]+1$ is always a valid π-value, so CF1 holds for $A[1\,..\,n+1]$. Similarly, for CF3 note that $A[1\,..\,n] \geq A_1[1\,..\,n]$ and $A[n+1] = A[n]+1 \geq A_1[n]+1 \geq A_1[n+1]$, which shows that CF3 holds for $A[1\,..\,n+1]$. Additionally, the second claim of the Lemma holds vacuously for $A[1\,..\,n+1]$, as so far it was not rejected.

Suppose that PIN-VALUE-CHECK returns no index j. Then by the induction assumption CF1 and CF3 hold, which ends the proof in this case.

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'. This ends the proof in this case.

It is left to consider the case in which PIN-VALUE-CHECK returns j such that $A[j] = A'[j]$. Then CF1 is satisfied: $A[j]$ is explicitly set to a valid π candidate while for $p > j$ the $A[p]$ is set to $A[p] = A[p-1]+1$, which is always a valid π candidate, by Val2. Furthermore, 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, $A[i\,..\,j] = A_1[i\,..\,j]$. 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 $A'[p] = A'[A[p]]$, and this condition is verified by ADJUST-LAST-SLOPE in line 8. If this equation is not satisfied by some p then clearly $A'[i\,..\,j-1]$ is not consistent with $A[i\,..\,j]$. Since $A_1[i\,..\,j] = A[i\,..\,j]$ this shows that no such A_1 exists and consequently A' is invalid. This shows the second subclaim.

Suppose that A' was not rejected. It is left to show that CF3 is satisfied when PIN-VALUE-CHECK returns j such that $A[j] = A'[j]$. 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 $A[p]$ for $p \in [i+1\,..\,n]$ satisfy $A[p] = A[i] + (p-i)$. Since A_1 is a valid π table $A_1[p] \leq A_1[i] + (p-i)$ for $p = i+1, \dots, n$ and thus:

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

and as A_1 was chosen arbitrarily, CF3 holds. □

Lemma 4 *Suppose that* PIN-VALUE-CHECK *returns no* j *and that* A *satisfies* CF1 *and* CF3. *If* CONSISTENCY-CHECK *returns* FALSE *and* $A[i] = 0$ *then* A' *is invalid. Otherwise after adjusting in line* 20 *of* ADJUST-LAST-SLOPE, CF1 *and* CF3 *hold.*

Proof Let as in the previous lemma A_1 denote any valid border array consistent with A'. Since A satisfies CF3, we know that $A[i] \ge A_1[i]$. When $A[i]$ is updated to the next largest valid π candidate, its new value is at least $A_1[i]$ (as $A_1[i]$ is itself a valid π value) and for each $p > i$ we have

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

which shows that CF3 is preserved after the adjusting.

We now prove that in fact $A[i] > A_1[i]$. Suppose for the sake of contradiction that $A[i] = A_1[i]$. It is not possible that $A[i\,..\,n+1] = A_1[i\,..\,n+1]$: since PIN-VALUE-CHECK returned no j, for each $p \ge i$ we have $A_1[p] = A[p] > A'[p]$. In such case by (2) it holds that $A'[p] = A'[A_1[p]]$ but from the answer of the PIN-VALUE-CHECK we know that this is not the case.

Consider the smallest position, say p, such that $A[p+1] > A_1[p+1]$; such a position exists as $A[i\,..\,n+1] \ge A_1[i\,..\,n+1]$ and $A[i\,..\,n+1] \ne A_1[i\,..\,n+1]$. Now consider $A_1[p]$: since $A_1[p+1] < A_1[p]+1$ then by (2) this means that $A_1[p] = A'[p]$. This is a contradiction, as PIN-VALUE-CHECK should have returned this p.

Therefore, when CONSISTENCY-CHECK returns NO then $A_1[i] < A[i]$ for an arbitrary A_1 that is consistent with A'. In particular, if $A[i] = 0$, there is no such A_1, and hence A' is invalid.

It is left to show that CF1 holds, i.e., that $A[i\,..\,n+1]$ 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 Recall the recursive formula (2) for π'. Its first case corresponds to j being the last element on the slope and the second 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: Lemma 2–Lemma 5 together show the correctness of VALIDATE-π'.

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

Proof We proceed by induction on n. If $n = 0$, then clearly $A[1] = 0$, CF1–CF3 hold trivially, 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 $w[1\,..\,n]$. □

In the following section we explain how to perform the pin value checks and consistency checks efficiently and bound the whole running time of the algorithm.

5 Performing Pin Value Checks

Consider the PIN-VALUE-CHECK and two indices j, j' such that

$$j < j' \quad \text{and} \quad A'[j'] - j' > A'[j] - j. \tag{4}$$

We call the relation defined in (4) a *domination*: we say that j' *dominates* j and write it as $j \prec j'$. We will 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 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.

Domination Properties 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 $A[j] \le A'[j]$ implies $A[j'] < A'[j']$:

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

Therefore if j is an answer to pin value check, so is j'. As a consequence, we do not need to keep track of j as a potential answer to the PIN-VALUE-CHECK.

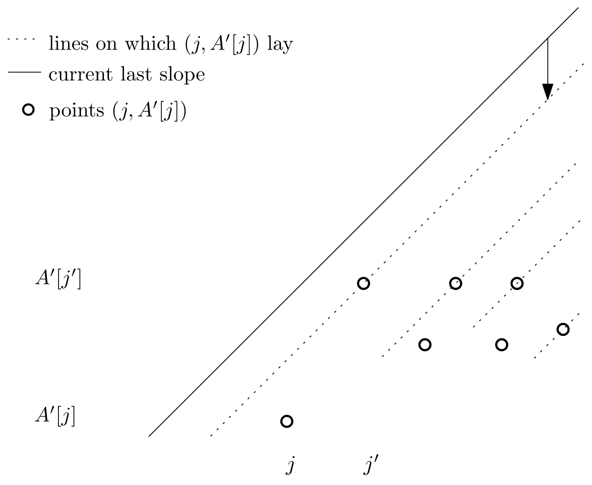

Fig. 4 Answering pin value check. The last slope is lowered until some point $(j', A'[j'])$ is on it. On the picture j' dominates j and j cannot be returned by pin value check: the 'slope' going through j is below the one going through j'

Data Stored VALIDATE-π' stores a list of positions $j_1 < j_2 < \cdots < j_k$ such that (for the sake of simplicity, let $j_0 = i$, where i is the current pin):

$$j_{\ell'} \not\succ j_\ell \quad \text{for all } 0 < \ell' < \ell, \tag{6a}$$

$$j_\ell \succ j \quad \text{for all } 0 < \ell \le k \text{ and } j \in [j_{\ell-1} + 1 \,..\, j_\ell - 1]. \tag{6b}$$

Answering PIN-VALUE-CHECK When PIN-VALUE-CHECK is asked, we check whether $A[j_1] \le A'[j_1]$ and return the answer. 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, as if $A'[j] \ge A[j]$ for some j, then $A'[j_1] \ge A[j_1]$, and moreover if $A'[j] > A[j]$, then also $A'[j_1] > A[j_1]$.

Suppose that $A'[j] \ge A[j]$ for some $j \in [j_{\ell-1} + 1 \,..\, j_\ell - 1]$. Since j_ℓ dominates j it holds that $A'[j_\ell] > A[j_\ell]$, by (5). Suppose now that $A'[j_\ell] \ge A[j_\ell]$ for some $j_\ell > j_1$. Since $j_1 < j_\ell$ and j_ℓ does not dominate j_1:

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

As j_1 and j_ℓ are on the last slope,

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

and hence

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

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 $j_1, \dots, j_k$ 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:

$$\begin{aligned} A[n] - n > A[j_\ell] - j_\ell \quad &\text{as } n \prec j_\ell, \\ A[j_\ell] - j_\ell \geq A[j_{\ell+1}] - j_{\ell+1} \quad &\text{as } j_\ell \not\succ j_{\ell+1} \text{ by (6a)}. \end{aligned}$$

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 $j_\ell, \dots, j_k$ were removed. It is left to show that (6a), (6b) are preserved after the removal. Consider first (6a). Take 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 (6b), $j_{\ell'} \succ j$. Since by assumption $n \succ j_{\ell'}$, by transitivity of $\succ$, also $n \succ j$. As for (6b), it holds since $j_{\ell-1} \not\succ n$ by the construction.

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.

Lemma 6 *All* Pin-Value-Check *calls can be made in amortised constant time.*

6 Performing Consistency Checks: Slow but Easy

In order to perform consistency check 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 $\mathcal{O}(\log n)$ per one consistency check. 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 $\mathit{name}[i][j]$ be an integer from $\{1, \dots, n\}$ such that $\mathit{name}[i][j] = \mathit{name}[i'][j]$ 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 of 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 $\mathit{name}[n - 2^j + 2][j]$ for all $j = 1, \dots, \log n$. We set $\mathit{name}[n+1][0]$ to $A'[n+1]$. For names with $j > 0$ 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 each $j > 0$ we keep a dictionary $M(j)$ mapping pair $(\mathit{name}[i][j-1], \mathit{name}[i + 2^{j-1}][j-1])$ to $\mathit{name}[i][j]$. To check if a given fragment $A'[n - 2^j + 2 \mathinner{..} n+1]$ occurs previously in the text, we look up the pair $(\mathit{name}[n - 2^j + 2][j-1], \mathit{name}[n - 2^{j-1} + 2][j-1])$ 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, which 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: we insert 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 [22]). Then the expected running time of the whole algorithm becomes $\mathcal{O}(n \log n)$, 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

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 also true of VALIDATE-π'. Roughly speaking, VALIDATE-π' runs VALIDATE-π and simply returns its answers. To this end we show that the minimum alphabet size required by the fixed prefix of A matches the minimum alphabet size required by A'.

Lemma 7 *Let $A'[1\,..\,n]$ be a valid π' function, $A[1\,..\,n+1]$ the maximal function consistent with $A'[1\,..\,n]$, and i 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 $B[1\,..\,i-1] = A[1\,..\,i-1]$ by Lemma 1. Thus A requires an alphabet larger than that required by $B[1\,..\,i-1]$, which is clearly no larger than the one required by the whole $B[1\,..\,n]$.

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

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

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

Note that VALIDATE-π' runs VALIDATE-π either on $A[1\,..\,i-1]$, or on $A[1\,..\,i-1]$ when $A[i] = 0$. In either case, all these values are fixed, and thus no position of A is inspected twice by VALIDATE-π.

We further note that Lemma 7 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 $\mathcal{O}(\log n)$ [20, Th. 3.3a]. This observation implies 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

This section describes our linear time online algorithm LINEAR-VALIDATE-π' by specifying necessary changes to VALIDATE-π'. It suffices to show how to perform consistency checks more efficiently, as each other operations works in amortised constant time. A natural approach is as follows: construct a suffix tree [10, 19, 24] 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 [19, 24] have a slightly bigger running time of $\mathcal{O}(n \log |\Sigma|)$, where Σ is the alphabet. As A' is a text over an alphabet $\{-1, 0, \dots, n-1\}$, i.e., of size $n+1$, these constructions would only guarantee an $\mathcal{O}(n \log n)$ 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 [24] so that it runs in time $\mathcal{O}(n)$ for alphabets of polylogarithmic size, which may be of independent interest. 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 [24], it guarantees the following construction of suffix trees.

Lemma 8 *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 [24] 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 a 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.

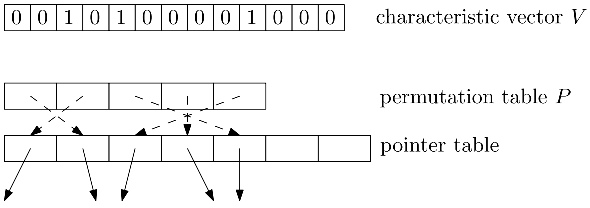

Fig. 5 Basic structure for succinct suffix tree

Simplifying Assumptions We assume that the value of $\lfloor \log n \rfloor$ is known. Since n is not known in 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.

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 $t \le \sqrt{\log n}$. Then this construction is folded twice to obtain the general case of $t = \Theta(\log n)$. One step of such a construction is depicted on Fig. 5.

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 $t \le \sqrt{\log n}$, all successive values of such permutation can be stored in 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., $j = \#\{k' \le k : \ V[k'] = 1\}$. 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 at the next available position 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 $m = \#\{k' : \ V[k'] = 1\}$, 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 $m+1$ 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 $P[j]=m+1$.

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 $k_{i-1} \le k < k_i$, 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.

If after an insertion the bottom structure has $2t+1$ elements, we choose its middle element, insert it into the top structure, and split the keys into two parts consisting of t elements, creating two new bottom structures out of them. This requires $\mathcal{O}(t)$ time but the amortised insertion time is only $\mathcal{O}(1)$: 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 8 does not apply to A' directly, as it may hold too many different values. To overcome this, we compress A' into $\mathit{Compress}(A')$, 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 $\mathit{Compress}(A')$. To compress A', we scan it from left to right. If $A'[i]=A'[i-j]$ for some $1 \le j \le \log^2 n$ we output $\#_0 j$. If $A'[i] \le \log^2 n$ 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 $\mathit{Compress}(A')$ in $\mathit{Start}[i]$.

Note that we need to know, whether a value $A'[n]$ appeared within the last $\log^2 n$ positions. To do this, we keep a table Prev, such that $\mathit{Prev}[i]$ gives the position of the last i in A' (or -1, if no i appeared so far). It is easily updated in constant time: when we read $A'[n]$ we set $\mathit{Prev}[A'[n]]$ to n and $\mathit{Prev}[n]$ to -1.

In this encoding only the last case of $A'[i] > \log^2 n$ and $A'[i]$ not occurring in $A'[i - \log^2 n \mathinner{.\,.} i-1]$ may result in more than one symbol of an alphabet of size $O(\log^2 n)$. We show that the number of different large values of π' is small, which allows bounding the total size of these encodings and hence the whole *Compress*(A') table by $\mathcal{O}(n)$.

Lemma 9 *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.*

Proof First note that each i such that $\pi'[i] > 0$ corresponds to a non-extensible occurrence of the border $w[1 \mathinner{.\,.} \pi'[i]]$, i.e., $\pi'[i]$ is the maximum j such that $w[1 \mathinner{.\,.} j]$ is a suffix of $w[1 \mathinner{.\,.} i]$ 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 $[2^k, 2^{k+1}) = [4 \cdot 2^{k'}, 8 \cdot 2^{k'})$ 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 (since they are ends of non-extendable borders). 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

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

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

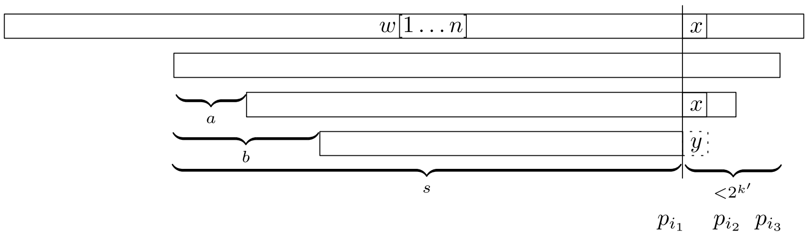

Fig. 6 Proof of Lemma 9, decreasing sequence

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

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

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

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} (note that $b < \frac{\ell}{2}$ and $\pi'[p_{i_1}] \ge \ell$):

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

see Fig. 7. Note that

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

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

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

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 $\pi'[p_{i_1}], \pi'[p_{i_2}] \ge \ell$. We identify the periods of the corresponding subwords $w[1\,..\,\pi'[p_{i_1}]]$ and $w[1\,..\,\pi'[p_{i_2}]]$, respectively:

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

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

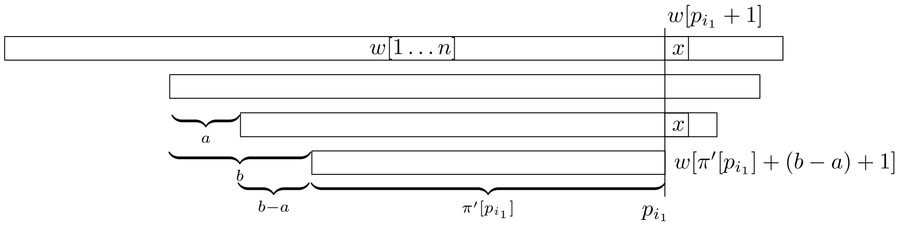

Fig. 7 An illustration of equality $w[p_{i_1}+1]=w[\pi'[p_{i_1}]+(b-a)+1]$

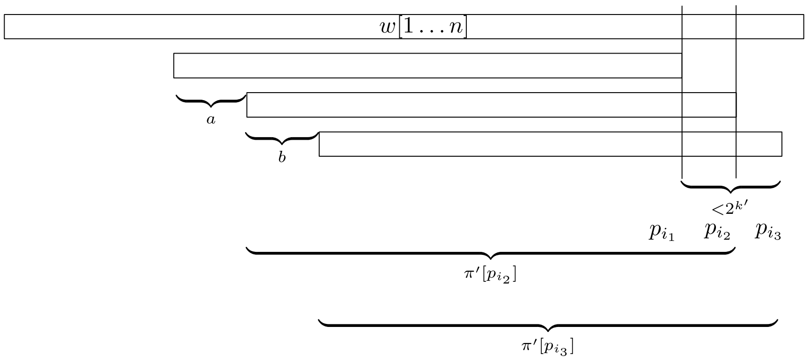

Fig. 8 Proof of Lemma 9, increasing sequence

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

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

$$a+b \le \frac{\pi'[p_{i_1}]}{2}, \frac{\pi'[p_{i_2}]}{2}. \tag{7}$$

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 $\pi'[p_{i_1}]$, $x \ne y$. We obtain a contradiction by showing that $x = y$.

Since the non-extensible border ending at p_{i_3} spans over position $p_{i_1}+1$ and $a+b < \pi'[p_{i_1}]$ (see (7)) it holds that

$$x = w\bigl[\bigl(\pi'[p_{i_1}]+1\bigr)-(a+b)\bigr]. \tag{8}$$

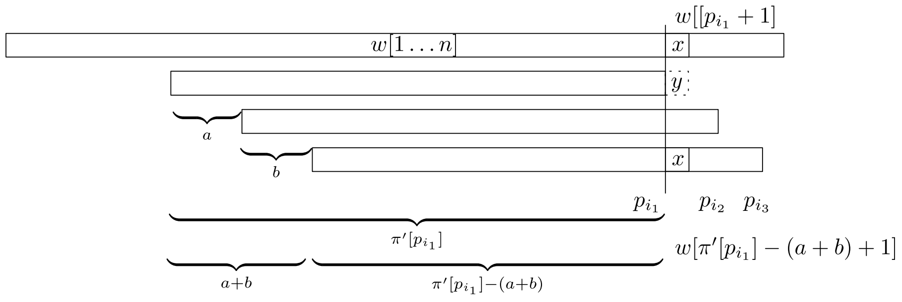

Fig. 9 Illustration for case 2a, when $w[(\pi'[p_{i_1}]+1)-(a+b)] = w[p_{i_1}+1]$

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 $\pi'[p_{i_1}]+1 \le \pi'[p_{i_2}]$,

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

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

$$y = w\big[\pi'[p_{i_1}]+1-b\big] = w\big[\pi'[p_{i_1}]+1-b-a\big] \tag{9}$$

and therefore by (8) and (9) $x = y$. 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

$$x' = w\big[\pi'[p_{i_2}]-b+1\big], \tag{10}$$

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 $\pi'[p_{i_2}]+1 \le \pi'[p_{i_1}]$,

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

By comparing the non-extensible prefixes ending at p_{i_2} and p_{i_3} we deduce that b is a period of $w[1\,..\,\pi'[p_{i_2}]]$. Since $a+b \le \frac{\pi'[p_{i_2}]}{2}$ by (7), it holds that

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

As a is a period of $w[1\,..\,\pi'[p_{i_1}]]$ and $\pi'[p_{i_1}] > \pi'[p_{i_2}]$ it is also a period of $w[1\,..\,\pi'[p_{i_2}]+1]$, hence

$$y' = w\big[\pi'[p_{i_2}]+1-a-b\big] = w\big[\pi'[p_{i_2}]+1-b\big]. \tag{11}$$

So by (10) and (11) $x' = y'$, contradiction. □

Lemma 9 can be used to bound the size of the compressed representation *Compress*(A') of A'.

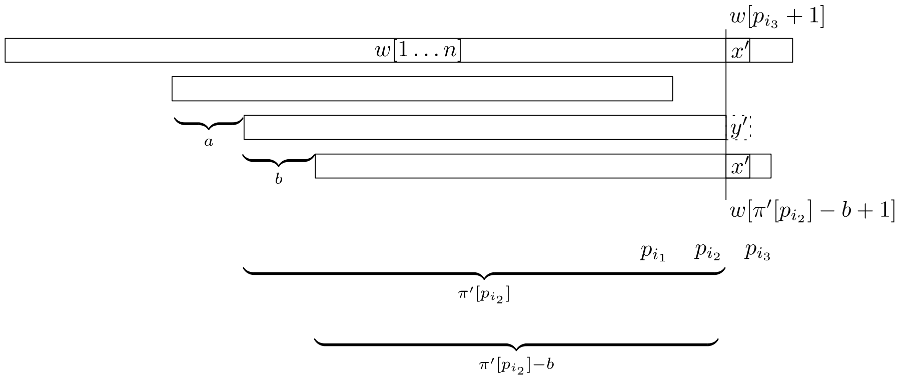

Fig. 10 Illustration for case 2b, when $w[(\pi'[p_{i_2}]+1)-b] = w[p_{i_3}+1]$

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 9, where $2^k \geq \log^2 n$, 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 *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

$$\sum_{k=2\log\log n}^{\log n}\left(48\frac{n}{\log^2 n}\log n\right) = (\log n - 2\log\log n + 1)\cdot 48\frac{n}{\log n} \leq 48n \in \mathcal{O}(n),$$

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

As the alphabet of *Compress*(A') is of polylogarithmic size, the suffix tree for *Compress*(A') can be constructed in linear time by Lemma 8.

8.3 Performing Consistency Checks on the *Compress*(A')

Subchecks Consider consistency check: is $A'[j\,..\,j+k-1] = A'[i\,..\,i+k-1]$, where $j = A[i]$? We first establish equivalence of this equality with equality of proper fragments of *Compress*(A'). Note, that $A'[\ell] = A'[\ell']$ does not imply the equality of two corresponding fragments of *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 follows:

Lemma 10 *Let $j = A[i]$. Then*

$$A'[j \mathinner{..} j+k-1] = A'[i \mathinner{..} i+k-1] \tag{12}$$

if and only if

$$\begin{aligned} &\mathit{Compress}\bigl(A'\bigr)\bigl[\mathit{Start}\bigl[j+\log^2 n\bigr] \mathinner{..} \mathit{Start}[j+k]-1\bigr] \\ &\quad = \mathit{Compress}\bigl(A'\bigr)\bigl[\mathit{Start}\bigl[i+\log^2 n\bigr] \mathinner{..} \mathit{Start}[i+k]-1\bigr] \end{aligned} \tag{13}$$

$$\text{and} \quad A'\bigl[j \mathinner{..} j+\min\bigl(k,\log^2 n\bigr)-1\bigr] = A'\bigl[i \mathinner{..} i+\min\bigl(k,\log^2 n\bigr)-1\bigr]. \tag{14}$$

Proof If $k \le \log^2 n$, the claim holds trivially, as (12) and (14) are exactly the same and (13) holds vacuously.

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

$\Leftarrow$ Suppose first that $A'[j \mathinner{..} j+k-1] = A'[i \mathinner{..} i+k-1]$. Then of course $A'[j \mathinner{..} j+\log^2 n-1] = A'[i \mathinner{..} i+\log^2 n-1]$, as $k > \log^2 n$ by case assumption. Thus (14) holds.

Note that $\mathit{Compress}(A')[\mathit{Start}[j+\log^2 n] \mathinner{..} \mathit{Start}[j+k]-1]$ is created using only $A'[j \mathinner{..} j+k-1]$: 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, $\mathit{Compress}(A')[\mathit{Start}[i+\log^2 n] \mathinner{..} \mathit{Start}[i+k]-1]$ is created using $A'[i \mathinner{..} i+k-1]$ exclusively. Since $A'[j \mathinner{..} j+k-1] = A'[i \mathinner{..} i+k-1]$, both fragments of $\mathit{Compress}(A')$ are created using the same input, and so they are equal. Thus (13) holds, which ends the proof in this direction.

$\Rightarrow$ Assume that (13) and (14) hold. We show by a simple induction on ℓ, that $A'[i+\ell] = A'[j+\ell]$. For $\ell < \log^2 n$ the claim is trivial, as it is explicitly stated in (14). 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 equal by the assumption.

– If they are both equal to $\#_0 m$ (i.e., both are equal to some value of A' that is $m \le \log^2 n$ positions earlier) then $A'[i+\ell] = A'[i+\ell-m]$ and $A'[j+\ell] = A'[j+\ell-m]$; by the inductive assumption $A'[i+\ell-m] = A'[j+\ell-m]$ (as $m \le \log^2 n$), which ends the case.
– If they are both equal to $\#_1 m$ (i.e., both are equal to $m \le \log^2 n$) then $A'[i+\ell] = A'[j+\ell] = m$.
– If they are equal to $\#_2 m_1 \ldots m_z \#_3$ (i.e., both are larger than $\log^2 n$ and are both encoded in binary as $m_1 \ldots m_z$) then $m_1 \ldots m_z$ encode some m in binary and $A'[i+\ell] = A'[j+\ell] = m$, which ends the last case. □

Similarly as in the Sect. 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 (13) the *compressed consistency checks*, checks of the form (14)—*short consistency checks* and the *near short consistency checks* when moreover $|i-j| < \log^2 n$.

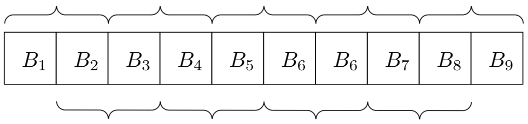

Fig. 11 Scheme of ranges for suffix trees

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

8.4 Performing Short Consistency Checks

Performing Near Short Consistency Checks To answer near short consistency checks efficiently, we split A' into blocks of $\log^2 n$ consecutive letters: $A' = B_1B_2\dots B_\ell$, see Fig. 11. Then we build suffix trees for each pair of consecutive blocks, i.e., $B_1B_2, B_2B_3, \dots, B_{\ell-1}B_\ell$. Each block contains at most $\log^2 n$ values smaller than $\log^2 n$, and at most $48\log n$ larger values by Lemma 9, so all suffix trees can be built in linear time by Lemma 8. 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. Such a query also gives the actual length of the longest common prefix of the two compared strings; this is used in performing short consistency checks.

Performing Short Consistency Checks Consider again a short consistency check, which is of the form 'does $A'[i\mathinner{..}i+k-1] = A'[j\mathinner{..}j+k-1]$', where $j = A[i]$ and $k \le \log^2 n$. To improve the running time, the results of previous short consistency checks are reused: we store j_{best} (which is one of indices for which previously we run short consistency check) such that

- $j \le j_{best} \le j + \log^2 n$
- the length (say L) of the common prefix of $A'[i\mathinner{..}i+k-1]$ and $A'[j_{best}\mathinner{..}j_{best}+k-1]$ is known.

To answer short consistency check we first compute the common prefix of $A'[j\mathinner{..}j+k-1]$ and $A'[j_{best}\mathinner{..}j_{best}+k-1]$ (which can be done using near short consistency check) and compare it with L. If it is smaller than $\min(L,k)$, then clearly the common prefix of $A'[j\mathinner{..}j+k-1]$ and $A'[i\mathinner{..}i+k-1]$ is smaller than k; if it equals L then we naively compute the common prefix of $A'[j+L\mathinner{..}j+k-1]$ and $A'[i+L\mathinner{..}i+k-1]$ by letter-to-letter comparisons. Also, in such a case we switch j_{best} to j, as it has a longer common prefix with $A[i\mathinner{..}i+k-1]$.

Simplifying Assumption To simplify the presentation and analysis, we assume that the adjusting of the last slope is done in a slightly different way than written in the code of ADJUST-LAST-SLOPE (see Algorithm 6): if the pin is assigned value $i' > i$ (in line 13 of Algorithm 6), firstly $A[i']$ is set to $A[i] + (i' - i)$, i.e., 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, even if it is an equality, we still treat it as a fail) 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.

Invariants During short consistency check we make sure that the following invariants for j_{best} and L are preserved:

$$L \leq k \tag{15a}$$

$$A'[j_{best} \mathinner{..} j_{best} + L - 1] = A'[i \mathinner{..} i + L - 1] \tag{15b}$$

$$j \leq j_{best} \leq j + \log^2 n \tag{15c}$$

$$\text{if } j \neq j_{best} \text{ then } A'[j \mathinner{..} j + L - 1] \neq A'[j_{best} \mathinner{..} j_{best} + L - 1] \tag{15d}$$

$$L = k \text{ or } A'[j_{best} + L] \neq A'[i + L]. \tag{15e}$$

We refer to them as (15a)–(15e).

The intuition behind the invariants is as follows: (15a) simply states that we are interested in common prefix of length at most k. The (15b) justifies the choice of j_{best}, i.e. we know the common prefix of A' starting at j_{best} and at i. The (15c) ensures that comparing A' starting at j and j_{best} can be done using near short consistency check. The (15d) says that if $j \neq j_{best}$ then there is a reason for that: $A'[i \mathinner{..} i + k - 1]$ and $A'[j \mathinner{..} j + k - 1]$ have a shorter common prefix then $A'[i \mathinner{..} i + k - 1]$ and $A'[j_{best} \mathinner{..} j_{best} + k - 1]$. Finally, (15e) shows maximality of L: either it is k (so it cannot be larger) or there is a mismatch at the 'next position'.

Potential The analysis of the running time is amortised. We define a *potential* of the configuration of LINEAR-VALIDATE-π' as

$$p = k - L + (j_{best} - j). \tag{16}$$

Let Δx denote the change of the value of x in some fragment of an algorithm (which will be always clear from the context); let s be the cost of comparisons and near short consistency checks (i.e. their number). Then the *amortised cost* is $\Delta p + s$. There are some additional costs, like comparing indices, checking conditions etc. All such costs are assigned either to letter-to-letter comparisons or to near short consistency checks.

Note that when the change of the potential is negative then it actually helps in paying for near short consistency checks and letter-by-letter comparisons. Since $0 \leq L \leq k$ and $j \leq j_{best} \leq j + \log^2 n$, at any point the potential is non-negative and at most $\log^2 n$, so the total cost at any point is the sum of amortised costs in each step and the potential, which is sublinear.

We pay for the amortised cost using *credit* that we get for the changes of n and j: For every increase Δn, we get $8\Delta n$ units of credit; for every change of j we get $8|\Delta j|$ units of credit. Clearly the sum of all Δn is n, so in this way we are scored at most $2n$ credit. We show that the sum of all $|\Delta j|$ is also $\mathcal{O}(n)$.

Lemma 11 *The sum of all $|\Delta j|$ over the whole run* VALIDATE-π' *is* $2n$.

Proof For the purpose of the proof, whenever we change the value of i or j let i', j' refer to the new values and i, j to the old ones.

It is enough to show that the sum of all increments of j is at most n then clearly the sum of all decrements of j are at most n as well.

Algorithm 7 LETTER-BY-LETTER

```
1: While $A'[j_{best} + L] = A'[i + L]$ and $L < k$ do        ▷ we can increment $L$
2:     $L \leftarrow L + 1$
3: End while                              ▷ $L$ is the length of the common prefix
```

The j increases only when the pin i is updated in line 13 of ADJUST-LAST-SLOPE, otherwise it can only decrease. Moreover, when j is incremented, it increases by at most Δi:

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

Note that in the third equality we essentially used the simplifying assumption: as i' and i are on the same (last) slope, we have $A[i'] = A[i] + (i' - i)$.

Since $i \le n$ and i only increases, its sum of increments is at most n. So the total sum of increments of j is at most n, as claimed. □

Letter-by-Letter Comparisons The letter by letter comparisons, see Algorithm 7, are used to ensure that (15e) holds: when we already know that L letters starting at $A'[i]$ and $A'[j_{best}]$ are the same but we are not sure whether this is the maximal possible value of L, we verify this naively. The amortised cost is only 1, as each successful comparison decreases the potential by 1.

Lemma 12 *If* (15a)–(15c) *are satisfied before* LETTER-BY-LETTER, *then* (15a)–(15c) *and* (15e) *are satisfied afterwards. The amortised cost of* LETTER-BY-LETTER *is* 1.

Proof For the purpose of the proof, let L_0 be the initial value of L and L_1 the final value of L; by 'L' we denote the value inside LETTER-BY-LETTER.

Note that i, j and k are not altered. For (15a), by assumption $L_0 \le k$ before COMMON-SHORT-CONSISTENCY-CHECK, we increment L by 1 and stop as soon as it reaches k, so $L_1 \le k$. For (15b) note that $A'[j_{best} \,.\,.\, j_{best} + L_0 - 1] = A'[i \,.\,.\, i + L_0 - 1]$ holds by the assumption and we verified $A'[j_{best} + L_0 \,.\,.\, j_{best} + L_1 - 1] = A'[i + L_0 \,.\,.\, i + L_1 - 1]$ letter by letter. Invariant (15c) holds as neither j nor j_{best} was changed. As for (15e), it is the termination condition of the while loop, so it holds upon its termination.

Concerning the amortised cost: i, j, j_{best} do not change, so $\Delta p = -\Delta L$, i.e. it is negative. On the other hand we make ΔL successful letter-to-letter comparisons and perhaps one unsuccessful one (we ignore the cost of checking whether $L = k$, as they are at most as high as the cost of letter-to-letter comparisons). So the cost of comparisons is at most $\Delta L + 1$. Hence the amortised cost is at most $-\Delta L + \Delta L + 1 = 1$, as claimed. □

Answering Short Consistency Checks Using j_{best} When we get new values of i, j and k we need to update j_{best} and L. It turns out that as soon as we update j_{best} and L so that they satisfy (15a)–(15c), answering short consistency check is easy: we first make letter-by-letter comparisons using LETTER-BY-LETTER to ensure that also (15e) holds, i.e. that L is maximal. Then we check the length of the common

Algorithm 8 COMMON-SHORT-CONSISTENCY-CHECK

```
1: LETTER-BY-LETTER
2: $\ell \leftarrow$ NEAR-SHORT-CONSISTENCY-CHECK($A'[j_{best} \,..\, j_{best}+k-1], A'[j \,..\, j+k-1]$)
3: if $\ell < L$ then
4:    return NO
5: else
6:    $j_{best} \leftarrow j$
7:    LETTER-BY-LETTER
8:    if $L = k$ then
9:       return YES
10:   else
11:      return NO
12:   end if
13: end if
```

prefix of $A'[j \,..\, j+k-1]$ and $A'[j_{best} \,..\, j_{best}+k-1]$ by a near short consistency check. If it is less than L, then the answer to short consistency check is no. If it is at least L, then we set j_{best} to j (as it is as good as j_{best}), run letter-by-letter comparison again to check whether L is k, and answer accordingly. It is easy to verify that the amortised cost of this procedure is constant and that all (15a)–(15e) hold afterward. Details are given in Algorithm 8 and lemmata below.

Lemma 13 *Assume that* (15a)–(15c) *are satisfied. Then* COMMON-SHORT-CONSISTENCY-CHECK *correctly answers the short consistency check, its amortised cost is* 6 *and all* (15a)–(15e) *hold after* COMMON-SHORT-CONSISTENCY-CHECK.

Proof Regarding the cost, the amortised cost of LETTER-BY-LETTERis 1 by Lemma 12, setting j_{best} to j can only lower potential, and NEAR-SHORT-CONSISTENCY-CHECK are answered in constant time using suffix trees.

We now show that after COMMON-SHORT-CONSISTENCY-CHECK all (15a)–(15e) hold. By assumption initially (15a)–(15c) hold. By Lemma 12 after the first LETTER-BY-LETTER they still hold and additionally (15e) holds. Suppose that $\ell < L$, in particular $j \neq j_{best}$. Then (15d) simply states that $\ell < L$, which is the case. So suppose that $\ell \geq L$. Resetting j_{best} to j may make (15e) invalid, but (15a)–(15c) are preserved: the (15a) holds as we do not change L, the (15b) holds as we know that $A'[j_{best} \,..\, j_{best}+k-1]$ has a common prefix of length L with both $A'[j \,..\, j+k-1]$ and $A'[i \,..\, i+k-1]$ and so also $A'[j \,..\, j+k-1]$ and $A'[i \,..\, i+k-1]$ have a common prefix of length L. The (15c) holds trivially. By Lemma 12 the (15a)–(15c) and (15e) hold after LETTER-BY-LETTER. Note that LETTER-BY-LETTER does not modify j and so (15d) trivially holds, as $j = j_{best}$.

Concerning the correctness: if $\ell < L$ then $j \neq j_{best}$ and from (15b) and (15d) we get that $A'[j \,..\, j+L-1]$ and $A'[i \,..\, i+L-1]$ are different. Since by (15a) we know that $L \leq k$, hence also $A'[j \,..\, j+k-1]$ and $A'[i \,..\, i+k-1]$ are different. This justifies the no answer. If $\ell \geq L$ then in the end $j = j_{best}$ and so by (15a)–(15b) and (15e) we know that $A'[j\,..\,j+k-1] = A'[i\,..\,i+k-1]$ if and only if $L = k$, which is exactly the answer returned by the algorithm. □

It remains to show how to update j_{best} and L.

Types of Short Consistency Checks The way we update j_{best} and L depends on why the short consistency check is made; we distinguish three situations in which ADJUST-LAST-SLOPE invokes short consistency check:

(**Type 1**) This is a first iteration of ADJUST-LAST-SLOPE and PIN-VALUE-CHECK did not return any index in this iteration.
(**Type 2**) This is not a first iteration of ADJUST-LAST-SLOPE and PIN-VALUE-CHECK did not return any index in this iteration.
(**Type 3**) The PIN-VALUE-CHECK did return an index in this iteration.

We begin with showing what are the changes of i, j, k and n in each of those types of short consistency check.

Lemma 14 *In Type* 1 *short consistency check it holds that* $\Delta i = \Delta j = 0$, $\Delta k \geq 0$ *and* $\Delta n \geq \max(1, \Delta k)$; *exactly* $8\Delta n$ *units of credit are issued.*

In Type 2 *short consistency check it holds that* $\Delta i = 0$, $\Delta j < 0$ *and* $\Delta k = \Delta n = 0$; *exactly* $8|\Delta j|$ *units of credit are issued.*

In Type 3 *short consistency check it holds that* $\Delta i > 0$, $\Delta j = \Delta i$ *and* $-\Delta i \leq \Delta k \leq 0$, $\Delta n \geq 0$; *exactly* $8\Delta j + 8\Delta n$ *units of credit are issued.*

Note in particular that when Δi, Δj are known, we can figure out which type of query this is: Type 3 short consistency check is unique with $\Delta i > 0$, Type 2 with $\Delta j < 0$ while Type 1 with $\Delta i = \Delta j = 0$.

Proof Recall that we issue $8\Delta n + 8|\Delta j|$ units of credit, which yields the claim on the number of credit issued in each of the cases.

Type 1 short consistency check: Since this is the first iteration of ADJUST-LAST-SLOPE it means that we read $A'[n]$ and it is not equal to $A'[A[n]]$. In particular, since the last invocation of ADJUST-LAST-SLOPE we read at least one additional value of A'. Hence $\Delta n \geq 1$. As PIN-VALUE-CHECK did not return any index, we do not modify i and j since the last invocation of the short consistency check, so $\Delta i = \Delta j = 0$. Concerning k, recall that the short consistency check is asked only on $A'[i\,..\min(n, i + \log^2 n - 1)]$, i.e. $k = \min(n - i + 1, \log^2 n)$. Hence, when k_0 and n_0 are the values of k and n when previous short consistency check was asked, we have $k_0 = \min(n_0 - i + 1, \log^2 n)$ (note that we can assume that $\log n$ and $\log n_0$ are the same, as we repeat the calculation as soon as $\lceil \log n \rceil$ increases). Then $k \geq k_0$ and $\Delta k \leq \Delta n$, but there is no guarantee that $\Delta k > 0$, i.e., $k_0 = k$ can happen when $n_0 - i + 1 > \log^2 n$.

Type 2 short consistency check: in this case short consistency check is asked in iteration of ADJUST-LAST-SLOPE that is not the first one, and the PIN-VALUE-CHECK did not return any index in this iteration. Which means that $A[i]$ is assigned the next candidate in line 14. Thus i, k are unchanged as compared to the previous

Algorithm 9 TYPE-1-UPDATE-j_{best}

```
1: continue
```

short consistency check, while j is decreased, hence $\Delta i = 0$, $\Delta j < 0$ and $\Delta k = 0$. Furthermore, we do not read any new value of A', so $\Delta n = 0$.

Type 3 short consistency check: In this case the short consistency check is run for the same slope, but pin is moved, thus the new value i' is larger than the old i. By our simplifying assumption we do not decrease the last slope, just place new i' on it, i.e. we set $A[i'] = A[i] + (i' - i)$, i.e., we take new j such that $\Delta j = \Delta i$. As n only increases, $\Delta n \geq 0$. Concerning k, recall again that $k = \min(n - i + 1, \log^2 n)$, hence $0 \geq \Delta k \geq -\Delta i$. □

In the following, we describe how to update j_{best} and L in those three different cases so that (15a)–(15c) are preserved.

Type 1 Updates In this case we do not need any update, as described in Algorithm 9.

Lemma 15 *Suppose that we are to make Type* 1 *short consistency check and all* (15a)–(15e) *hold. Then* (15a)–(15c) *are preserved and the amortised cost is at most* Δn.

Proof Let us inspect the change of potential:

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

By Lemma 14 we know that $\Delta j = 0$ and $\Delta k \leq \Delta n$, we do not change j_{best} nor L so $\Delta L = \Delta j_{best} = 0$. Hence

$$\begin{aligned}\Delta p &\leq \Delta n - 0 + 0 - 0\\ &= \Delta n.\end{aligned}$$

Concerning the invariants: as L is unchanged and $\Delta k \geq 0$ by Lemma 14 we get that (15a) is preserved. Similarly, since we do not change j, j_{best}, L, the (15b)–(15c) are preserved. □

This allows calculating the whole cost of answering Type 1 short consistency check.

Corollary 3 *In Type* 1 *of short consistency check the amortised cost of* TYPE-1-UPDATE-j_{best} *and* COMMON-SHORT-CONSISTENCY-CHECK *is covered by the released credit. The* TYPE-1-UPDATE-j_{best} *followed by* COMMON-SHORT-CONSISTENCY-CHECK *preserves* (15a)–(15e) *and returns the correct answer to short consistency check.*

Proof By Lemma 15 the update of j_{best} and L has amortised cost at most Δn. By Lemma 13 the amortised cost of COMMON-SHORT-CONSISTENCY-CHECK is

Algorithm 10 TYPE-2-UPDATE-j_{best}

```
1: if $j + \log^2 n < j_{best}$ then
2:     $j_{best} \leftarrow j, L \leftarrow 0$
3: end if
```

at most 6. On the other hand, by Lemma 14 we know that $8\Delta n \geq 6 + \Delta n$ credit is issued, which suffice to pay for the amortised cost.

Concerning the correctness, by Lemma 15 the (15a)–(15c) are satisfied after TYPE-1-UPDATE-j_{best} which by Lemma 13 means that after COMMON-SHORT-CONSISTENCY-CHECK all (15a)–(15e) hold and the answer to short consistency check is correct. □

Type 2 Updates Since j is decreased, it might be that j and j_{best} no longer satisfy (15b), (as $j + \log^2 n < j_{best}$). In such a case we set $j \leftarrow j_{best}$ and $L \leftarrow 0$, see Algorithm 10.

Lemma 16 *Assume that all* (15a)–(15e) *hold and we are to make Type* 2 *short consistency check. Then after* TYPE-2-UPDATE-j_{best} *the* (15a) *and* (15c) *are preserved. The amortised cost is at most* $|\Delta j| + 1$.

Proof Suppose that $j + \log^2 n \geq j_{best}$. The invariants (15b)–(15c) hold by assumption, as none of i, j_{best}, L and k was modified. For (15c) note that $j_{best} \leq j + \log^2 n$ holds by case assumption and $j \leq j_{best}$ held by assumption even before the decrement of j, so it holds now as well.

The change of the potential: by Lemma 14, we know that $\Delta i = \Delta k = \Delta n = 0$ and $\Delta j < 0$. Since L and j_{best} were not changed, we have

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

The cost is 1 for the comparison and so the amortised cost is $|\Delta j| + 1$.

If $j + \log^2 n < j_{best}$ then after setting $j_{best} \leftarrow j$ and $L \leftarrow 0$ the (15a)–(15c) trivially hold. The change of potential is

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

By Lemma 14 we know that $\Delta k = 0$. As $j + \log^2 n < j_{best}$ we obtain that $\Delta j_{best} < -\log^2 n$. Since L was reset to 0 we have $-\Delta L = -(-L_0) = L_0$, where L_0 was the previous value of L. We know that $L_0 \leq k \leq \log^2 n$ and so

$$\begin{aligned}\Delta p &< 0 + \log^2 n - \log^2 n - \Delta j \\ &= |\Delta j|.\end{aligned}$$

There is additional cost 1 for the comparison of j and j_{best} (we hide the cost of changing j_{best} and L in it). Hence the amortised cost is at most $1 + |\Delta j|$. □

Algorithm 11 TYPE-3-UPDATE-j_{best}

```
1: $j_{best} \leftarrow j_{best} + \Delta j$, $L \leftarrow L - \Delta j$
2: if $L < 0$ then
3:     $j_{best} \leftarrow j$, $L \leftarrow 0$
4: end if
```

Corollary 4 *In Type* 2 *of short consistency check the amortised cost of* TYPE-2-UPDATE-j_{best} *and* COMMON-SHORT-CONSISTENCY-CHECK *is covered by the issued credit. The* TYPE-2-UPDATE-j_{best} *followed by* COMMON-SHORT-CONSISTENCY-CHECK *preserve* (15a)–(15e) *and correctly answers short consistency check.*

Proof By Lemma 16, the update of j_{best} and L has amortised cost at most $|\Delta j| + 1$. By Lemma 13, the amortised cost of COMMON-SHORT-CONSISTENCY-CHECK is 6. On the other hand, by Lemma 14, $8|\Delta j| \geq 7 + \Delta j$ credit is issued, which suffice to pay for the amortised cost.

Concerning the correctness, by Lemma 16 after TYPE-2-UPDATE-j_{best} the (15a)–(15c) hold and so by Lemma 13 adter COMMON-SHORT-CONSISTENCY-CHECK all (15a)–(15e) hold and the answer to short consistency check is correct. □

Type 3 Updates It is left to show how to update j_{best} and L in the Type 3 short consistency check, see Algorithm 11. In this case both j and i were increased by the same value Δj, see Lemma 14. This means that the new $A'[j \mathinner{..} j + k - 1]$ and $A'[i \mathinner{..} i + k - 1]$ are the suffixes of the old ones. In particular, $A'[j_{best} \mathinner{..} j_{best} + L - 1]$ has nothing to do with $A'[i \mathinner{..} i + L - 1]$; still, if we also increase j_{best} by Δj then the new $A'[j_{best} \mathinner{..} j_{best} + L - 1]$ is also a suffix of the old one. Unfortunately, as every table we consider is a suffix of the old one, we have to decrease L by Δj as well. If this turns L non-positive then $A'[j_{best} \mathinner{..} j_{best} + L - 1]$ is empty and we reset j_{best} to j and L to 0.

Lemma 17 *Suppose that* (15a)–(15e) *hold and we are to make Type* 3 *short consistency check. Then* TYPE-3-UPDATE-j_{best} *preserves* (15a)–(15c). *The amortised cost is at most* $1 + \Delta j$.

Proof Consider the case in which j_{best} and L are not reset. By Lemma 14 we get that k is decreased by at most Δj, while we decrease L by Δj, hence (15a) is preserved. Concerning (15b) let L', i' and j'_{best} be the previous values of L, i and j_{best}. Then $A'[j_{best} \mathinner{..} j_{best} + L - 1]$ is an ending block of $A'[j'_{best} \mathinner{..} j'_{best} + L' - 1]$ and $A'[i \mathinner{..} i + L - 1]$ is an ending block of $A'[i' \mathinner{..} i' + L' - 1]$. Hence $A'[j_{best} \mathinner{..} j_{best} + L - 1] = A'[i \mathinner{..} i + L - 1]$ follows from $A'[j'_{best} \mathinner{..} j'_{best} + L' - 1] = A'[i' \mathinner{..} i' + L' - 1]$. So (15b) is preserved. For (15c) note that we decremented j and j_{best} by the same value Δj, so (15c) is preserved.

Concerning the change of potential in this case,

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

By Lemma 14 $\Delta k \leq 0$. We decrease L by Δj, so $\Delta L = -\Delta j$ and increase j_{best} by Δj, so $\Delta j_{best} = \Delta j$. Hence

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

The additional cost is 1 for the test, so the amortised cost is at most $\Delta j + 1$.

Now consider the case in which after the decrement by Δj the L is non-positive, i.e., we reset j_{best} to j and L to 0. Then (15a)–(15b) hold trivially, as $L = 0$, and (15c) holds because $j = j_{best}$. Concerning the cost, we pay 1 for comparisons and the change of potential is:

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

By Lemma 14 $\Delta k \leq 0$. Since decreasing L by Δj made it non-positive and then we set it to 0, i.e., increase L, so $\Delta L \geq -\Delta j$. Lastly, $j_{best} - j$ is now equal 0 and used to be non-negative by (15c), so $\Delta j_{best} - \Delta j \leq 0$. Hence

$$\begin{aligned}\Delta p &\leq 0 + \Delta j + 0 \\ &= \Delta j.\end{aligned}$$

So the amortised cost is at most $1 + \Delta j$. □

Corollary 5 *In Type* 3 *of short consistency check the amortised cost of* TYPE-3-UPDATE-j_{best} *and* COMMON-SHORT-CONSISTENCY-CHECK *is covered by the issued credit. The* TYPE-3-UPDATE-j_{best} *followed by* COMMON-SHORT-CONSISTENCY-CHECK *preserve* (15a)–(15e) *and returns a proper answer to short consistency check.*

Proof By Lemma 17 the update of j_{best} and L has amortised cost at most $1 + \Delta j$. By Lemma 13 the amortised cost of COMMON-SHORT-CONSISTENCY-CHECK is at most 6. On the other hand, by Lemma 14 we obtain that at least $8\Delta j \geq 7 + \Delta j$ credit is issued, which suffice to pay for the amortised cost.

Concerning the correctness, by Lemma 16 after TYPE-3-UPDATE-j_{best} the (15a)–(15c) hold and so by Lemma 13 after COMMON-SHORT-CONSISTENCY-CHECK all (15a)–(15e) hold and furthermore the answer to the short consistency check is correct. □

In the end, the short consistency check is performed as follows: depending on which type it is, we run one of TYPE-1-UPDATE-j_{best}, TYPE-2-UPDATE-j_{best}, TYPE-3-UPDATE-j_{best}. Afterwards we apply COMMON-SHORT-CONSISTENCY-CHECK. By Corollary 3–5 the answer returned to short consistency check is correct and the issued credit covers the whole cost. Since the issued credit is linear, we are done.

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.

Acknowledgements 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 partially supported by the Israeli Centers of Research Excellence (I-CORE) program, Center No. 4/11.

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). http://drops.dagstuhl.de/opus/volltexte/2009/1825
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 (1999)
4. Crochemore, M., Rytter, W.: Jewels of Stringology. World Scientific Publishing Company, Singapore (2002)
5. Crochemore, M., Hancart, C., Lecroq, T.: Algorithms on Strings. Cambridge University Press, Cambridge (2007)
6. Crochemore, M., Iliopoulos, C., Pissis, S., Tischler, G.: Cover array string reconstruction. In: CPM 2010. Lecture Notes in Computer Science, vol. 6129, pp. 251–259. Springer, Berlin (2010)
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. In: 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. RAIRO Theor. Inform. Appl. **43**(2), 281–297 (2009). doi:10.1051/ita:2008030
10. Farach, M.: Optimal suffix tree construction with large alphabets. In: Proceedings of FOCS '97, pp. 137–143. IEEE Computer Society, Washington (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:10.1016/S0022-0000(05)80064-9
13. Hancart, C.: On Simon's string searching algorithm. Inf. Process. Lett. **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). doi:10.1007/978-3-642-00982-2_36
15. I, T., Inenaga, S., Bannai, H., Takeda, M.: Verifying and enumerating parameterized border arrays. Theor. Comput. Sci. **412**(50), 6959–6981 (2011). doi:10.1016/j.tcs.2011.09.008
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 (1972). doi:10.1145/800152.804905
17. Knuth, D.E., Morris, J.H. Jr., Pratt, V.R.: Fast pattern matching in strings. SIAM J. Comput. **6**(2), 323–350 (1977)
18. Matiyasevich, Y.: Real-time recognition of the inclusion relation. J. Sov. Math. **1**, 64–70 (1973). Published (in Russian) in Zap. Nauč. Semin. POMI, **20**, 104–114 (1971)
19. McCreight, E.M.: A space-economical suffix tree construction algorithm. J. ACM **23**(2), 262–272 (1976). doi:10.1145/321941.321946
20. Moore, D., Smyth, W.F., Miller, D.: Counting distinct strings. Algorithmica **23**(1), 1–13 (1999). http://link.springer.de/link/service/journals/00453/bibs/23n1p1.html
21. Morris, J.H. Jr., Pratt, V.R.: A linear pattern-matching algorithm. Tech. Rep. 40, University of California, Berkeley (1970)
22. Pagh, R., Rodler, F.F.: Cuckoo hashing. J. Algorithms **51**(2), 122–144 (2004). doi:10.1016/j.jalgor.2003.12.002
23. Simon, I.: String matching algorithms and automata. In: Results and Trends in Theoretical Computer Science. LNCS, vol. 812, pp. 386–395. Springer, Berlin (1994)
24. Ukkonen, E.: On-line construction of suffix trees. Algorithmica **14**(3), 249–260 (1995). doi:10.1007/BF01206331