Sorting suffixes of two-pattern strings

Frantisek Franek W. F. Smyth [*]

Algorithms Research Group
Department of Computing & Software
McMaster University
Hamilton, Ontario
Canada L8S 4L7

April 19, 2004

Abstract

Recently, several authors presented linear recursive algorithms for sorting suffixes of a string. All these algorithms employ a similar three-step approach, based on an initial division of the suffixes of $\boldsymbol{x}$ into two sets: in step 1 sort the first set using recursive reduction of the problem, in step 2 determine the order of the suffixes in the second set based on the order of the suffixes in the first set, and in step 3 merge the two sets together. To optimize such an algorithm either for space or time, it may not be sufficient to optimize one of the three steps, since in doing so, one might increase the resources required for the others to an unacceptable extent.

Franek, Lu, and Smyth introduced two-pattern strings as a generalization of Sturmian strings. Like Sturmian strings, two-pattern strings are generated by iterated morphisms, but they exhibit a much richer structure.

In this paper we show that the suffixes of two-pattern strings can be sorted in linear time using a variant of the three step approach outlined above. It turns out that, given the order of the suffixes in a two-pattern string, one can almost directly list in linear time all the suffixes of its expansion under a two-pattern morphism.

[*]also Department of Computing, Curtin University, Perth WA 6845, Australia.

1 Introduction

Ever since Manber and Myers in [MM93] introduced suffix arrays as data structures comparable to suffix trees for most pattern matching tasks in strings, yet requiring significantly less memory, the search was on for a linear time algorithm for their construction. Such an algorithm for suffix tree construction had been known since 1997 [F97]. In 2003 to our knowledge three different groups of researchers independently proposed linear recursive algorithms to sort string suffixes: [KA03, KSPP03, KS03]. Though different, all three algorithms employ three steps, based on a separation of the suffixes into two sets. In step 1 the first set is ordered using recursive reduction of the problem, in step 2 the suffixes of the second set are sorted based on the order of the suffixes in the first set, and in step 3 both ordered sets are merged together. The fact that all three algorithms follow this basic approach, yet use a completely different separation into sets, a different way of ordering the second set based on the first set, and a different merge technique, points to some common fundamental aspect of these algorithms. To optimize such an algorithm either for space or time, it may not be sufficient to optimize one of the three steps, since in doing so, one might increase the resources required for the others to an unacceptable extent.

Two-pattern strings were introduced in [FLS03] as a generalization of Sturmian strings. Like Sturmian strings, two-pattern strings are generated by iterated morphisms, but they exhibit a much richer structure. It was shown in [FLS04] that the iterated construction of these strings could be used to compute all the repetitions and near-repetitions in time linear in string length.

This paper was motivated by our investigation of the three different linear suffix sorting algorithms discussed above and our desire to fully understand the underlying phenomena. Thus, we investigated whether the recursive nature of two-pattern strings could be used in sorting of the suffixes in the approach of the three algorithms mentioned. As it turned out, the "natural" recursive reduction of two-pattern strings can be used for step 1, and then steps 2 and 3 can be simplified into a single step: from having the suffixes of the reduced string ordered, one can almost directly list the suffixes of the two-pattern string in the right order.

For the sake of completeness, let us recall the definition of a two-pattern string [FLS03]. Throughout this paper, a **binary string** means a string over the alphabet $\{a, b\}$.

Definition 1.1 *An ordered pair* $(\boldsymbol{p}, \boldsymbol{q})$ *of nonempty binary strings is said to be* **suitable** *if and only if*

- $\boldsymbol{p}$ *is* **primitive** (*that is,* $\boldsymbol{p}$ *has no nonempty border*)*;*
- $\boldsymbol{p}$ *is not a suffix of* $\boldsymbol{q}$*;*
- $\boldsymbol{q}$ *is neither a prefix nor a suffix of* $\boldsymbol{p}$*;*
- $\boldsymbol{q}$ *is not* $\boldsymbol{p}$*-regular.*

Definition 1.2 *A binary string* $\boldsymbol{q}$ *is said to be* $\boldsymbol{p}$**-regular** *if and only if* $\boldsymbol{q} = \boldsymbol{upvu}$ *for some choice of* (*possibly empty*) *substrings* $\boldsymbol{u}$ *and* $\boldsymbol{v}$*.*

Definition 1.3 $\sigma = [\boldsymbol{p}, \boldsymbol{q}, i, j]_\lambda$ *is an* **expansion of scope** $\boldsymbol{\lambda}$*, if* $(\boldsymbol{p}, \boldsymbol{q})$ *is suitable,* $|\boldsymbol{p}| \leq \lambda$*,* $|\boldsymbol{q}| \leq \lambda$*, and* $1 \leq i < j$ *are integers.*

Definition 1.4 *A binary string* $\boldsymbol{x}$ *is a* **two-pattern string of scope** $\boldsymbol{\lambda}$ *if there exists a sequence* $\{\sigma_1, \sigma_2, \ldots, \sigma_m\}$ *of expansions of scope* λ *so that* $\boldsymbol{x} = \sigma_1 \circ \cdots \circ \sigma_m(a)$*.*

It was mentioned at the end of [FLS03] that if the definition of $\boldsymbol{p}$-regularity were made more restrictive, a larger class of two-pattern string could be obtained. The more restrictive definition, sufficient to give two-pattern strings all their desired properties, contained a few typographical errors as it was given in [FLS03], and so we provide a corrected definition here:

Definition 1.5 *A binary string* $\boldsymbol{q}$ *is said to be* $\boldsymbol{p}$**-regular** *(*$\boldsymbol{p}$ *a binary string) if and only if there exist (possibly empty) strings* $\boldsymbol{u}, \boldsymbol{v}$ *together with nonnegative integers* $n_1, n_2, \ldots, n_k$*,* $k \geq 1$*,* $r \geq 1$*, such that*

- *the integers* n_i *assume at most two distinct values — that is,*
$$\big|\{n_i : \ i \in 1..k\}\big| \leq 2;$$
- $\boldsymbol{q} = (\boldsymbol{up}^r\boldsymbol{vp}^{n_1})(\boldsymbol{up}^r\boldsymbol{vp}^{n_2}) \cdots (\boldsymbol{up}^r\boldsymbol{vp}^{n_k})\boldsymbol{u}$ *for some* $\boldsymbol{u}$*,* $\boldsymbol{v}$*,* $r \geq 0$*, where* $\boldsymbol{v} = \varepsilon$ *if* $r = 0$*.*

It was shown in [FLS03] that complete two-pattern strings can be recognized in linear time: the recognition algorithm outputs an essentially unique sequence of expansions. So in the following we can assume that not only do we have a complete two-pattern string, but also the sequence of expansions that iteratively generates the string.

In the next section we describe the principles underlying the algorithm for sorting suffixes of a two-pattern string. In Section 3 we provide an overview of the algorithm itself, while Section 4 gives proofs of some of the main lemmas on which the algorithm is based.

2 The Principles Underlying the Algorithm

For the sake of clarity and brevity, we introduce several symbols: we use the symbol $\boldsymbol{u} < \boldsymbol{v}$ for strings $\boldsymbol{u}$, $\boldsymbol{v}$ to express that $\boldsymbol{u}$ is lexicographically smaller than $\boldsymbol{v}$. We use the symbol $\boldsymbol{u} \prec \boldsymbol{v}$ (or $\boldsymbol{u} \succ \boldsymbol{v}$) to express the fact that $\boldsymbol{u} < \boldsymbol{v}$ yet $\boldsymbol{u}$ is not a prefix of $\boldsymbol{v}$ (or $\boldsymbol{v} < \boldsymbol{u}$ yet $\boldsymbol{v}$ is not a prefix of $\boldsymbol{u}$). Note that $\boldsymbol{u} < \boldsymbol{v}$ iff ($\boldsymbol{u} \prec \boldsymbol{v}$ or $\boldsymbol{u}$ is a prefix of $\boldsymbol{v}$). We use the symbol $\boldsymbol{u} \succ\!\prec \boldsymbol{v}$ to indicate that either $\boldsymbol{u} \prec \boldsymbol{v}$ or $\boldsymbol{u} \succ \boldsymbol{v}$.

For a given expansion $\sigma = [\boldsymbol{p}, \boldsymbol{q}, i, j]_\lambda$, we will write $\overline{\boldsymbol{u}}$ instead of $\sigma(\boldsymbol{u})$ if it causes no confusion. Note that we define $\overline{\varepsilon} = \varepsilon$.

For a binary string $\boldsymbol{u}$, we will use $\hat{\boldsymbol{u}}$ to denote its ones-complement; that is, the string formed by interchanging a's and b's in $\boldsymbol{u}$.

In accordance with [FLS03], if $\boldsymbol{x}$, $\boldsymbol{y}$ are complete two-pattern strings, σ an expansion, and $\boldsymbol{y} = \sigma(\boldsymbol{x})$, then the occurrences of copies of $\boldsymbol{p}$ and copies of $\boldsymbol{q}$ in the concatenation of blocks $\boldsymbol{p}^i\boldsymbol{q}$ and $\boldsymbol{p}^j\boldsymbol{q}$ as defined by $\sigma(\boldsymbol{x})$ are called **restrained** copies. Any other occurrence of $\boldsymbol{p}$ or $\boldsymbol{q}$ is referred to as **free**. A consecutive sequence of restrained copies of $\boldsymbol{p}$'s and/or $\boldsymbol{q}$'s will also be referred to as a **restrained configuration** or a **restrained substring** of $\boldsymbol{y}$.

Throughout the following discussion we assume that the scope λ is fixed and that $\boldsymbol{y} = \sigma(\boldsymbol{x})$, where $\boldsymbol{x}$ is a complete two-pattern string of scope λ and $\sigma = [\boldsymbol{p}, \boldsymbol{q}, i, j]_\lambda$ an expansion of scope λ. Moreover we assume that all suffixes of $\boldsymbol{x}$ are lexicographically sorted: $\boldsymbol{\rho}_1 < \cdots < \boldsymbol{\rho}_{|\boldsymbol{x}|}$. We then describe how to order the suffixes of $\boldsymbol{y}$. We may assume further that $\boldsymbol{q} < \boldsymbol{p}$: if it were not, then $\hat{\boldsymbol{q}} < \hat{\boldsymbol{p}}$, we sort all of the suffixes of $\hat{\boldsymbol{y}} = \hat{\sigma}(\boldsymbol{x})$, where $\hat{\sigma} = [\hat{\boldsymbol{p}}, \hat{\boldsymbol{q}}, i, j]_\lambda$, and reversing the order, we get all suffixes of $\boldsymbol{y}$ ordered properly (by Lemma 4.2).

Since $\boldsymbol{q} < \boldsymbol{p}$, by Lemma 4.1, for any suffixes $\boldsymbol{\rho}_1$, $\boldsymbol{\rho}_2$ of $\boldsymbol{x}$, if $\boldsymbol{\rho}_1 < \boldsymbol{\rho}_2$, then

$\overline{\boldsymbol{\rho}_1} < \overline{\boldsymbol{\rho}_2}$.

We put all the suffixes of $\boldsymbol{y}$ into disjoint buckets of five types $\boldsymbol{A}$–$\boldsymbol{E}$:

- For $\boldsymbol{\delta}$ a suffix of $\boldsymbol{p}$, we define
  - $\boldsymbol{A}_{\boldsymbol{\delta},k} = \{\boldsymbol{\delta p}^k\boldsymbol{q}\overline{\boldsymbol{\rho}} :\ \boldsymbol{\rho}$ a proper suffix of $\boldsymbol{x}$ *or* $\boldsymbol{\rho} = \varepsilon\}$, $0 < k < i$;
  - $\boldsymbol{A}_{\boldsymbol{\delta},i} = \{\boldsymbol{\delta p}^i\boldsymbol{q}\overline{\boldsymbol{\rho}} :\ \boldsymbol{\rho}$ a proper suffix of $\boldsymbol{x}$ *or* $\boldsymbol{\rho} = \varepsilon\}$, $\boldsymbol{\delta}$ a suffix of $\boldsymbol{q}$;
  - $\boldsymbol{A}_{\boldsymbol{\delta},i} = \{\boldsymbol{\delta p}^i\boldsymbol{q}\overline{\boldsymbol{\rho}} :\ b\boldsymbol{\rho}$ a proper suffix of $\boldsymbol{x}, \boldsymbol{\rho}$ can be empty$\}$, $\boldsymbol{\delta}$ not a suffix of $\boldsymbol{q}$;
  - $\boldsymbol{A}_{\boldsymbol{\delta},k} = \{\boldsymbol{\delta p}^k\boldsymbol{q}\overline{\boldsymbol{\rho}} :\ b\boldsymbol{\rho}$ a proper suffix of $\boldsymbol{x}, \boldsymbol{\rho}$ can be empty$\}$, $i < k < j$.
  - $\boldsymbol{B}_{\boldsymbol{\delta}} = \{\boldsymbol{\delta q}\overline{\boldsymbol{\rho}} : \boldsymbol{\rho}$ a proper nontrivial suffix of $\boldsymbol{x}\}$.
- For $\boldsymbol{\delta}$ a suffix of $\boldsymbol{q}$, we define
  - $\boldsymbol{C}_{\boldsymbol{\delta}} = \{\boldsymbol{\delta p}^i\boldsymbol{q}\overline{\boldsymbol{\rho}} :\ a\boldsymbol{\rho}$ a proper suffix of $\boldsymbol{x}, \boldsymbol{\rho}$ can be empty$\}$, $\boldsymbol{\delta}$ not a suffix of $\boldsymbol{p}$;
  - $\boldsymbol{D}_{\boldsymbol{\delta}} = \{\boldsymbol{\delta p}^j\boldsymbol{q}\overline{\boldsymbol{\rho}} :\ b\boldsymbol{\rho}$ a proper suffix of $\boldsymbol{x}, \boldsymbol{\rho}$ can be empty$\}$.
- Finally we define
  - $\boldsymbol{E} = \{\boldsymbol{\delta q} :\ \boldsymbol{\delta}$ a suffix of $\boldsymbol{p}\} \cup \{\boldsymbol{\delta} :\ \boldsymbol{\delta}$ a suffix of $\boldsymbol{q}\}$.

(where the term *proper suffix* refers to a suffix that is not equal to the whole string and the term *trivial suffix* refers to the empty suffix).

It is easy to check that any suffix of $\boldsymbol{y}$ belongs to one of the buckets $\boldsymbol{A}$–$\boldsymbol{E}$. We are going to order the suffixes in buckets $\boldsymbol{A}$–$\boldsymbol{D}$ based on the ordering of the suffixes for $\boldsymbol{x}$ (Step 1), then merge in the suffixes from $\boldsymbol{E}$ (Steps 2 & 3); since $|\boldsymbol{E}| \leq 2\lambda$, this will not destroy the linearity of the algorithm. Note that the order within each bucket is determined by the order of suffixes of $\boldsymbol{x}$:

in the bucket $\boldsymbol{A}_{\boldsymbol{\delta},k}$:	$\boldsymbol{\delta p}^k\boldsymbol{q}\overline{\boldsymbol{\rho}_1} < \boldsymbol{\delta p}^k\boldsymbol{q}\overline{\boldsymbol{\rho}_2}$ if $\boldsymbol{\rho}_1 < \boldsymbol{\rho}_2$;
in the bucket $\boldsymbol{B}_{\boldsymbol{\delta}}$:	$\boldsymbol{\delta q}\overline{\boldsymbol{\rho}_1} < \boldsymbol{\delta q}\overline{\boldsymbol{\rho}_2}$ if $\boldsymbol{\rho}_1 < \boldsymbol{\rho}_2$;
in the bucket $\boldsymbol{C}_{\boldsymbol{\delta}}$:	$\boldsymbol{\delta p}^i\boldsymbol{q}\overline{\boldsymbol{\rho}_1} < \boldsymbol{\delta p}^i\boldsymbol{q}\overline{\boldsymbol{\rho}_2}$ if $\boldsymbol{\rho}_1 < \boldsymbol{\rho}_2$;
and in the bucket $\boldsymbol{D}_{\boldsymbol{\delta}}$:	$\boldsymbol{\delta p}^j\boldsymbol{q}\overline{\boldsymbol{\rho}_1} < \boldsymbol{\delta p}^j\boldsymbol{q}\overline{\boldsymbol{\rho}_2}$ if $\boldsymbol{\rho}_1 < \boldsymbol{\rho}_2$.

Thus, it is straightforward to list the suffixes in each bucket in the correct order, given the order of the suffixes of $\boldsymbol{x}$.

We make use of the following notation: if X, Y are sets of suffixes of $\boldsymbol{y}$, we write $X \lll Y$ iff $(\forall x \in X)(\forall y \in Y)(x < y)$. The major observation our algorithm is based on is that the buckets are ordered by $\lll$; that is, pairwise orderings can be made between bucket pairs of types

$$\boldsymbol{AA}, \boldsymbol{AB}, \boldsymbol{AC}, \boldsymbol{AD}, \boldsymbol{BB}, \boldsymbol{BC}, \boldsymbol{BD}, \boldsymbol{CC}, \boldsymbol{CD}, \boldsymbol{DD}, \tag{1}$$

based on five mutually exclusive (and exhaustive) conditions on any pair $\boldsymbol{\delta_1}$, $\boldsymbol{\delta_2}$ of suffixes of $\boldsymbol{p}$ and/or $\boldsymbol{q}$:

(C1) $\boldsymbol{\delta}_1 \prec \boldsymbol{\delta}_2$;

(C2) $\boldsymbol{\delta}_1 \succ \boldsymbol{\delta}_2$;

(C3) $\boldsymbol{\delta}_1$ a proper prefix of $\boldsymbol{\delta}_2$;

(C4) $\boldsymbol{\delta}_2$ a proper prefix of $\boldsymbol{\delta}_1$;

(C5) $\boldsymbol{\delta}_1 = \boldsymbol{\delta}_2 = \boldsymbol{\delta}$.

Observe that, given $\boldsymbol{\delta_1}$ and $\boldsymbol{\delta_2}$, to determine which of these conditions holds requires at most λ letter comparisons (since $|\boldsymbol{\delta_1}| \leq \lambda$, $|\boldsymbol{\delta_2}| \leq \lambda$).

Thus, for example, two $\boldsymbol{A}$ buckets can be ordered as follows:

(C1) $\boldsymbol{A}_{\boldsymbol{\delta}_1,k_1} \lll \boldsymbol{A}_{\boldsymbol{\delta}_2,k_2}$.

(C2) $\boldsymbol{A}_{\boldsymbol{\delta}_2,k_2} \lll \boldsymbol{A}_{\boldsymbol{\delta}_1,k_1}$.

(C3) Let $\boldsymbol{\delta}_2 = \boldsymbol{\delta}_1\boldsymbol{\delta}'_1$ for some nonempty $\boldsymbol{\delta}'_1$:

- (a) if $\boldsymbol{\delta}'_1 \prec \boldsymbol{p}$, then $\boldsymbol{A}_{\boldsymbol{\delta}_2,k_2} \lll \boldsymbol{A}_{\boldsymbol{\delta}_1,k_1}$;
- (b) otherwise, $\boldsymbol{A}_{\boldsymbol{\delta}_1,k_1} \lll \boldsymbol{A}_{\boldsymbol{\delta}_2,k_2}$.

(C4) Let $\boldsymbol{\delta}_1 = \boldsymbol{\delta}_2\boldsymbol{\delta}'_2$ for some nonempty $\boldsymbol{\delta}'_2$:

- (a) If $\boldsymbol{\delta}'_2 \prec \boldsymbol{p}$, then $\boldsymbol{A}_{\boldsymbol{\delta}_1,k_1} \lll \boldsymbol{A}_{\boldsymbol{\delta}_2,k_2}$;
- (b) otherwise, $\boldsymbol{A}_{\boldsymbol{\delta}_2,k_2} \lll \boldsymbol{A}_{\boldsymbol{\delta}_1,k_1}$.

(C5) (a) If $k_1 < k_2$, then $\boldsymbol{A}_{\boldsymbol{\delta},k_1} \lll \boldsymbol{A}_{\boldsymbol{\delta},k_2}$;

- (b) if $k_1 = k_2$, then $\boldsymbol{A}_{\boldsymbol{\delta},k_1} = \boldsymbol{A}_{\boldsymbol{\delta},k_2}$;

(c) if $k_1 > k_2$, then $\boldsymbol{A}_{\boldsymbol{\delta},k_2} \ll \boldsymbol{A}_{\boldsymbol{\delta},k_1}$.

It is not hard to prove that this ordering is correct. The demonstration for cases (C1), (C2) and (C5) is straightforward. For (C3), observe that we are comparing $\boldsymbol{\delta}_1\boldsymbol{p}^{k_1}\boldsymbol{q}\cdots$ with $\boldsymbol{\delta}_2\boldsymbol{p}^{k_2}\boldsymbol{q}\cdots$, hence $\boldsymbol{p}^{k_1}\boldsymbol{q}\cdots$ with $\boldsymbol{\delta}_1'\boldsymbol{p}^{k_2}\boldsymbol{q}\cdots$. Since $\boldsymbol{\delta}_1'$ is a suffix of $\boldsymbol{\delta}_2$, it is also a suffix of $\boldsymbol{p}$ and so cannot be a prefix of $\boldsymbol{p}$. It follows that either $\boldsymbol{\delta}_1' \prec \boldsymbol{p}$ or $\boldsymbol{\delta}_1' \succ \boldsymbol{p}$, and the result follows. The proof for (C4) is exactly analogous.

Furthermore the $\boldsymbol{AA}$ ordering is efficient, since the cases (a) and (b) in (C3) and (C4) can be processed in at most λ constant-time steps in addition to the λ steps that may be required to identify which condition holds: thus a total of at most 2λ steps altogether.

The results for the other pairs listed in (1) are similar: the details vary slightly from one case to another. The main result is that any of the pairs can be processed in at most 3λ steps, a constant. To avoid distracting the reader with unnecessary and uninteresting detail, we do not include the other cases here. For those details, please access

`http://www.cas.mcmaster.ca/~franek/web-publications.html`

3 The High-Level Logic of the Algorithm

We describe only the recursive step (Step 1) that takes us from $\boldsymbol{x}$ and its sorted suffixes to the corresponding sorted suffixes of $\boldsymbol{y} = \sigma(\boldsymbol{x})$, where $\sigma = [\boldsymbol{p}, \boldsymbol{q}, i, j]_\lambda$. Recall that we assume $\boldsymbol{q} < \boldsymbol{p}$.

1. Create names $(A, \boldsymbol{\delta})$ for every suffix $\boldsymbol{\delta}$ of $\boldsymbol{p}$. (*This requires at most λ steps.*)

2. Sort the names according to the order described in the previous section for mutual comparison of the four $\boldsymbol{A}$ buckets. (*This requires at most $2\lambda^3$ steps as we are sorting λ names and each comparison requires $\leq 2\lambda$ steps.*)

3. Replace every name $(A, \boldsymbol{\delta})$ by a sequence of names $(A, \boldsymbol{\delta}, k)$, $1 \leq k < j$. Let us call the resulting sequence BUCKETS. (*Now we have the names of $\boldsymbol{A}$ buckets in the proper order. This requires at most $|\boldsymbol{y}|$ steps as the size of* BUCKETS *is* $\leq |\boldsymbol{y}|$.)

4. Create names $(B, \boldsymbol{\delta})$ for every suffix $\boldsymbol{\delta}$ of $\boldsymbol{p}$. (*This requires at most* λ *steps.*)

5. Merge into BUCKETS all names $(B, \boldsymbol{\delta})$ according to comparisons as described in comparing $\boldsymbol{A}$ buckets to $\boldsymbol{B}$ buckets. (*This requires at most* $|\text{BUCKETS}|3\lambda^2$ *steps, as we are merging in* λ *names and each comparison requires* $\leq 3\lambda$ *steps, hence at most* $|\boldsymbol{y}|3\lambda^2$ *steps.*)

6. Create names $(C, \boldsymbol{\delta})$ for every suffix $\boldsymbol{\delta}$ of $\boldsymbol{q}$ that is not a suffix of $\boldsymbol{p}$. (*This requires at most* λ^2 *steps.*)

7. Merge into BUCKETS all names $(C, \boldsymbol{\delta})$ according to comparisons as described in comparing $\boldsymbol{A}$ buckets to $\boldsymbol{C}$ buckets and $\boldsymbol{B}$ buckets to $\boldsymbol{C}$ buckets. (*This requires at most* $|\text{BUCKETS}|3\lambda^2$ *steps, hence at most* $|\boldsymbol{y}|3\lambda^2$ *steps.*)

8. Create names $(D, \boldsymbol{\delta})$ for every suffix $\boldsymbol{\delta}$ of $\boldsymbol{q}$. (*This requires at most* λ *steps.*)

9. Merge into BUCKETS all names $(D, \boldsymbol{\delta})$ according to comparisons as described in comparing $\boldsymbol{A}$ buckets to $\boldsymbol{D}$ buckets, $\boldsymbol{B}$ buckets to $\boldsymbol{D}$ buckets, $\boldsymbol{C}$ buckets to $\boldsymbol{D}$ buckets. (*Now we have all bucket names in the proper order. This requires at most* $|\text{BUCKETS}|3\lambda^2$ *steps, hence at most* $|\boldsymbol{y}|3\lambda^2$ *steps.*)

10. Traverse BUCKETS and replace each name by a sequence of suffixes according to the sequence of suffixes of $\boldsymbol{x}$. Let us call this sequence SUFFIXES. (*Now we have all suffixes from buckets* $\boldsymbol{A}$–$\boldsymbol{D}$ *in proper order. This requires at most* $|\boldsymbol{y}|$ *steps as the size of* SUFFIXES *is* $\leq |\boldsymbol{y}|$.)

11. Merge into SUFFIXES the suffixes from the bucket $\boldsymbol{E}$. (*This requires at most* $|\text{SUFFIXES}|4\lambda^2$ *steps, as we are merging in* 2λ *suffixes, each of length* $\leq 2\lambda$*, hence at most* $|\boldsymbol{y}|4\lambda^2$ *steps.*)

SUFFIXES now contains the sorted list of all suffixes of $\boldsymbol{y}$ and it took less than $\alpha|\boldsymbol{y}|$ steps, where we set $\alpha = 2\lambda^3 + 14\lambda^2 + 3\lambda + 2$. Since every reduction of a complete two-pattern string at least halves its length, altogether the algorithm with all iterative steps included took less than $\boldsymbol{\alpha}n + \boldsymbol{\alpha}\frac{n}{2} + \boldsymbol{\alpha}\frac{n}{4} + \cdots < 2\boldsymbol{\alpha}n$ steps, where n is the size of the input string.

4 The Supporting Lemmas

The first lemma establishes that the ordering of suffixes is invariant under an expansion σ.

Lemma 4.1 *Let $\sigma = [\boldsymbol{p}, \boldsymbol{q}, i, j]_\lambda$ be an expansion and $\boldsymbol{q} < \boldsymbol{p}$. Let $\boldsymbol{x}$ and $\boldsymbol{y}$ be two-pattern strings of scope λ and let $\boldsymbol{y} = \sigma(\boldsymbol{x})$. Let $\boldsymbol{\rho}_1$, $\boldsymbol{\rho}_2$ be suffixes of $\boldsymbol{x}$ so that $\boldsymbol{\rho}_1 < \boldsymbol{\rho}_2$. Then $\sigma(\boldsymbol{\rho}_1) < \sigma(\boldsymbol{\rho}_2)$.*

Proof Since $\boldsymbol{\rho}_1 < \boldsymbol{\rho}_2$, either

1. $\boldsymbol{\rho}_1 \prec \boldsymbol{\rho}_2$ and so there exists k such that $(\forall 1 \le m < k)(\boldsymbol{\rho}_1[m] = \boldsymbol{\rho}_2[m])$ and $\boldsymbol{\rho}_1[k] = a$ and $\boldsymbol{\rho}_2[k] = b$. Let $|\sigma(\boldsymbol{\rho}_1[1..k{-}1])| = |\sigma(\boldsymbol{\rho}_2[1..k{-}1])| = n$, and let $r = n{+}i|\boldsymbol{p}|$. Then

   $$\sigma(\boldsymbol{\rho}_1[1..r]) = \sigma(\boldsymbol{\rho}_2[1..r]),$$

   $\sigma(\boldsymbol{\rho}_1[r{+}1..r{+}|\boldsymbol{q}|]) = \boldsymbol{q}$, and $\sigma(\boldsymbol{\rho}_2[r{+}1..r{+}|\boldsymbol{p}|]) = \boldsymbol{p}$. Since $\boldsymbol{q} < \boldsymbol{p}$, it follows that $\boldsymbol{q} \prec \boldsymbol{p}$ and thus $\sigma([1..r{+}|\boldsymbol{q}|]) \prec \sigma([1..r{+}|\boldsymbol{p}|])$, so that $\sigma(\boldsymbol{\rho}_1) \prec \sigma(\boldsymbol{\rho}_2)$.

   or

2. $\boldsymbol{\rho}_1$ is a prefix of $\boldsymbol{\rho}_2$, hence $\sigma(\boldsymbol{\rho}_1)$ a prefix of $\sigma(\boldsymbol{\rho}_2)$, so that $\sigma(\boldsymbol{\rho}_1) < \sigma(\boldsymbol{\rho}_2)$.

$\square$

The next lemma tells us that interchanging a and b in a binary string reverses the order of the suffixes.

Lemma 4.2 *Let $\boldsymbol{\rho}_1 < \cdots < \boldsymbol{\rho}_n$ be the sequence of all suffixes of a binary string $\boldsymbol{u}$ in an ascending lexicographic order. Then $\hat{\boldsymbol{\rho}}_1 > \cdots > \hat{\boldsymbol{\rho}}_n$ is the sequence of all suffixes of $\hat{\boldsymbol{u}}$ in a descending lexicographic order.*

Proof Follows by induction from the fact that $\boldsymbol{x} < \boldsymbol{y}$ iff $\hat{\boldsymbol{x}} > \hat{\boldsymbol{y}}$ for any two binary strings $\boldsymbol{x}$ and $\boldsymbol{y}$. $\square$

The next three lemmas are technical lemmas required for some of the proofs (see website referenced above) that the pairs (1) can be processed correctly in $O(3\lambda)$ time. Essentially these lemmas tell us that the ordering of restrained suffixes of $\boldsymbol{y}$ can be accomplished in at most 2λ constant-time algorithmic steps.

Lemma 4.3 *Let $\boldsymbol{x}$, $\boldsymbol{y}$ be two-pattern strings of scope λ, $\sigma = [\boldsymbol{p}, \boldsymbol{q}, i, j]_\lambda$ an expansion, and $\boldsymbol{y} = \sigma(\boldsymbol{x})$. Let $\boldsymbol{u}$ be a non-empty binary string and let $\boldsymbol{uqp}$ be a suffix of a restrained configuration $\boldsymbol{pqp}$ of $\boldsymbol{y}$ and let $\boldsymbol{qp}$ be a restrained configuration of $\boldsymbol{y}$. Then $\boldsymbol{uqp} \rightarrowtail\!\!\!\!\leftarrowtail \boldsymbol{qp}$ and whether $\boldsymbol{uqp} \prec \boldsymbol{qp}$ or $\boldsymbol{uqp} \succ \boldsymbol{qp}$ can be determined in $\leq 2\lambda$ steps.*

Proof Arguing by contradiction, we are assuming that $\boldsymbol{uqp} \rightarrowtail\!\!\!\!\leftarrowtail \boldsymbol{qp}$ does not hold. Since $\boldsymbol{u}$ is non-empty, it follows that $\boldsymbol{qp}$ must be a prefix of $\boldsymbol{uqp}$. Since $\boldsymbol{u}$ is a prefix of $\boldsymbol{p}$, the last $\boldsymbol{p}$ of $\boldsymbol{qp}$ and the last $\boldsymbol{p}$ of $\boldsymbol{uqp}$ intersect, contradicting the primitiveness of $\boldsymbol{p}$. The second part of the claim follows from the fact that $|\boldsymbol{qp}| \leq 2\lambda$. $\square$

Lemma 4.4 *Let $\boldsymbol{x}$, $\boldsymbol{y}$ be two-pattern strings of scope λ, $\sigma = [\boldsymbol{p}, \boldsymbol{q}, i, j]_\lambda$ an expansion, and $\boldsymbol{y} = \sigma(\boldsymbol{x})$. Let $\boldsymbol{u}$ be a non-empty binary string and let $\boldsymbol{up}$ be a suffix of a restrained configuration $\boldsymbol{qp}$ of $\boldsymbol{y}$. Let $1 \leq k$, and let $\boldsymbol{p}^k\boldsymbol{q}$ be a restrained configuration of $\boldsymbol{y}$. Then $\boldsymbol{up} \rightarrowtail\!\!\!\!\leftarrowtail \boldsymbol{p}^k\boldsymbol{q}$ and whether $\boldsymbol{up} \prec \boldsymbol{p}^k\boldsymbol{q}$ or $\boldsymbol{up} \succ \boldsymbol{p}^k\boldsymbol{q}$ can be determined in $\leq 2\lambda$ steps.*

Proof Arguing by contradiction, we are assuming that $\boldsymbol{up} \rightarrowtail\!\!\!\!\leftarrowtail \boldsymbol{p}^k\boldsymbol{q}$ does not hold. It follows that $\boldsymbol{up}$ is a prefix of $\boldsymbol{p}^k\boldsymbol{q}$ as $\boldsymbol{u}$ is a suffix of $\boldsymbol{q}$ and hence $|\boldsymbol{u}| < |\boldsymbol{q}|$ and $|\boldsymbol{u}| + |\boldsymbol{p}| < k|\boldsymbol{p}| + |\boldsymbol{q}|$.

1. if $|\boldsymbol{u}| < |\boldsymbol{p}|^r$, $1 \leq r \leq k$, r maximal such, then $\boldsymbol{up}$ and the r-th copy of $\boldsymbol{p}$ from $\boldsymbol{p}^k\boldsymbol{q}$ have a non-empty intersection, contradicting the fact that $\boldsymbol{p}$ is primitive.

2. if $|\boldsymbol{u}| = |\boldsymbol{p}|^r$, $1 \leq r \leq k$, r maximal such, then $\boldsymbol{p}$ is a suffix of $\boldsymbol{q}$, a contradiction.

3. Thus $|\boldsymbol{u}| > |\boldsymbol{p}|^k$. Since $\boldsymbol{u}$ is a suffix of a restrained $\boldsymbol{q}$, $\boldsymbol{q} = (\boldsymbol{vp}^k)^t\boldsymbol{w}$ for some $t \geq 1$, $\boldsymbol{w}$ a prefix of $\boldsymbol{vp}^k$.

   (a) $\boldsymbol{w} = \varepsilon$. Then $\boldsymbol{p}$ is a suffix of $\boldsymbol{q}$, a contradiction.

   (b) $\boldsymbol{w}$ is a proper prefix of $\boldsymbol{v}$. Then $\boldsymbol{v} = \boldsymbol{ww}'$. Then $\boldsymbol{wp}$ is a prefix of $\boldsymbol{vp}^k\boldsymbol{w}$, hence $\boldsymbol{wp}$ a prefix of $\boldsymbol{ww}'\boldsymbol{p}^k\boldsymbol{w}$, hence $\boldsymbol{p}$ is a prefix of $\boldsymbol{w}'\boldsymbol{p}^k\boldsymbol{w}$. If $\boldsymbol{w}'$ were a prefix of $\boldsymbol{p}$, it would contradict the primitiveness of $\boldsymbol{p}$, and so $\boldsymbol{p}$ must be a prefix of $\boldsymbol{w}'$. Thus $\boldsymbol{w}' = \boldsymbol{pw}''$ and $\boldsymbol{v} = \boldsymbol{ww}' = \boldsymbol{wpw}''$. It follows that $\boldsymbol{q} = (\boldsymbol{vp}^k)^t\boldsymbol{w} = \boldsymbol{q} = (\boldsymbol{wpw}''\boldsymbol{p}^k)^t\boldsymbol{w}$, contradicting the fact that $\boldsymbol{q}$ is not $\boldsymbol{p}$-regular.

(c) $\boldsymbol{w} = \boldsymbol{v}$ makes $\boldsymbol{q} = (\boldsymbol{w}\boldsymbol{p}^k)^t\boldsymbol{w}$ contradicting the fact that $\boldsymbol{q}$ is not $\boldsymbol{p}$-regular.

(d) $\boldsymbol{w} = \boldsymbol{v}\boldsymbol{p}^r\boldsymbol{p}_1$ for $0 \le r < k$, $\boldsymbol{p}_1$ a prefix of $\boldsymbol{p}$. If $\boldsymbol{p}_1 = \varepsilon$, then $\boldsymbol{p}$ is a suffix of $\boldsymbol{q}$, a contradiction. Thus $\boldsymbol{p}_1 \ne \varepsilon$. Thus $\boldsymbol{v}\boldsymbol{p}^r\boldsymbol{p}_1\boldsymbol{p}$ is a prefix of $\boldsymbol{v}\boldsymbol{p}^k\boldsymbol{w}$. Since $r < k$, we have $\boldsymbol{p}$ a prefix of $\boldsymbol{p}_1\boldsymbol{p}$, contradicting the primitiveness of $\boldsymbol{p}$.

Our assumption leads to a contradiction, and so the first part of the lemma holds. As above, the second part holds since $|\boldsymbol{u}\boldsymbol{p}| < 2\lambda$. $\square$

Lemma 4.5 *Let $\boldsymbol{x}$, $\boldsymbol{y}$ be two-pattern strings of scope λ, $\sigma = [\boldsymbol{p}, \boldsymbol{q}, i, j]_\lambda$ an expansion, and $\boldsymbol{y} = \sigma(\boldsymbol{x})$. Let $\boldsymbol{u}$ be a non-empty binary string and let $\boldsymbol{u}\boldsymbol{p}^k\boldsymbol{q}$, $1 \le k$, be a suffix of a restrained configuration $\boldsymbol{p}^{k+1}\boldsymbol{q}$ or $\boldsymbol{q}\boldsymbol{p}^k\boldsymbol{q}$ of $\boldsymbol{y}$. Let $\boldsymbol{q}\boldsymbol{p}$ be a restrained configuration of $\boldsymbol{y}$. Then $\boldsymbol{u}\boldsymbol{p}^k\boldsymbol{q} \succ\!\prec \boldsymbol{q}\boldsymbol{p}$ and whether $\boldsymbol{u}\boldsymbol{p}^k\boldsymbol{q} \prec \boldsymbol{q}\boldsymbol{p}$ or $\boldsymbol{u}\boldsymbol{p}^k\boldsymbol{q} \succ \boldsymbol{q}\boldsymbol{p}$ can be determined in $\le 2\lambda$ steps.*

Proof If $\boldsymbol{u}$ is suffix of $\boldsymbol{p}$ and if $|\boldsymbol{u}| > |\boldsymbol{q}|$, then $|\boldsymbol{u}\boldsymbol{p}^k| > |\boldsymbol{q}\boldsymbol{p}|$. It follows that $\boldsymbol{u}\boldsymbol{p} \succ\!\prec \boldsymbol{q}\boldsymbol{p}$: otherwise $\boldsymbol{q}\boldsymbol{p}$ is a prefix of $\boldsymbol{u}\boldsymbol{p}$ making the last $\boldsymbol{p}$ of $\boldsymbol{q}\boldsymbol{p}$ intersect with the last $\boldsymbol{p}$ of $\boldsymbol{u}\boldsymbol{p}$, contradicting the primitiveness of $\boldsymbol{p}$.

If $\boldsymbol{u}$ is suffix of $\boldsymbol{p}$ and if $|\boldsymbol{u}| = |\boldsymbol{q}|$, then $|\boldsymbol{u}\boldsymbol{p}^k| = |\boldsymbol{q}\boldsymbol{p}|$. It follows that $\boldsymbol{u}\boldsymbol{p} \succ\!\prec \boldsymbol{q}\boldsymbol{p}$: otherwise $\boldsymbol{q}\boldsymbol{p} = \boldsymbol{u}\boldsymbol{p}$ making $\boldsymbol{u} = \boldsymbol{q}$ and so $\boldsymbol{q}$ is a suffix of $\boldsymbol{p}$, a contradiction.

Of course, since $\boldsymbol{u}\boldsymbol{p} \succ\!\prec \boldsymbol{q}\boldsymbol{p}$ implies that $\boldsymbol{u}\boldsymbol{p}^k\boldsymbol{q} \succ\!\prec \boldsymbol{q}\boldsymbol{p}$ and the proof of the first part of the statement is completed.

Thus either $\boldsymbol{u}$ is a suffix of $\boldsymbol{p}$ and $|\boldsymbol{u}| < |\boldsymbol{q}|$, or $\boldsymbol{u}$ is a suffix of $\boldsymbol{q}$ (and so $|\boldsymbol{u}| < |\boldsymbol{q}|$). Arguing by contradiction, we are assuming that $\boldsymbol{u}\boldsymbol{p}^k\boldsymbol{q} \succ\!\prec \boldsymbol{q}\boldsymbol{p}$ does not hold. It follows that $\boldsymbol{u}$ is a prefix of $\boldsymbol{q}$ as $|\boldsymbol{u}| < |\boldsymbol{q}|$.

1. if $|\boldsymbol{q}| < |\boldsymbol{u}| + |\boldsymbol{p}|^r$, $1 \le r \le k$, then $\boldsymbol{q}\boldsymbol{p}$ and the r-th copy of $\boldsymbol{p}$ in $\boldsymbol{u}\boldsymbol{p}^k\boldsymbol{q}$ have a non-empty intersection, contradicting the fact that $\boldsymbol{p}$ is primitive.

2. if $|\boldsymbol{q}| = |\boldsymbol{u}| + |\boldsymbol{p}|^r$, $1 \le r \le k$, then $\boldsymbol{p}$ is a suffix of $\boldsymbol{q}$, a contradiction.

3. Thus $|\boldsymbol{q}| > |\boldsymbol{p}|^k$ and so $\boldsymbol{q} = (\boldsymbol{u}\boldsymbol{p}^k)^t\boldsymbol{v}$ for some $t \ge 1$, $\boldsymbol{v}$ a prefix of $\boldsymbol{v}\boldsymbol{u}^k$.

   (a) $\boldsymbol{v} = \varepsilon$. Then $\boldsymbol{p}$ is a suffix of $\boldsymbol{q}$, a contradiction.

(b) $\boldsymbol{v}$ is a proper prefix of $\boldsymbol{u}$. Then $\boldsymbol{u} = \boldsymbol{v}\boldsymbol{v}'$. Then $\boldsymbol{v}\boldsymbol{p}$ is a prefix of $\boldsymbol{u}\boldsymbol{p}^k\boldsymbol{v}$, hence $\boldsymbol{v}\boldsymbol{p}$ is a prefix of $\boldsymbol{v}\boldsymbol{v}'\boldsymbol{p}^k\boldsymbol{v}$. If $\boldsymbol{v}'$ were a prefix of $\boldsymbol{p}$, it would contradict the primitiveness of $\boldsymbol{p}$, and so $\boldsymbol{p}$ must be a prefix of $\boldsymbol{v}'$. Thus $\boldsymbol{v}' = \boldsymbol{p}\boldsymbol{v}''$ and $\boldsymbol{u} = \boldsymbol{v}\boldsymbol{v}' = \boldsymbol{v}\boldsymbol{p}\boldsymbol{v}''$. It follows that $\boldsymbol{q} = (\boldsymbol{u}\boldsymbol{p}^k)^t\boldsymbol{v} = \boldsymbol{q} = (\boldsymbol{v}\boldsymbol{p}\boldsymbol{v}''\boldsymbol{p}^k)^t\boldsymbol{v}$, contradicting the fact that $\boldsymbol{q}$ is not $\boldsymbol{p}$-regular.

(c) $\boldsymbol{v} = \boldsymbol{u}$ makes $\boldsymbol{q} = (\boldsymbol{v}\boldsymbol{p}^k)^t\boldsymbol{v}$ contradicting the fact that $\boldsymbol{q}$ is not $\boldsymbol{p}$-regular.

(d) $\boldsymbol{v} = \boldsymbol{u}\boldsymbol{p}^r\boldsymbol{p}_1$ for $0 \le r < k$, $\boldsymbol{p}_1$ a prefix of $\boldsymbol{p}$. If $\boldsymbol{p}_1 = \varepsilon$, then $\boldsymbol{p}$ is a suffix of $\boldsymbol{q}$, a contradiction. Thus $\boldsymbol{p}_1 \ne \varepsilon$. Thus $\boldsymbol{u}\boldsymbol{p}^r\boldsymbol{p}_1\boldsymbol{p}$ is a prefix of $\boldsymbol{u}\boldsymbol{p}^k\boldsymbol{u}\boldsymbol{p}^r\boldsymbol{p}_1\boldsymbol{p}$, and so $\boldsymbol{p}_1\boldsymbol{p}$ is a prefix of $\boldsymbol{p}^{k-r}\boldsymbol{u}\boldsymbol{p}^r\boldsymbol{p}_1$. Since $r < k$, $k-r \ge 1$ and so we have $\boldsymbol{p}$ prefix of $\boldsymbol{p}_1\boldsymbol{p}$, contradicting the primitiveness of $\boldsymbol{p}$.

Our assumption leads to a contradiction, and so the first part of the lemma holds. As above, the second part holds since $|\boldsymbol{u}\boldsymbol{p}| < 2\lambda$. $\square$

References

[F97] M. Farach, **Optimal suffix tree construction with large alphabets**, in *Proc. 38th Annual Symposium on Foundations of Computer Science*, IEEE (1997) pp. 137–143.

[FLS03] F. Franek, W. Lu, and W. F. Smyth, **Two-pattern strings I — a recognition algorithm**, *J. Discrete Algorithms 1–5/6* (2003) pp. 445–460.

[FLS04] F. Franek, W. Lu, and W. F. Smyth, **Two-pattern strings II — computing all repetitions and near-repetitions**, submitted to *J. Discrete Algorithms*.

[KA03] P. Ko and S. Aluru, **Space efficient linear time construction of suffix arrays**, *Proceedings of the 14th Annual Symposium CPM*, LNCS 2676, Springer (2003) pp. 200–210.

[KSPP03] D. K. Kim, J. S. Sim, H. Park, and K. Park, **Linear-time construction of suffix arrays**, *Proceedings of the 14th Annual Symposium CPM*, LNCS 2676, Springer (2003) pp. 186–199.

[KS03] J. Kärkkäinen and P. Sanders, **Simple linear work suffix array construction**, *Proceedings of the 30th International Colloquium on Automata, Languages and Programming*, LNCS 2719, Springer (2003) pp. 943–955.

[MM93] U. Manber and G. Myers, **Suffix arrays: a new method for on-line string searches**, *SIAM Journal on Computing 22–5* (1993) pp. 935–948.

Acknowledgements

The first author would like to acknowledge the support and hospitality of the School of Computing, Curtin University, Perth, Australia during the research for this paper. The research of both authors was supported in part by their respective research grants from the Natural Sciences and Engineering Research Council of Canada.