Universal Reconstruction of a String

Paweł Gawrychowski[1,⋆], Tomasz Kociumaka[1,⋆⋆,⋆⋆⋆], Jakub Radoszewski[1,†,⋆⋆⋆], Wojciech Rytter[1,2,⋆⋆⋆], and Tomasz Waleń[1,⋆⋆⋆]

[1] Faculty of Mathematics, Informatics and Mechanics, University of Warsaw, Warsaw, Poland
[gawry,kociumaka,jrad,rytter,walen]@mimuw.edu.pl
[2] Faculty of Mathematics and Computer Science, Copernicus University, Toruń, Poland

Abstract. Many properties of a string can be viewed as sets of dependencies between substrings of the string expressed in terms of substring equality. We design a linear-time algorithm which finds a solution to an arbitrary system of such constraints: a generic string satisfying a system of substring equations. This provides a general tool for reconstructing a string from different kinds of repetitions or symmetries present in the string, in particular, from runs or from maximal palindromes. The recursive structure of our algorithm in some aspects resembles the suffix array construction by Kärkkäinen and Sanders (J. ACM, 2006).

1 Introduction

Let s be a string of length n, $s = s_0 \dots s_{n-1}$. For $0 \le p \le q < n$, we denote a substring $s_p \dots s_q$ by $s[p..q]$. A *substring equation* is a constraint of the form "$s[p..q] = s[p'..q']$". We say that a string s of length n is a *solution* to a system of substring equations E (*satisfies* E) if it satisfies each equation of the system.

Clearly, every system has a solution which is a string over a unary alphabet, i.e., a^n. Thus, we focus on *generic* solutions containing the largest number of different characters. The important feature of any such solution is that it can be used to describe all solutions of a system of substring equations:

Observation 1. *If s is a generic solution of length n to a system E, then for each string s' of length n that satisfies E there exists a letter-to-letter morphism (a coding) μ such that $\mu(s) = s'$.*

⋆ Work done while the author held a post-doctoral position at Warsaw Center of Mathematics and Computer Science.
⋆⋆ Supported by Polish budget funds for science in 2013-2017 as a research project under the 'Diamond Grant' program.
⋆⋆⋆ Supported by the grant NCN2014/13/B/ST6/00770 of the Polish Science Center.
† Supported by the Polish Ministry of Science and Higher Education under the 'Iuventus Plus' program in 2015-2016 grant no 0392/IP3/2015/73. The author also receives financial support of Foundation for Polish Science.

In particular, every two generic solutions of the same length are equivalent up to renaming letters. We denote one of the generic solutions by $\Phi(E)$.

Our main result. We design a linear-time algorithm which computes a generic solution $\Phi(E)$ for a system E of substring equations.

The fact that our solution is generic lets us solve in $\mathcal{O}(n)$ time many classic string recovery problems.

Algorithms recovering (reverse engineering, inferring) a string from many internal structures are known. This includes recovery from border array [13,11,12], strong border array [15], prefix array [7], the set of maximal palindromes [17], minimum and maximum cover array [9,26], Lyndon factorization [27], suffix array [3], directed acyclic word graph (DAWG) [3], suffix tree [19,5,28], and parameterized border array [18]. For all but the last one of the aforementioned problems, there are algorithms running in linear time and constructing a string over the smallest possible alphabet. For parameterized border array reconstruction, the fastest algorithm works in $\mathcal{O}(n^{1.5})$ time. A more difficult task is reconstruction of a string from the set of all runs. In [25] an $\mathcal{O}(n^2)$-time algorithm for this problem is presented and, moreover, it is shown that recovering a string over the smallest alphabet is NP-hard. Another hard problem is string reconstruction from the longest previous factor (LPF) array, which is NP-complete even without restrictions on the alphabet size [16]. Recovery problems have also been investigated for indeterminate strings [1,4].

A solution to our general problem provides a single tool for several existing recovery problems. This mainly includes problems which can be expressed as finding a string satisfying a conjunction of certain explicit substring equality constraints and implicit substring inequality constraints.

Our further results. We obtain linear-time algorithms for inferring a string from its border array, prefix array, set of maximal palindromes or set of runs. In particular, we improve the quadratic reconstruction algorithm from runs of [25]. In all cases the algorithms compute a generic solution.

Overview of the paper. In Section 2, we present a naive algorithm and name certain properties of generic solutions $\Phi(E)$. Also, we provide a way to remove all redundant equations in the system. In Section 3, we present a simple $\mathcal{O}(|E| + n \log n)$-time algorithm, which is based on the doubling technique. It shares some features with the construction algorithm of the KMR identifiers [21], but it processes equations in decreasing lengths, as opposed to the increasing order in KMR. Then, in Section 4, we design a linear-time solution. It is a recursive algorithm which to some extent resembles the suffix array construction algorithm by Kärkkäinen & Sanders [20]. In order to achieve $\mathcal{O}(n)$ running time, in this version of the algorithm we apply the maximum spanning tree construction algorithm by Fredman and Willard [14], which is not feasible in practice. To overcome this issue, in Section 5 we change some details of the algorithm, so that no heavy word-RAM machinery is required. We conclude with Section 6, where we present applications to several string recovery problems.

2 Basic observations

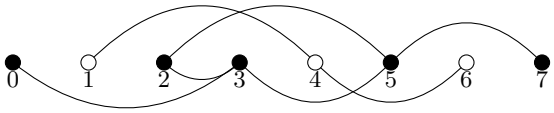
A naive approach to finding a generic n-character solution to a system of equations E is to transform substring equations into equations of individual letters. If "$s[p..q] = s[p'..q']$" belongs to E, then for every $i \in \{0, \ldots, q-p\}$, the equation on letters $s_{p+i} = s_{p'+i}$ must be satisfied. The latter can be represented as edges in a *positions graph* whose vertices $\{0, \ldots, n-1\}$ correspond to positions in s. This graph lets us easily characterize generic solutions. The idea of using a graph to represent constraints on letters already appeared in the context of prefix array reconstruction for indeterminate strings [6,4].

Observation 2. *For $s = \Phi(E)$, we have $s[i] = s[j]$ if and only if i, j are in the same connected component of the positions graph.*

Example 3. If the equations are:

$$E: \quad s[0..2] = s[3..5], \quad s[2..2] = s[3..3], \quad s[3..5] = s[5..7],$$

then we obtain the following positions graph:



Its connected components are $\{0, 2, 3, 5, 7\}$ and $\{1, 4, 6\}$. The generic string that satisfies E is a string $\Phi(E) =$ `abaababa`.

The approach described above in general works in $\mathcal{O}(|E|n)$ time, where $|E|$ is the number of equations in E. However, it is much more efficient for short equations.

Observation 4. *If all equations of E are of length 1, then the size of the positions graph is $\mathcal{O}(|E| + n)$. Consequently, $\Phi(E)$ can be computed in linear time.*

We call two systems E and E' *equivalent*, denoted $E \equiv E'$, if both have exactly the same solutions, i.e., $\Phi(E) = \Phi(E')$. Observe that this relation is hereditary in some sense: if $E \equiv E'$, then $E \cup F \equiv E' \cup F$ for any system F.

In the remainder of the paper, we represent each equation "$s[p..q] = s[p'..q']$" as a triple $(p, p', q-p+1)$. We refer to p, p' as the *starting positions* and to $q-p+1$ as the *length* of the equation.

Our algorithms follow the lines of Observation 4, converting the input system E to an equivalent system composed of equations of length 1. As opposed to the naive algorithm, we control the number of the equations. This is achieved using two kinds of transformations, which we refer to as `Split` and `Reduce`.

A `Split` operation transforms a single equation into an equivalent system of shorter equations. More formally, to split an equation (i, j, ℓ), we choose a collection $\mathcal{I}$ of integer intervals $\{b, \ldots, e-1\}$ such that $\bigcup \mathcal{I} = \{0, \ldots, \ell-1\}$ and

replace $\{(i,j,\ell)\}$ with $\{(i+b, j+b, e-b) : \{b,\dots,e-1\} \in \mathcal{I}\}$. It is easy to verify that this indeed produces an equivalent system of equations.

While a `Split` transformation lets us decrease the lengths of equations, it increases the number of equations. To control the latter, we apply a `Reduce` operation, which finds $E' \subseteq E$ such that $E' \equiv E$. Such an operation can be also seen as a sequence of removals of a *redundant* equation: we find an equation $(i,j,\ell) \in E$ such that $E \setminus \{(i,j,\ell)\} \equiv E$, and remove it from E.

For a system of equations E represented as triples, we define its *equations graph* $G(E)$ as a weighted graph $(V(E), E)$, where $V(E) = \bigcup_{(i,j,\ell)\in E}\{i,j\}$, and triple (i,j,ℓ) represents edge (i,j) with weight ℓ. Note that the equations graph coincides with the positions graph for every system of equations of length 1. The equations graph lets us conveniently describe some properties of the system E using notions of graph theory. We say that a system of equations E is *acyclic* if the underlying graph $G(E)$ is acyclic, i.e., if $G(E)$ is a forest. For a weighted graph G, by $\mathrm{MST}(G)$ we denote the edge-set of a maximum-weight spanning forest of G.

Lemma 5. *Let $F = \mathrm{MST}(G(E))$ be a maximum-weight spanning forest of $G(E)$. Then $F \equiv E$.*

Proof. It is well-known that a maximum spanning forest can be constructed by iteratively removing the lightest edge on a cycle (this is the so-called *red rule*, upon which Kruskal's algorithm is based). Suppose that $G(E)$ contains a cycle C. By removing the lightest edge of C, denoted (i,j,ℓ), we obtain a set of edges E'.

Note that $C' = C \setminus \{(i,j,\ell)\}$ is a sequence of edges $(i,i_1,\ell_1),\dots,(i_{k-1},j,\ell_k)$ such that $\ell \le \min(\ell_1,\dots,\ell_k)$. By transitivity, the equation (i,j,ℓ) is implied by the equations from C'. Hence, $C \equiv C'$, and consequently, $E \equiv E'$.

Applying this argument inductively, we obtain that $\mathrm{MST}(G(E)) \equiv E$. □

3 $\mathcal{O}(|E| + n \log n)$-time algorithm

We start with an $\mathcal{O}(|E| + n\log n)$-time algorithm, which uses simple split and reduction rules. We say that a system of equations E is *p-uniform* if all equations in E have length p. For uniform systems, it is easy to design an efficient implementation of the reduction rule obtained through Lemma 5. We denote the underlying procedure as `UniformReduce`(E).

Lemma 6. *Given a p-uniform system E, an equivalent acyclic subsystem $F \subseteq E$ can be constructed in $\mathcal{O}(|E|)$ time.*

Proof. By Lemma 5, it suffices to take $F = \mathrm{MST}(G(E))$. For a uniform system of equations, $G(E)$ has uniform weights, so any spanning forest is maximal. Such a forest can be constructed using a textbook graph search algorithm. □

A complementary split rule `UniformSplit`(E,p) is used to transform each equation (i,j,ℓ) from a system E into a pair of equations of a specified length p:

$$\{(i,j,\ell)\} \equiv \{(i,j,p), (i+\ell-p, j+\ell-p, p)\}.$$

It is applicable whenever all equations satisfy $p \leq \ell \leq 2p$.

The pseudocode of an algorithm using these two rules is provided below. In each step, it processes a system of equations of length between 2^k and $2^{k+1}-1$, which consists both of equations obtained from the preceding step and equations from the input system. First, these equations are transformed into a 2^k-uniform system using the `UniformSplit` operation. Then the resulting system is reduced into an acyclic system using Lemma 6.

Algorithm 1: $\mathcal{O}(|E| + n \log n)$-time solution

Input: A system of equations E
Output: A generic solution $\Phi(E)$

$F_{\lfloor \log n \rfloor + 1} := \emptyset$
for $k := \lfloor \log n \rfloor$ **downto** 0 **do**
    $E_k := \{(i, j, \ell) \in E : 2^k \leq \ell < 2^{k+1}\}$
    $F_k := \texttt{UniformSplit}(F_{k+1} \cup E_k,\ 2^k)$ {now F_k is 2^k-uniform}
    $F_k := \texttt{UniformReduce}(F_k,\ 2^k)$ {using Lemma 6}
return $\Phi(F_0)$ {using Observation 4}

Proposition 7. *Let E be a system of m equations over n positions. Algorithm 1 computes the universal solution $\Phi(E)$ of E in $\mathcal{O}(m + n \log n)$ time.*

Proof. The iteration of the **for**-loop indexed with k runs in $\mathcal{O}(|E_k| + |F_k|)$ time. Note that $\sum_k |E_k| = m$ and $|F_k| < n$, since each F_k is acyclic. □

4 Linear-time algorithm

There are two main ideas behind the improvement from $\mathcal{O}(n \log n)$ to $\mathcal{O}(n)$ in the running time of the algorithm. First, we apply advanced machinery to efficiently implement for an arbitrary system the reduction rule following from Lemma 5. The other idea relies on a novel application of splitting. Previously, we used it only to manipulate the lengths of equations: to make them smaller and uniform. Now, we also apply split operations to restrict the starting positions of long equations. This is useful since the $|E| < n$ bound on the size of an acyclic system is actually $|E| < |V(E)|$. Thus, we introduce *special* positions.

4.1 Properties of k-special integers

Definition 8. *We say that a non-negative integer i is* k-special *if none of the k least significant digits of the quaternary representation of i is zero, i.e., if the suffix of length k of $(i)_4$ does not contain a zero.*

We denote the set of k-special integers by $\mathcal{S}_k$.

Observation 9. *Let i, j be non-negative integers such that $i \equiv j \pmod{4^k}$. Then $i \in \mathcal{S}_k$ if and only if $j \in \mathcal{S}_k$.*

Example 10. An integer i is 2-special unless its remainder modulo 16 is one of the following: $\{(00)_4, (01)_4, (02)_4, (03)_4, (10)_4, (20)_4, (30)_4\}$. We have $\mathcal{S}_2 = \{5, 6, 7, 9, 10, 11, 13, 14, 15, 21, 22, 23, 25, 26, 27, 29, 30, 31, \ldots\}$.

Fact 11. ***(a)*** *For every positive integer n we have $|\mathcal{S}_k \cap \{0, \ldots, n-1\}| \leq \left(\frac{3}{4}\right)^k n$.*
(b) *If $i, j \in \mathcal{S}_k$, then there exists an integer $r \in \{0, 1, 2\}$ such that $i + r4^k, j + r4^k \in \mathcal{S}_{k+1}$. Moreover, such r can be found in constant time.*

Proof. (a) Observe that $|\mathcal{S}_k \cap \{0, \ldots, 4^k - 1\}| = 3^k$. Hence, due to Observation 9, the claim is valid whenever n is a multiple of 4^k. The technical generalization of the proof for arbitrary n is omitted in this version.

(b) Let c and d be the $(k+1)$-th least significant digits of $(i)_4$ and $(j)_4$, respectively. If both c and d are non-zero, we take $r = 0$. If both are equal to 0, we take $r = 1$. Otherwise, exactly one of c, d is 0. If the other is equal to 1 or 2, we choose $r = 1$. In the remaining cases, we take $r = 2$. □

4.2 Split rules

An equation (i, j, ℓ) is called *k-special* if the positions i, j are both k-special integers. We say that an equation is *r-short* if its length does not exceed r.

Lemma 12. *Every k-special equation can be split in $\mathcal{O}(1)$ time into a constant number of k-special equations each of which is $(k+1)$-special or 4^{k+1}-short.*

Proof. If the input equation (i, j, ℓ) is already 4^{k+1}-short, there is nothing to do. Otherwise, we apply Fact 11(b) to find $r \in \{0, 1, 2\}$ such that $(i + r4^k, j + r4^k, \ell - r4^k)$ is $(k+1)$-special, and we use it in the decomposition along with $(i, j, 4^{k+1})$. □

By `SpecialSplit`(E, k) we denote a procedure which applies the lemma above for every equation in E and returns a pair of systems (E_1, E_2) such that equations in E_1 are 4^{k+1}-short, equations in E_2 are $(k+1)$-special, and $E \equiv E_1 \cup E_2$.

Lemma 13. *Every 4^{k+1}-short k-special equation can be split in $\mathcal{O}(1)$ time into a constant number of 4^k-short k-special equations.*

Proof. If the input equation (i, j, ℓ) is already 4^k-short, there is nothing to do. Otherwise, we split it into at most four 4^k-short k-special equations. We replace the input equation with $(i, j, 4^k)$ and $(i + 4^k, j + 4^k, \ell - 4^k)$. By Observation 9, the latter is also k-special, and we might need to further split it into shorter equations. This step needs to be performed at most 4 times since $\ell \leq 4^{k+1}$. □

By `SimpleSplit`(E, k) we denote a procedure which applies the lemma above for every equation in E and returns a system E' equivalent with E.

4.3 Reduction rule

The general reduction procedure `Reduce`(E) is based on implementing Lemma 5 using a celebrated result by Fredman and Willard:

Theorem 14 ([14]). *In the standard word-RAM model of computation with word size $w = \Omega(\log n)$, maximum-weight spanning forest of a graph with w-bit integer weights can be computed in linear time.*

Corollary 15. *After $\mathcal{O}(n)$-time preprocessing, given an arbitrary equation system E, an equivalent acyclic system $F \subseteq E$ can be constructed in $\mathcal{O}(|E|)$ time.*

Proof. We follow the approach given by Lemma 5: we build the equations graph $G(E)$ and apply Theorem 14 to compute its maximum-weight spanning forest $F \subseteq E$. To avoid renaming vertices, we store them in an array of size n. During the preprocessing phase, we initialize this representation to store an empty graph, and then add an edge for each equation in E. Once we are done computing $\mathrm{MST}(G(E))$, we clean up iterating through E once again. □

4.4 Algorithm

In this section we apply the split and reduction rules developed above to obtain a linear-time algorithm. The main idea is to use `SpecialSplit` for subsequent values $k = 0, 1, \ldots$ in order to further and further restrict the starting positions of long equations. This is interleaved with reductions which bound the number of such equations to $|\mathcal{S}_k \cap \{0, \ldots, n-1\}| \leq (\frac{3}{4})^k n$, which is $\mathcal{O}(n)$ in total. For large enough k there are no k-special equations. In the end we are left with the remaining products of `SpecialSplit`, i.e., for every k with $\mathcal{O}((\frac{3}{4})^k n)$ equations which are both 4^{k+1}-short and k-special. They are processed in the order of decreasing k using alternating calls of `SimpleSplit` and `Reduce`, similarly as in Algorithm 1. The following recursive procedure implements this approach.

Procedure Shorten(E, k)

Input: An acyclic system E of k-special equations
Output: An equivalent acyclic system of k-special 4^k-short equations

if $E = \emptyset$ **then return** $\emptyset$
$(E_1, E_2) :=$ `SpecialSplit`$(E,\ k)$ {using Lemma 12}
$F :=$ `Shorten`(`Reduce`$(E_2),\ k+1)$ {recursive call, Corollary 15}
$F' :=$ `SimpleSplit`$(E_1 \cup F,\ k)$ {using Lemma 13}
return `Reduce`(F') {using Corollary 15}

Let us analyze its running time. By Fact 11(a), the size of any acyclic system of k-special equations does not exceed $(\frac{3}{4})^k n$. Consequently, we have $|E| \leq (\frac{3}{4})^k n$ and $|F| \leq (\frac{3}{4})^{k+1} n$. Lemmas 12 and 13 imply that split operations work in

$\mathcal{O}((\frac{3}{4})^k n)$ time and, in particular, return systems of this size. Thus, by Corollary 15, the reduction also works in $\mathcal{O}((\frac{3}{4})^k n)$ time. Consequently, the total time to compute $\texttt{Shorten}(E, 0)$ is $\mathcal{O}(\sum_{k\geq 0}(\frac{3}{4})^k n) = \mathcal{O}(n)$.

Before we apply the `Shorten` procedure, we need to make sure the input system is acyclic and consists of 0-special equations. The latter condition is void, so we just perform a reduction.

Algorithm 2: MAIN. An $\mathcal{O}(|E| + n)$-time solution

Input: A system of equations E
Output: A generic solution $\Phi(E)$

$E' := \texttt{Reduce}(E)$
$E'' := \texttt{Shorten}(E',\ 0)$ $\{E''$ is 1-short and $E'' \equiv E\}$
return $\Phi(E'')$ {using Observation 4}

This way, we complete the proof of our main result.

Theorem 16. *Let E be a system of m equations between substrings of a string of length n. There exists an $\mathcal{O}(n+m)$-time algorithm that finds the generic string $\Phi(E)$ that satisfies E.*

5 Practical implementation

The disadvantage of the algorithm presented in the previous section is that it uses the algorithm of Fredman and Willard (Theorem 14), which is very efficient in asymptotic terms but complicated and thus impractical. However, without much effort we are able to restrict the weights to powers of 2 not exceeding n. In this case, the MST can be computed using a simple solution based on the the Dijkstra-Jarník-Prim algorithm, which uses a priority queue as the underlying data structure (rather than union-find in Kruskal's algorithm). We maintain a single instance of the queue designed to efficiently handle a small universe.

Lemma 17. *In the standard word-RAM model of computation with word size $w = \Omega(\log N)$ (where N is a power of 2), one can implement a priority queue for keys within $\{2^0, 2^1, \ldots, 2^{\log N - 1}\}$ supporting standard operations (insert, find maximum, delete maximum) in $\mathcal{O}(1)$ time after $\mathcal{O}(N)$-time initialization.*

Proof sketch. The idea behind the priority queue is to store an integer whose bits represent keys present in the queue. Elements in the queue are stored in lists with each list responsible for a single key. To implement this queue, we still require word-RAM model, but we just use two standard bit operations. First, given a non-negative integer $x < N$, we want to locate position of the highest bit set to **1** in its binary representation, denoted $\mathrm{msb}(x)$. Second, given $x < N$ and $k < \log N$, we want to flip the k-th bit of x. Details are left for the full version.

Corollary 18. *After $\mathcal{O}(n)$-time preprocessing, given an arbitrary system E of equations whose lengths are powers of two, an equivalent acyclic subsystem $F \subseteq E$ can be constructed in $\mathcal{O}(|E|)$ time.*

5.1 Adjusted algorithm

We modify the definition of a k-special equation as follows. An equation (i, j, ℓ) is *strongly k-special* if $i, j \in \mathcal{S}_k$ and additionally $\ell = 2^p$ for some integer $p \geq 2k$. In particular, we may use Corollary 18 for any system of strongly special equations. The lower bound $\ell \geq 4^k$ is useful to implement the split operations, both adjusted below for strongly k-special equations. Due to space constraints, we omit the proof of Lemma 19.

Lemma 19. *Every strongly k-special equation can be split in $\mathcal{O}(1)$ time into a constant number of strongly k-special equations each of which is strongly $(k+1)$-special or 4^{k+1}-short.*

Lemma 20. *Every strongly k-special equation of length up to 4^{k+1} can be split in $\mathcal{O}(1)$ time into a constant number of 4^k-short strongly k-special equations.*

Proof. It suffices to split each equation equally into equations of length 4^k. □

Apart from slightly different implementation of subroutines, the procedure `Shorten` remains unchanged. However, before we apply it for $k = 0$, we need to take into account that while every equation is 0-special, it does not need to by strongly 0-special. However, it is easy to split any equation into two strongly 0-special ones. This is exactly the `UniformSplit` operation of Section 3.

6 Applications

In this section we apply Theorem 16 for several string recovery problems. In this class of problems, we are supposed to find an example string of a given length n which satisfies certain properties, or state that no such string exists. We assume an unbounded alphabet $\Sigma = \mathbb{N}$. In each of the problems, the property is expressible as a system of equations on substrings $E_=$ and a system of inequalities (more precisely, non-equalities) of substrings $E_\neq$ that the string should satisfy.

Lemma 21 provides a verification tool for the question whether there are strings consistent with $E_=$ and $E_\neq$. This is because either $\Phi(E_=)$ is valid or there are no solutions possible.

Lemma 21. *Let $E_=$ and $E_\neq$ be sets of equations and inequalities over n positions. If $\Phi(E_=)$ does not satisfy $E_\neq$ then there exists no string of length n that satisfies both $E_=$ and $E_\neq$. Moreover, one can check in $\mathcal{O}(n+|E_\neq|)$ time if $\Phi(E_=)$ satisfies all inequalities $E_\neq$.*

Proof. By Observation 1, any string s satisfying $E_=$ must be an image of $t = \Phi(E_=)$ through a letter-to-letter morphism. Therefore, every substring inequality satisfied by s is also satisfied by t. To check if t satisfies $E_\neq$, one can use longest common extension queries (i.e., longest common prefix queries); see [8]. □

We start the presentation of applications with two simpler examples. The *prefix array* $\mathrm{PREF}[1..n-1]$ stores in $\mathrm{PREF}[i]$ the length of the longest common prefix of s and $s[i..n-1]$.

Lemma 22. *For every array $A[1..n-1]$ with values in $\{0,\ldots,n-1\}$ there exists a set of equations $E_=$ and a set of inequalities $E_\neq$ such that A is the prefix array of a string s if and only if s satisfies both $E_=$ and $E_\neq$. Moreover, $|E_=| \leq n$, $|E_\neq| \leq n$ and both sets can be constructed in $\mathcal{O}(n)$ time.*

Proof. We take the following equations: $E_= = \{s[0..A[i]-1] = s[i..i+A[i]-1] : i = 1,\ldots,n-1\}$ and inequalities: $E_\neq = \{s[A[i]] \neq s[i+A[i]] : i+A[i] < n; i = 1,\ldots,n-1\}$. □

We say that a string u is a *border* of a string v, if u occurs both as a proper prefix and as a proper suffix of v. The *border array* $B[1..n-1]$ is an integer array such that $B[i]$ is the length of the longest border of $s[0..i]$.

Lemma 23. *For every array $A[1..n-1]$ with values in $\{0,\ldots,n-1\}$ there exists a set of equations $E_=$ and a set of inequalities $E_\neq$ such that A is the border array of a string s if and only if s satisfies both $E_=$ and $E_\neq$. Moreover, $|E_=| \leq n$ and this set can be constructed in $\mathcal{O}(n)$ time.*

Proof. We take the following equations: $E_= = \{s[0..A[i]-1] = s[i-A[i]+1..i] : i = 1,\ldots,n-1\}$. The inequalities state in an analogous way that $s[0..i]$ have no border of length exceeding $A[i]$: $E_\neq = \{s[0..j-1] \neq s[i-j+1..i] : i = 1,\ldots,n-1;\ A[i] < j \leq i\}$. □

A *period* of a string v is such a positive integer $p \leq |v|$ that $v[i] = v[i+p]$ for $0 \leq i < |v|-p$. A *run* is a triple (i,j,p) such that p is the smallest period of $s[i..j]$, $|s[i..j]| \geq 2p$ and neither $s[i-1..j]$ nor $s[i..j+1]$ have period p (possibly because $i = 0$ or $j = |s|-1$). By $Runs(s)$ we denote the set of all runs in s. Every string of length n has less than n runs [2].

Lemma 24. *For a set R ($|R| < n$) of integer triples (i,j,p), $0 \leq i < j \leq n-1$, $p \in \{1,\ldots,\lfloor n/2 \rfloor\}$, there exists a set of equations $E_=$ and a set of inequalities $E_\neq$ such that R is the set of all runs of a string s of length n if and only if s satisfies both $E_=$ and $E_\neq$. Moreover, $|E_=| \leq n$ and this set can be constructed in $\mathcal{O}(n)$ time.*

Proof. The fact that R is a set of runs of a string s can be described using equations $E_= = \{s[i..j-p] = s[i+p..j] : (i,j,p) \in R\}$. Inequalities say that no run is extendible or has a smaller period, and that no other runs exist in s. □

A string v is called a *palindrome* if v is equal to its reverse $v^R = v_{|v|-1}\ldots v_1 v_0$. A substring $s[i..j]$ is called a *maximal palindrome* if it is a palindrome, but $s[i-1..j+1]$ is not a palindrome. The set of maximal palindromes in s, denoted as $MaxPal(s)$, determines the structure of all palindromic substrings of s. Due to space constraints we omit the proof of the following lemma.

Lemma 25. *For a set P ($|P| < 2n$) of integer pairs (i,j), $0 \leq i \leq j \leq n-1$, there exists a set of equations $E_=$ and a set of inequalities $E_\neq$ over $2n$ positions such that P is the set of maximal palindromes of a string t of length n if and only if there exists a string s of length $2n$ that satisfies both $E_=$ and $E_\neq$ (then $s = tt^R$). Moreover, $|E_=| \leq 3n$, $|E_\neq| = \mathcal{O}(n)$ and both sets can be constructed in $\mathcal{O}(n)$ time.*

Theorem 26. *In $\mathcal{O}(n)$ time one can recover a string of length n from its prefix array, its border array, its runs structure, or its maximal palindromes.*

Proof. First, we apply one of Lemmas 22–25 to generate in $\mathcal{O}(n)$ time a system of equations $E_=$ for the particular recovery problem. Using Theorem 16, we compute a generic string $s = \Phi(E_=)$. Lemma 21 guarantees that if a solution to the recovery problem exists, then s is such a solution. Finally, we use a linear-time algorithm to compute its prefix array/border array/runs/maximal palindromes (see [2,24,10]), and check if the result matches the input. Note that for the maximal palindromes recovery problem, this way we obtain $s = tt^R$, so we need to take the first half of $s = \Phi(E_=)$ as the final solution t.

In the case of prefix array and maximal palindromes we have $|E_{\neq}| = \mathcal{O}(n)$. Therefore in these cases we obtain a simpler algorithm to check if s is a solution using Lemma 21. Either approach gives $\mathcal{O}(n)$-time recovery. □

Our linear-time algorithm to recover a string from its runs improves upon an $\mathcal{O}(n^2)$-time algorithm by Matsubara et al. [25]. Finally, recovery from runs can be extended to gapped repeats and subrepetitions [23] with running time equal to the running time of the respective construction algorithms [23,22].

References

1. Alatabbi, A., Rahman, M.S., Smyth, W.: Inferring an indeterminate string from a prefix graph. J. of Discrete Algorithms (2014)
2. Bannai, H., I, T., Inenaga, S., Nakashima, Y., Takeda, M., Tsuruta, K.: The "runs" theorem. ArXiv e-prints 1406.0263v6 (2015)
3. Bannai, H., Inenaga, S., Shinohara, A., Takeda, M.: Inferring strings from graphs and arrays. In: Rovan, B., Vojtás, P. (eds.) Mathematical Foundations of Computer Science. LNCS, vol. 2747, pp. 208–217. Springer (2003)
4. Blanchet-Sadri, F., Bodnar, M., Winkle, B.D.: New bounds and extended relations between prefix arrays, border arrays, undirected graphs, and indeterminate strings. In: Mayr, E.W., Portier, N. (eds.) Symposium on Theoretical Aspects of Computer Science. LIPIcs, vol. 25, pp. 162–173. Schloss Dagstuhl–Leibniz-Zentrum für Informatik (2014)
5. Cazaux, B., Rivals, E.: Reverse engineering of compact suffix trees and links: A novel algorithm. J. Discrete Algorithms 28, 9–22 (2014)
6. Christodoulakis, M., Ryan, P.J., Smyth, W.F., Wang, S.: Indeterminate strings, prefix arrays & undirected graphs. ArXiv e-prints 1406.3289 (2014)
7. Clément, J., Crochemore, M., Rindone, G.: Reverse engineering prefix tables. In: Albers, S., Marion, J.Y. (eds.) Symposium on Theoretical Aspects of Computer Science. LIPIcs, vol. 3, pp. 289–300. Schloss Dagstuhl–Leibniz-Zentrum für Informatik (2009)
8. Crochemore, M., Hancart, C., Lecroq, T.: Algorithms on Strings. Cambridge University Press, Cambridge (2007)
9. Crochemore, M., Iliopoulos, C.S., Pissis, S.P., Tischler, G.: Cover array string reconstruction. In: Amir, A., Parida, L. (eds.) Combinatorial Pattern Matching. LNCS, vol. 6129, pp. 251–259. Springer (2010)
10. Crochemore, M., Rytter, W.: Jewels of Stringology. World Scientific (2003)

11. Duval, J., Lecroq, T., Lefebvre, A.: Border array on bounded alphabet. Journal of Automata, Languages and Combinatorics 10(1), 51–60 (2005)
12. Duval, J., Lecroq, T., Lefebvre, A.: Efficient validation and construction of border arrays and validation of string matching automata. RAIRO-Theor. Inf. Appl. 43(02), 281–297 (2009)
13. Franek, F., Gao, S., Lu, W., Ryan, P., Smyth, W., Sun, Y., Yang, L.: Verifying a border array in linear time. J. on Combinatorial Mathematics and Combinatorial Computing 42, 223–236 (2002)
14. 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)
15. Gawrychowski, P., Jeż, A., Jeż, L.: Validating the Knuth-Morris-Pratt failure function, fast and online. Theor. Comput. Syst. 54(2), 337–372 (2014)
16. He, J., Liang, H., Yang, G.: Reversing longest previous factor tables is hard. In: Dehne, F., Iacono, J., Sack, J. (eds.) Algorithms and Data Structures – WADS 2011. LNCS, vol. 6844, pp. 488–499. Springer (2011)
17. I, T., Inenaga, S., Bannai, H., Takeda, M.: Counting and verifying maximal palindromes. In: Chávez, E., Lonardi, S. (eds.) String Processing and Information Retrieval. LNCS, vol. 6393, pp. 135–146. Springer (2010)
18. I, T., Inenaga, S., Bannai, H., Takeda, M.: Verifying and enumerating parameterized border arrays. Theor. Comput. Sci. 412(50), 6959–6981 (2011)
19. I, T., Inenaga, S., Bannai, H., Takeda, M.: Inferring strings from suffix trees and links on a binary alphabet. Discrete Appl. Math. 163, 316–325 (2014)
20. Kärkkäinen, J., Sanders, P., Burkhardt, S.: Linear work suffix array construction. J. ACM 53(6), 918–936 (2006)
21. Karp, R.M., Miller, R.E., Rosenberg, A.L.: Rapid identification of repeated patterns in strings, trees and arrays. In: Fischer, P.C., Zeiger, H.P., Ullman, J.D., Rosenberg, A.L. (eds.) 4th Annual ACM Symposium on Theory of Computing. pp. 125–136. ACM (1972)
22. Kociumaka, T., Radoszewski, J., Rytter, W., Waleń, T.: Internal pattern matching queries in a text and applications. In: Indyk, P. (ed.) Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms. pp. 532–551. SIAM (2015)
23. Kolpakov, R., Podolskiy, M., Posypkin, M., Khrapov, N.: Searching of gapped repeats and subrepetitions in a word. In: Kulikov, A.S., Kuznetsov, S.O., Pevzner, P.A. (eds.) Combinatorial Pattern Matching. LNCS, vol. 8486, pp. 212–221. Springer (2014)
24. Kolpakov, R.M., Kucherov, G.: Finding maximal repetitions in a word in linear time. In: 40th Annual Symposium on Foundations of Computer Science. pp. 596–604. FOCS'99, IEEE Computer Society (1999)
25. Matsubara, W., Ishino, A., Shinohara, A.: Inferring strings from runs. In: Holub, J., Zdárek, J. (eds.) Prague Stringology Conference. pp. 150–160. Czech Technical University (2010)
26. Moosa, T.M., Nazeen, S., Rahman, M.S., Reaz, R.: Linear time inference of strings from cover arrays using a binary alphabet (Extended abstract). In: Rahman, M.S., Nakano, S. (eds.) WALCOM: Algorithms and Computation. LNCS, vol. 7157, pp. 160–172. Springer (2012)
27. Nakashima, Y., Okabe, T., I, T., Inenaga, S., Bannai, H., Takeda, M.: Inferring strings from Lyndon factorization. In: Csuhaj-Varjú, E., Dietzfelbinger, M., Ésik, Z. (eds.) Mathematical Foundations of Computer Science. LNCS, vol. 8635, pp. 565–576. Springer (2014)
28. Starikovskaya, T., Vildhøj, H.W.: A suffix tree or not a suffix tree? J. Discrete Algorithms 32, 14–23 (2015)