Degenerate String Reconstruction from Cover Arrays

(Extended Abstract)

Dipankar Ranjan Baisya, Mir Md. Faysal, and M. Sohel Rahman

AℓEDA Group
Department of Computer Science and Engineering (CSE)
Bangladesh University of Engineering and Technology (BUET)
Dhaka 1000
Bangladesh
`msrahman@cse.buet.ac.bd`

Abstract. Regularities in degenerate strings have recently been a matter of interest because of their use in the fields of molecular biology, musical text analysis, cryptanalysis and so on. In this paper, we study the problem of reconstructing a degenerate string from a cover array. We present two efficient algorithms to reconstruct a degenerate string from a valid cover array one using an unbounded alphabet and the other using minimum sized alphabet.

Keywords: degenerate strings, string reconstruction, algorithms

1 Introduction

A degenerate string (also referred to as an indeterminate string in the literature) is a generalization of a (regular) string, in which each position contains either a single character or a nonempty set of characters. The problems of degenerate pattern matching [9–11, 15] and finding regularities in degenerate strings [1, 2, 4, 8, 14] have been addressed with great enthusiasm over the last decade. Authors in [4] described the way of finding all covers of an indeterminate string in $O(n)$ time on average. Another interesting avenue for research is to explore the problem of inferring a string given some arbitrary data structure (e.g., array, tree etc.) related to some of these regularities. However, despite several results on regular string inference in the literature [5–7, 12] the problem of degenerate string inference is yet to be explored extensively. To the best of our knowledge the only work on this topic is the recent work of Nazeen et al. [13] where the authors presented string inference algorithms considering border arrays of degenerate strings. The authors in [13] mentioned that similar inference algorithms for cover arrays of degenerate strings could be worth-investigating as a future research topic. Inspired by the future research direction mentioned there, in this paper, we first present an algorithm for degenerate string reconstruction from an input cover array using an unbounded alphabet. Then we modify this algorithm such that it uses a least sized alphabet. Notably, the problem of inferring (regular) strings from cover arrays has already been tackled in [12].

The rest of this paper is organized as follows. Section 2 presents some definitions and notations. Section 3 discusses some important properties of a cover array and extends those in the context of degenerate strings. In Section 4 we describe the algorithms and related results. Finally, we briefly conclude in Section 5.

Dipankar Ranjan Baisya, Mir Md. Faysal, M. Sohel Rahman: Degenerate String Reconstruction from Cover Arrays, pp. 191–205.
Proceedings of PSC 2013, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-05330-0

2 Preliminaries

A string is a sequence of zero or more symbols from an alphabet Σ. The set of all strings over Σ is denoted by Σ^*. The *length* of a string X is denoted by $|X|$. The *empty string*, the string of length zero, is denoted by ϵ. The i-th symbol of a string X is denoted by $X[i]$. A string $W \in \Sigma^*$, is a *substring* of X if $X = UWV$, where $U, V \in \Sigma^*$. Conversely, X is called a *superstring* of W. We denote by $X[i..j]$ the substring of X that starts at position i and ends at position j. A string $W \in \Sigma$ is a *prefix* (*suffix*) of X if $X = WY$ ($X = YW$), for $Y \in \Sigma^*$. A string W is a *subsequence* of X (or X a *supersequence* of W) if W is obtained by deleting zero or more symbols at any positions from X. For example, *ace* is a subsequence of *abcabbcde*. For a given set S of strings, a string W is called a common subsequence of S if W is a subsequence of every string in S.

A string U is a *period* of X if X is a prefix of U^k for some positive integer k, or equivalently if X is a prefix of UX. The *period* of X is the shortest period of X. For example, if X = *abcabcab*, then *abc*, *abcabc* and the string X itself are periods of X, while *abc* is the *period* of X.

A degenerate string is a sequence $X = X[1]X[2]\cdots X[n]$, where $X[i] \subseteq \Sigma$ for all i, and Σ is a given alphabet of fixed size. A position of a degenerate string may match more than one elements from the alphabet Σ; such a position is said to have a *non-solid* symbol. If in a position we have only one element of Σ, then we refer to this position as *solid*. The definition of length for degenerate strings is the same as for regular strings: a degenerate string X has length n, when X has n positions, where each position can be either solid or non-solid. We represent non-solid positions using [..] and solid positions omitting [..]. The example in Table 1 identifies the solid and non-solid positions of a degenerate string.

Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
$X =$	a	a	[abc]	a	[ac]	b	c	a	a	[ac]	b	a	c	[abc]	a	[bc]

Table 1. An example of a degenerate string

Table 1 presents a degenerate string having non-solid symbols at Positions 3, 5, 10, 14 and 16. The rest of the positions contain solid symbols. Let λ_i, $|\lambda_i| \geq 2, 1 \leq i \leq s$, be pairwise distinct subsets of Σ. We form a new alphabet $\Sigma' = \Sigma \cup \lambda_1, \lambda_2, \ldots, \lambda_s$ and define a new relation *match* ($\approx$) on Σ' as follows:

Type 1. for every $\mu_1, \mu_2 \in \Sigma, \mu_1 \approx \mu_2$ if and only if $\mu_1 = \mu_2$;
Type 2. for every $\mu \in \Sigma$ and every $\lambda \in \Sigma' - \Sigma, \mu \approx \lambda$ if and only if $\mu \in \lambda$;
Type 3. for every $\lambda_i, \lambda_j \in \Sigma' - \Sigma, \lambda_i \approx \lambda_j$ if and only if $\lambda_i \cap \lambda_j \neq \emptyset$.

Observe that the relation match ($\approx$) is reflexive and symmetric but not necessarily transitive. For example, if $\lambda = [a, b]$, then we have $a \approx \lambda$ and $b \approx \lambda$. But clearly $a \not\approx b$.

From the example in Table 1, we have a Type 1 match between Positions 2 and 4, as both positions are solid and contain the letter a. Positions 3 and 6 give a match of Type 2 as the letter b is contained in the non-solid symbol $[abc]$. A match of Type 3 can be found between Positions 3 and 5, as the symbols at these two positions have a and c common. Although Positions 5 and 3 match and Positions 3 and 6 match, Positions 5 and 6 do not match, illustrating the non-transitivity of the matching operation for degenerate strings.

Cover is an interesting regularity in strings that in some sense generalizes the concept of quasiperiodicity [3]. We say that a string S covers a string U if every letter of U is contained in some occurrence of S as a substring of U. Then S is called a cover of U. Clearly, S must be a (proper) substring of to be a (proper) cover of U. Although a string can be considered to be a cover of itself, we follow the convention in the literature and consider only the proper covers. The *cover array* C of a regular string $X[1..n]$, is a data structure used to store the length of the longest proper cover of every prefix of X. So for all $i \in \{1..n\}$, $C[i]$ stores the length of the longest proper cover of $X[1..i]$ or 0. In fact, since every cover of any cover of X is also a cover of X, it turns out that, the cover array C compactly describes all the covers of every prefix of X. For every prefix $X[1..i]$ of X, the following sequence

$$C^1[i], C^2[i], \ldots, C^m[i] \tag{1}$$

is well defined and monotonically decreasing to $C^m[i] = 0$ for some $m \geq 1$ and this sequence identifies every cover of $X[1..i]$. Here, $C^k[i]$ is the length of the kth longest cover of $X[1..i]$, for $1 \leq k \leq m$.

Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
$X =$	a	b	a	a	b	a	b	a	a	b	a	a	b	a	b	a	a	b	a
$C =$	0	0	0	0	0	3	0	3	0	5	6	0	5	6	0	8	9	10	11

Table 2. Cover array of *abaababaabaababaaba*

From Table 2 we can see that, cover array of X, has the entries $C[19] = 11,\ C[11] = 6,\ C[6] = 3$ *and* $C[3] = 0$ representing the three covers of X having length $11,\ 6$ and 3 respectively.

The definition for cover is the same for both regular and degenerate strings. However, the definition of the cover array for a degenerate string changes. For a degenerate string, $C[i]$ stores the list of the lengths of the covers of $X[1..i]$. More elaborately, each $C[i]$ is a list $\langle C^p[i] \rangle$ such that $1 \leq p \leq |C[i]|$, where $C^p[i]$ denotes the pth largest cover of $X[1..i]$. As the matching operations of degenerate strings are not transitive, cover array algorithms for regular strings cannot be readily extended to degenerate strings.

Index	1	2	3	4	5	6
$X =$	a	b	a	[ab]	[ab]	a
$C =$	0	0	0	2	3	4
					2	3

Table 3. Cover array of *aba*[*ab*][*ab*]*a*

Also, Sequence 1, does not fully apply to the covers of degenerate strings. From Table 2 and Sequence 1, the degenerate string X should have two covers of length 4 and 2, as $C[6]$ contains 4 and $C[4]$ contains 2. However, as can be seen from Table 2, X has covers of length 4 and 3, but not of length 2. So for covers of degenerate strings Sequence 1 gives wrong information. Note that, the space requirement for representing the cover array of a degenerate string of length n is $O(n^2)$.

3 Basic Validation of a cover array

In this section, we discuss some basic properties of a valid cover array. The properties discussed here are mostly in the context of reconstruction of a (degenerate) string from a given cover array while we scan/proceed one position at a time from left to right. i.e., in an online fashion. Further validation properties will be discussed in the following section where we describe the algorithms. For $i \geq 2$, we say an integer j is a *candidate-length* (i.e., "candidate" to be the length) of a cover of $X[1..i]$, if $j \in \{1,\ldots,i-1\}$. Thus candidate-lengths of covers of $X[1..i]$ is $\pi_i = \{1,2,\ldots,i-1\}$. We say that an array $C[1..n]$ is a valid cover array if and only if it is the cover array of at least one degenerate string X with n positions (i.e., having length n). We also use the notion of an equivalence of strings based on their cover arrays as follows. We say that two strings $X_1[1..n]$ and $X_2[1..n]$ are *C-Equivalent* if and only if both of them have the same cover array $C[1..n]$. Given a degenerate string X of length n on alphabet Σ, we define $\Sigma_i \subseteq \Sigma$ to be the set of symbols used by the prefix $X[1..i]$. Further we say that a symbol $\psi \in \Sigma_i - \Sigma_{i-1}$ is *required by* $C[i]$.

Clearly, the only cover of $X[1]$ is necessarily an empty word. Thus we must have $C[1] = 0$, irrespective of any strings. Also, as has been discussed above, the list of lengths of the nonempty covers $C[i]$ of $X[1..i]$ is taken from π_i. We now present the following useful observation, argument and theorem.

Observation 1 *Suppose $C[1..n]$ is the cover array of a string $X[1..n]$. Then the following hold true.*

a. If $1 \in C[i]$, then $1 \in C[j]\ \forall\ 1 < j \leq i-1,\ i > 1$.
b. If $f \in C[i]$, and $f \in C[j]$ such that $j \geq i$ then $j - i \leq f$.
c. If $C[k] = 0$ for $1 \leq k \leq m$, then $C[m+1] \leq m-1$.

Lemma 1. *Suppose $C[1..n]$ is the cover array of a string $X[1..n]$. If $1 \in C[i]$, then $\{1,2,\ldots,i-1\} \subseteq C[i],\ i > 1$.*

Proof. Proof will be provided in the journal version. □

Lemma 2. *Suppose we have a cover array C. Suppose we have correctly reconstructed a degenerate string X_1 of length $i-1$ based on $C[1..i-1]$. Also assume that we have also correctly reconstructed X_2 of length i for $C[1..i]$. Further, suppose that $Z = \Sigma_{X_2} - \Sigma_{X_1}$. Then the following hold true:*

a. Suppose, $|Z| = 1$ and $Z = \{\psi\}$. Also, assume that ψ is required for $C^p[i]$. Then ψ can only be put at the following positions of $X_1 : \{C^p[i], 2 \times C^p[i], 3 \times C^p[i] \ldots\}$ to get X_2

b. Suppose $|Z| > 1$ and $Z = \{\psi_1,\ldots,\psi_k\}$. Also, assume that $\psi_j,\ 1 \leq j \leq k$ is required for $C^{p_j}[i]$

  i. Then $\psi_1,\psi_2,\ldots,\psi_k$ can only be put at the following positions of X_1: $\{C^{p_j}[i], 2\times C^{p_j}[i], 3 \times C^{p_j}[i],\ldots\}$

  ii. Assume that λ is the non-solid character containing all letters of Z. We use λ_k to denote the character containing $\psi_1,\ldots,\psi_k$. So, $\lambda_1 = \psi_1$ and hence is a solid character and $\lambda_k = [\psi_1..\psi_k]$. Further assume that $X_2^k = X_1'[1..i-1]\lambda_k$. We get $X_1'[1..i-1]$ by placing the new characters $\psi_i,\ \psi_i \in \lambda_k$ at the aforementioned specific Positions of X_1. Then all of $X_2^i[1..i-1],\ 1 \leq i \leq k$ along with X_1 are $C - Equivalent$.

c. X_1 and $X_2[1..i-1]$ are $C - Equivalent$.

Proof. Proof will be provided in the journal version. □

Now we are ready to present and prove the following important theorem.

i	1	2	3	4	5	6
$C[i]$	0	1	2	3	4	2
			1	2	3	
				1	2	
					1	

Table 4. An example of cover array of Degenerate string

i	1	2	3	4	5
$X[i]$	a	a	a	a	a

Table 5. Degenerate string of cover array up to Position 5

i	1	2	3	4	5	6
$X[i]$	a	a	a	a	a	b
		b		b		

Table 6. Degenerate string of cover array up to Position 6

Theorem 2 *Suppose $C[1..n]$ is an array of $n \geq 1$ lists of integers. If the following condition (Condition 1) is satisfied, then $C[1..n]$ is a cover array of some degenerate string $X[1..n]$.*

Condition 1

a. $C[1] \ = \ 0$
b. $C[i] \subseteq \{0\} \cup \pi_i, \ for\ 2 \leq i \leq n.$
  i. If $X[1..i]$ has the empty word for its only cover then we have $C[i] = \{0\}$
  ii. If $X[1..i]$ has nonempty covers then $C[i] = \{j | j \in \pi_i$ and $X[i] \approx X[j]\}$

Proof. Proof will be provided in the journal version. □

4 Our Algorithms

4.1 *CrAyDSRUn*

Our problem is to reconstruct a degenerate string of length n, given a valid cover array C. In this section, we focus on an unbounded alphabet and propose an algorithm called ***CrAyDSRUn*** (Cover Array Degenerate String Reconstruction from Unbounded Alphabet) for this problem. Given an array $C[1..n]$, ***CrAyDSRUn*** determines whether $C[1..n]$ is a valid cover array for at least one degenerate string and if so, it constructs one such degenerate string. Before presenting the algorithm, we first need to present some relevant definitions and notions.

Assume that, we have successfully reconstructed $X[1..i-1]$. We use ψ_i to denote the new set of characters introduced in $X[i]$, i.e., $\psi_i = \Sigma_i - \Sigma_{i-1}$. Now, we want to extend $X[1..i-1]$ to get $X[1..i]$ based won $C[1..i]$. Suppose that $a \in C[i]$. So, we must have a cover of length a for $X[1..i]$. Also if we need a new character, we have to place that it at Position i and other necessary positions of $X[1..i-1]$ (See Lemma 2). We denote by A'_i the set of symbols that are not allowed only at Position i, i.e., $A'_i = \bigcup_{j \in \pi_i - C[i]} X[j]$. On the other hand, we denote by A_i the set of symbols that can be assigned to $X[i]$. We now have the following lemma.

Lemma 3. *For every degenerate string $X[1..i]$ the following hold true:*

a. If $C^p[i] \neq 0$ for $1 \leq p \leq |C[i]|$ then

$$X[i] \approx X[C^p[i]],\ C^p[i] \in \pi_i \text{ and } A_i = \psi_i \cup \left(\bigcup_{1 \leq p \leq |C[i]|} (X[C^p[i]] - A'_i) \right)$$

b. if $C^p[i] = 0$ is the only entry of $C^p[i]$, then $C^p[i] \notin \pi_i$, $A_i = \psi_i$ and $|A_i| = |\psi_i| = 1$

Proof. *Proof will be provided in the journal version.* □

We note that our string inference algorithm follows a similar approach used in [13] to reconstruct degenerate strings from border arrays. The main differences lie in manipulating A_i, A'_i, validity checking of X and placing appropriate characters at appropriate positions. The steps of ***CrAyDSRUn*** are formally presented in Algorithm 1 (in Appendix). We assume that, we have an array α representing an unbounded alphabet from which we take the *basic* letters i.e., the non-degenerate letters from the alphabet Σ. The ***CrAyDSRUn*** algorithm takes an array $C[1..n]$ as input. It first checks the trivial validity condition whether $C[1] = 0$ or not; subsequently for every position $2 \leq i \leq n$, it checks whether $C^p[i] \in \pi_i$, $i \leq p \leq |C[i]|$. Algorithm ***CrAyDSRUn*** returns the input cover array as invalid as soon as it finds a violation of the conditions checked above. As long as the result of the above checking is positive, Algorithm ***CrAyDSRUn*** constructs A'_i and A_i for each position $2 \leq i \leq n$. To keep track of the alphabet size of each prefix $X[1..i]$, our algorithm uses an array k where $k[i] = |\Sigma_i|$.

To manipulate A'_i, we use function $getInvalidChar(Position,\ CoverValue)$ that takes two parameters. $Position$ refers to the position of the cover array and $CoverValue$ refers to one of the values of that position. To manipulate A_i, we use function $getProbableValidChar(Position,\ CoverValue)$. If a $CoverValue$ appears for the first time in C at Position i, then our algorithm will extend the string such that there is a cover of length equal to $CoverValue$ by putting the characters in $X[covervalue]$ at $X[i]$ provided that the positions of $(X[i-1]$ *and* $X[CoverValue-1])$, $(X[i-2]$ *and* $X[CoverValue-2]), \ldots,$ $(X[i-(CoverValue-1)]$ *and* $X[1])$ have at least one common character. Otherwise, the cover array is invalid and the function returns indicating that (see Lemma 4 later).

If a $Covervalue$ appears previously in C, then our algorithm uses a variable $lastpos$ to hold the immediate previous position of $CoverValue$ in the cover array. In this case, our algorithm will extend the string such that there is a cover of length equal to $CoverValue$ by putting the common characters in $X[CoverValue]$ *and* $X[lastpos]$ at Position i provided that the positions of $(X[CoverValue-1],\ X[lastpos-1]$ *and* $X[i-1]), \ldots, (X[1],\ X[lastpos-(CoverValue-1)]$ *and* $X[i-(CoverValue-1)])$ have at least one common character. Otherwise, the cover array is invalid and the function returns indicating that (see Lemma 5 later).

Now, we focus our attention on how we can effectively check whether multiple positions have common characters among them. To find whether there exists common character at two positions, we use Bit Vector technique [4]. In our algorithm, we use ν to indicate Bit Vector. Although we are reconstructing over an unbounded alphabet, when we compare between two positions for common characters we have already placed characters in those positions previously. We will also create the Bit Vector again if new characters arrive at that position. If two positions have common characters then we save this record in a two dimensional array H. For example, if

Positions a and b have common characters then we mark the entry of $H[a][b]$. We will update H incrementally. For example, for Position 2, we need to check Position 1 and 2 whether they have common characters or not. Again, for Position 3, we need to check Positions 1 and 3 and Positions 2 and 3 whether they have common characters or not. So we can see for Position 3, there are two entries to update in H namely $H[1][3]$ and $H[2][3]$. It is notable that for any Position i, all the entries of $H[1][1]$, $H[1][2]$, $\ldots$, $H[i-1][i-1]$ will remain unchanged. Because even if new character arrives, it will be placed in the positions stated in Lemma 2 and according to that lemma $X[1..i-1]$ will still be $C-Equivalent$. Now for Position n, we have at most $n-1$ entries such as $H[1][n], H[2][n], H[3][n], \ldots, H[n-1][n]$ to update. That is how, we have pre-computed H while placing characters in X. If we need to check whether there exists common characters between three positions namely a, b, c, we need to AND the Bit Vector of these three positions. If the result of this AND is non-zero then we can say there exists common character between Positions a, b and c.

In order to place characters of A_i in proper positions our algorithm uses function $PlaceCharInProperPosition(Position,\ CoverValue,\ necessarychar)$. Here $necessarychar$ indicates the necessary character to fill the positions of X. Whenever some $C^p[i] \neq 0$, ***CrAyDSRUn*** puts a character $v \in A_i$ into $X[i]$; v is also included in $X[C^p[i]]$ and $X[j]$ *where* $2 \leq j < i$ *and* $C^p[i] \in C[j]$ if $v \notin \Sigma_i$. It is notable that we have included character set $\Sigma_i - A_i - A'_i$ at Position i. We can safely add those characters because they are not invalid at Position i. By adding these characters we make sure that we do not need to add any more characters in the previous positions of Position i if no new characters arrive. We now report the following Observations which basically support the correctness of our approach.

Lemma 4. *Given a cover array* $C[1..n]$*, suppose* $C[i] = \ell$ *(we need to have a cover of length* ℓ *at Position i) such that* $\ell \notin \{C[1] \cup C[2] \cup \cdots \cup C[i-1]\}$ *and* $X[1] \cap X[i-(\ell-1)] \neq \phi$, $X[2] \cap X[i-(\ell-2)] \neq \phi, \ldots, X[\ell-1] \cap X[i-1] \neq \phi$ *then we must include* $X[\ell] - A'_i$ *at position* i*. If in any one of the above intersection returns* ϕ *then the input cover array is not valid.*

Proof. *Proof will be provided in the journal version.* □

Lemma 5. *Given a cover array* $C[1..n]$*, suppose* $C[i] = \ell$ *(we need to have a cover of length* ℓ *at Position i) such that* $\ell \in \{C[1] \cup C[2] \cup \cdots \cup C[i-1]\}$ *and let* p *be the immediate previous position of* i *where* $\ell \in C[p]$ *and* $1 \leq p \leq (i-1)$ *and* $\{X[1] \cap X[p-(\ell-1)] \cap X[i-(\ell-1)]\} \neq \phi$, $\{X[2] \cap X[p-(\ell-2)] \cap X[i-(\ell-2)]\} \neq \phi, \ldots, \{X[\ell-1] \cap X[p-1] \cap X[i-1]\} \neq \phi$*, then we must include* $X[\ell] \cap X[p] - A'_i$ *at position* i*. If in any one of the above intersection returns* ϕ *then the input cover array is invalid. Because if any one of the intersection returns* ϕ*, then we can not have a cover of length* ℓ *at Position* i*.*

Proof. *Proof will be provided in the journal version.* □

Based on the above discussions we have the following theorem.

Theorem 3 *Given a valid cover array* $C[1..n]$*, the algorithm* **CrAyDSRUn** *checks for its validity at every position and as long as it is valid it reconstructs a degenerate string* $X[1..n]$ *on an unbounded alphabet for which* $C[1..n]$ *is a cover array.*

Proof. *Proof will be provided in the journal version.* □

Now we focus on the complexity of algorithm ***CrAyDSRUn***. We start with the following theorem.

Theorem 4 *Algorithm* **CrAyDSRUn** *runs in* $O(N\ |\Sigma|)$ *time, where* N *is the product of string length and maximum list length of cover array* C.

Proof. Proof will be provided in the journal version. □

Theorem 5 *Algorithm* **CrAyDSRUn** *runs in linear time on average.*

Proof. Proof will be provided in the journal version. □

Algorithm 1 ***CrAyDSRUn***(C,n)

```
if $C[1] \neq \{0\}$ then
    return $C$ invalid at index 1
end if
$X[1] \leftarrow \{\alpha[1]\}$
$k[1] \leftarrow 1$
for $i \leftarrow 2$ to $n$ do
    $k[i] \leftarrow k[i-1]$
    $A \leftarrow \phi$
    $X[i] \leftarrow \phi$
    $\pi \leftarrow \{1..i-1\}$
    $A'_i \leftarrow \phi$
    for all $k,\ \pi - C[i]$ do
        $getInvalidChar(i, k)$
    end for
    $probablevalidchar \leftarrow \phi$
    for $j \leftarrow 1$ to $|C[i]|$ do
        if $C^j[i] \neq \{0\}$ then
            if $C^j[i] \notin \pi_i$ then
                return invalid at index $i$
            end if
            if $1 \in C[i]$ then
                if Observation 1a & Lemma 1 not satisfied then
                    return invalid at index $i$
                end if
            end if
            if Observation 1b not satisfied at position $i$ then
                return invalid at index $i$
            end if
            if Observation 1c not satisfied at position $i$ then
                return invalid at index $i$
            end if
            $getProbableValidChar(i, C^j[i])$
            $A \leftarrow probablevalidchar - A'_i$
            if $A \neq \phi$ then
                $X[i] \leftarrow X[i] \cup A$
                if $\Sigma_i - A_i - A'_i \neq \phi$ then
                    Add $\Sigma_i - A_i - A'_i$ at position $i$
                end if
                update $\nu$ position $i$
            else
                $k[i] \leftarrow k[i-1] + 1$
                $A \leftarrow \{\alpha[k[i]]\}$
                $placecharacterinproperposition(i, C^j[i], A)$
            end if
        else
            $k[i] \leftarrow k[i-1] + 1$
            $A \leftarrow \{\alpha[k[i]]\}$
            $X[i] \leftarrow X[i] \cup A$
            update $\nu$ at postition $i$
        end if
    end for
end for
return $X$
```

```
function getInvalidChar($position, covervalue$)
    $d \leftarrow covervalue - 1$
    $k \leftarrow position - 1$
    $flag \leftarrow 0$
    $flag2 \leftarrow 0$
    $lastpos \leftarrow 0$
    $lastpos \leftarrow find\ immediate\ previous\ position\ i\ of\ position\ where\ covervalue \in\ c[i]\&\ 1 \leq i \leq position - 1$
    if $covervalue = 1$ then
        if $covervalue \in c[k]$ then
            $A'_i \leftarrow\ A'_i \cup X[covervalue]$
            return $A'_i$
        else
            return $false$
        end if
    else if $lastpos = 0$ then
        for $i \leftarrow 1\ to\ covervalue - 1$ do
            $p \leftarrow \phi$
            $b \leftarrow position\%covervalue$
            $p \leftarrow X[i] \cap X[i + b]$
            if $p = \phi$ then
                $flag \leftarrow 1$
                $break$
            end if
        end for
    else if $lastpos \neq\ 0$ then
        $k \leftarrow position\ -\ covervalue\ +\ 1$
        $j \leftarrow\ 1$
        for $i \leftarrow\ lastpos - covervalue + 1\ to\ lastpos - 1$ do
            $p \leftarrow \phi$
            $p \leftarrow X[i] \cap X[k] \cap X[j]$
            $j + +$
            $k + +$
            if $p = \phi$ then
                $flag2 \leftarrow 1$
                $break$
            end if
        end for
    end if
    if $lastpos\ = 0\&\ flag = 0$ then
        $A'_i \leftarrow A'_i \cup X[covervalue]$
        return $A'_i$
    else if $lastpos\ = 0\& flag = 1$ then
        return $false$
    end if
    if $lastpos \neq 0\&\ flag2 = 0$ then
        $A'_i \leftarrow A'_i \cup (X[covervalue] \cap\ X[lastpos])$
        return $A'_i$
    else if $lastpos \neq 0\&\ flag2 = 1$ then
        return $false$
    end if
end function
```

```
function getProbaleValidChar($position, covervalue$)
    $d \leftarrow covervalue - 1$
    $k \leftarrow position - 1$
    $lastpos \leftarrow 0$
    $flag \leftarrow 0$
    $flag2 \leftarrow 0$
    $lastpos \leftarrow find\ immediate\ previous\ position\ i\ of\ position\ where\ covervalue \in\ c[i]\&\ 1 \leq i \leq position - 1$
    if $covervalue = 1$ then
        if $covervalue \in c[k]$ then
            $probablevalidchar \leftarrow\ probablevalidchar \cup X[covervalue]$
            return
        else
            return $invalid\ at\ position$
        end if
    else if $lastpos = 0$ then
        for $i \leftarrow 1\ to\ covervalue - 1$ do
            $p \leftarrow \phi$
            $b \leftarrow position\%covervalue$
            $p \leftarrow X[i] \cap X[i + b]$
            if $p = \phi$ then
```

```
                $flag \leftarrow 1$
                $break$
            end if
        end for
    else if $lastpos \neq\ 0$ then
        $k \leftarrow position\ -\ covervalue\ +\ 1$
        $j \leftarrow\ 1$
        for $i \leftarrow\ lastpos - covervalue + 1\ to\ lastpos - 1$ do
            $p \leftarrow \phi$
            $p \leftarrow X[i] \cap X[k] \cap X[j]$
            $j++$
            $k++$
            if $p = \phi$ then
                $flag2 \leftarrow 1$
                $break$
            end if
        end for
    end if
    if $lastpos\ = 0\&\ flag = 0$ then
        $probablevalidchar \leftarrow probablevalidchar \cup X[covervalue]$
        return $probablevalidchar$
    else if $lastpos\ = 0\& flag = 1$ then
        return $invalid\ at\ position$
    end if
    if $lastpos \neq 0\&\ flag2 = 0$ then
        $probablevalidchar \leftarrow probablevalidchar \cup (X[covervalue] \cap\ X[lastpos])$
        return $probablevalidchar$
    else if $lastpos \neq 0\&\ flag2 = 1$ then
        return $invalid\ at\ position$
    end if
end function
```

```
function PlaceCharInProperPosition($position, covervalue, necessarychar$)
    $b \leftarrow position\%covervalue$
    if $b \neq 0$ then
        for ($i \leftarrow covervalue;\ i \leq\ position;\ i \leftarrow i + b$) do
            $X[i] \leftarrow\ X[i] \cup necessarychar$
            $update\ \nu\ at\ position\ i$
        end for
    else if $b = 0$ then
        for ($i \leftarrow covervalue;\ i \leq\ position;\ i \leftarrow i + covervalue$) do
            $X[i] \leftarrow\ X[i] \cup necessarychar$
            $update\ \nu\ at\ position\ i$
        end for
    end if
end function
```

Table 7 shows an example run of the algorithm.

4.2 *CrAyDSRin*

Now we present a modified version of algorithm ***CrAyDSRUn*** that reconstructs a degenerate string using a minimum sized alphabet. We call this algorithm ***CrAyDSRin*** (Cover Array Degenerate String Reconstruction from Minimal Alphabet). As before, suppose we are reconstructing a degenerate string $X = X[1..n]$ from a cover array $C[1..n]$ and assume that we have successfully reconstructed $X[1..i-1]$. Now, we want to extend $X[1..i-1]$ to get $X[1..i]$ based on $C[1..i]$. Recall from Section 4.1 that, we use A'_i and A_i to denote the set of symbols that, respectively, are not allowed and allowed to be assigned to $X[i]$. Now we present an extended version of Lemma 3.b below.

Lemma 6. *For every degenerate string* $X[1..i]$*, if* $C^p[i] = 0$ *is the only entry in* $C[i]$*, then* $C^p[i] \notin \pi_i$ *and* $A_i = \psi_i \cup (\Sigma_{i-1}\ -\ A'_i)$*.*

Proof. Proof will be provided in the journal version. □

Algorithm 2 CrAyDSRIn(C,n)

```
if $C[1] \neq \{0\}$ then
    return $C$ invalid at index 1
end if
$X[1] \leftarrow \{\alpha[1]\}$
$k[1] \leftarrow 1$
for $i \leftarrow 2$ to $n$ do
    $k[i] \leftarrow k[i-1]$
    $A \leftarrow \phi$
    $X[i] \leftarrow \phi$
    $\pi \leftarrow \{1..i-1\}$
    $A'_i \leftarrow \phi$
    for all $k,\ \pi - C[i]$ do
        $getInvalidChar(i,k)$
    end for
    $probablevalidchar \leftarrow \phi$
    for $j \leftarrow 1$ to $|C[i]|$ do
        if $C^j[i] \neq \{0\}$ then
            if $C^j[i] \notin \pi_i$ then
                return $invalid\ at\ index\ i$
            end if
            if $1 \in C[i]$ then
                if $Observation$ 1a & $Lemma$ 1 $not\ satisfied$ then
                    return $invalid\ at\ index\ i$
                end if
            end if
            if $Observation$ 1b $not\ satisfied\ at\ position\ i$ then
                return $invalid\ at\ index\ i$
            end if
            if $Observation$ 1c $not\ satisfied\ at\ position\ i$ then
                return $invalid\ at\ index\ i$
            end if
            $getProbableValidChar(i, C^j[i])$
            $A \leftarrow probablevalidchar - A'_i$
            if $A \neq \phi$ then
                $X[i] \leftarrow X[i] \cup A$
                if $\Sigma_i - A_i - A'_i \neq \phi$ then
                    $Add\ \Sigma_i - A_i - A'_i\ at\ position\ i$
                end if
                $update\ \nu\ position\ i$
            else
                if $j = 1$ then
                    $k[i] \leftarrow k[i-1] + 1$
                    $A \leftarrow \{\alpha[k[i]]\}$
                else
                    for $m \leftarrow 1\ to\ j-1$ do
                        if $C^j[i] \in C[C^m[i]]$ then
                            $A \leftarrow A \cup (X[C^m[i]] - A'_i)$
                            $break$
                        end if
                    end for
                    if $m = j$ then
                        $k[i] \leftarrow k[i] + 1$
                        $A \leftarrow \{\alpha[k[i]]\}$
                    end if
                end if
                $placecharainproperposition(i, C^j[i], A)$
            end if
        else
            $A \leftarrow \alpha[1..k[i]] - A'_i$
            if $A = \phi$ then
                $k[i] \leftarrow k[i] + 1$
                $A \leftarrow \{\alpha[k[i]]\}$
            end if
            $X[i] \leftarrow X[i] \cup A$
            $update\ \nu\ at\ position\ i$
        end if
    end for
end for
return $X$
```

Itn	i	1	2	3	4	5	6	7	8	9	$k[i]$	Explanation
0	$X[i]$	a									k[1]=1	
1	$X[i]$	a	a								k[2]=1	$\pi_2 = \{1\}$
												$A'_2 = \phi,\ A_2 = \{a\}$
2	$X[i]$	a	a	a							k[3]=1	$\pi_3 = \{1, 2\}$
												$A'_3 = \phi,\ A_3 = \{a\}$
3	$X[i]$	a	a	a	a						k[4]=1	$\pi_4 = \{1, 2, 3\}$,
												$A'_4 = \phi,\ A_4 = \{a\}$
4	$X[i]$	a	a	a	a	a					k[5]=1	$\pi_5 = \{1, 2, 3, 4\}$,
												$A'_5 = \phi,\ A_5 = \{a\}$
5	$X[i]$	a	a	a	a	a	b				k[6]=3	$\pi_6 = \{1, 2, 3, 4, 5\},\ A'_6 = \{a\}$
					b		c					for $c^1[6] = 4$ place new symbol 'b' in position 4, and 6
			c		c							for $c^2[6] = 2$ place new symbol 'c' in position 2,4, and 6
												$A_6 = \psi_6 = \{b, c\}$
6	$X[i]$	a	a	a	a	a	b	a			k[7]=3	$\pi_7 = \{1, 2, 3, 4, 5, 6\},\ A'_7 = \phi$
					b		c	b				$A_7 = \{a\},\ \Sigma_7 - A_7 - A'_7 = \{b, c\}$
			c		c			c				
7	$X[i]$	a	a	a	a	a	b	a	b		k[8]=3	$\pi_8 = \{1, 2, 3, 4, 5, 6, 7\},\ A'_8 = \{c\}$,
			c		b		c	b	a			$A_8 = \{b\},\ \Sigma_8 - A_8 - A'_8 = \{a\}$
					c			c				
8	$X[i]$	a	a	a	a	a	b	a	b	d	k[9]=4	$\pi_9 = \{1, 2, 3, 4, 5, 6, 7\},\ A'_9 = \{a, b, c\}$
			c		b		c	b	a			$A_9 = \psi_9 = \{d\}$
					c			c				

Table 7. An example run of algorithm ***CrAyDSRUn***

The algorithm ***CrAyDSRin*** is formally presented in Algorhitm 2. ***CrAyDSRin*** algorithm works exactly like ***CrAyDSRUn*** algorithm except for that it computes A_i slightly differently. In particular, it computes A_i following Lemmas 3.a and 6 (instead of Lemma 3.b).

Lemma 7. *Let* $X[1..n]$ *be a degenerate string and* $k[1..n]$ *be the array computed by the algorithm* **CrAyDSRin** *given a valid cover array* C*. Then, for* $1 \leq i \leq n$ *we have* $k[i] = |\Sigma_{i-1} \cup A_i| = k[i-1] + |A_i| - |\Sigma_{i-1} \cap A_i|$.

Proof. The proof immediately follows from the algorithm ***CrAyDSRin*** and Lemma 6. □

Lemma 8. *Suppose given a valid cover array* $C[1..n]$*, the algorithm* **CrAyDSRin** *returns an degenerate string* $X[1..n]$ *and computes the array* $k[1..n]$*. Then, the minimum cardinality of an alphabet required to build each prefix* $X[1..i]$ *is equal to* $k[i]$.

Proof. Proof will be provided in the journal version. □

The above discussion can be summarized in the following theorem.

Theorem 6 *Given a cover array $C[1..n]$ the algorithm* **CrAyDSRin** *checks for its validity at every position and as long as it is valid it reconstructs an indeterminate string $X[1..n]$ on a minimum sized alphabet for which $C[1..n]$ is a cover array.*

The runtime analysis of algorithm ***CrAyDSRin*** follows readily from the analysis of algorithm ***CrAyDSRUn***. The only extra work the former does is the calculation of $\Sigma_{i-1} - A'_i$ which can be done in $O(m|\Sigma|)$ time. Therefore we have the following results.

Theorem 7 *Algorithm* **CrAyDSRin** *runs in $O(N|\Sigma|)$ time, where N is the the product of string length and maximum list length of the cover array C.*

Corollary 9. *Algorithm* **CrAyDSRin** *runs in linear time on average.*

Itn	i	1	2	3	4	5	6	7	8	9	$k[i]$	Explanation
0	$X[i]$	a									k[1]=1	
1	$X[i]$	a	a								k[2]=1	$\pi_2 = \{1\}$ $A'_2 = \phi,\ A_2 = \{a\}$
2	$X[i]$	a	a	a							k[3]=1	$\pi_3 = \{1,2\}$ $A'_3 = \phi,\ A_3 = \{a\}$
3	$X[i]$	a	a	a	a						k[4]=1	$\pi_4 = \{1,2,3\}$ $A'_4 = \phi,\ A_4 = \{a\}$
4	$X[i]$	a	a	a	a	a					k[5]=1	$\pi_5 = \{1,2,3,4\}$ $A'_5 = \phi,\ A_5 = \{a\}$
5	$X[i]$	a	a b	a	a b	a	b				k[6]=2	$\pi_6 = \{1,2,3,4,5\},\ A'_6 = \{a\}$ for $c^1[6] = 4$ place new symbol 'b' in position 4 and 6 for $c^2[6] = 2,\ c^2[6] \in c[c^1[6]]$ $A_6 = \psi_6 = \{b\}$
6	$X[i]$	a	a b	a	a b	a	b	a b			k[7]=2	$\pi_7 = \{1,2,3,4,5,6\},\ A'_7 = \phi$ $A_7 = \{a\},\ \Sigma_7 - A_7 - A'_7 = \{b\}$
7	$X[i]$	a	a b	a	a b c	a	b c	a b	c a		k[8]=3	$\pi_8 = \{1,2,3,4,5,6,7\},\ A'_8 = \{b\}$, for $c^1[8] = 6$ place new symbol 'c' in position 6 and 8 for $c^2[8] = 2,\ c^2[8] \in c[c^1[8]]$ $A_8 = \psi_8 = \{c\},\ \Sigma_8 - A_8 - A'_8 = \{a\}$
8	$X[i]$	a	a b	a	a b c	a	b c	a b	c a	c	k[9]=3	$\pi_9 = \{1,2,3,4,5,6,7,8\},\ A'_9 = \{a,b\}$ $A_9 = \{a,b,c\} - \{a,b\} = \{c\}$

Table 8. An example run of algorithm CrAyDSin

Table 8 shows an example run of **CrAyDSRin** *Algorithm.*

5 Conclusion

In this paper, we have presented efficient algorithms for inferring a degenerate string given a valid cover array. We have presented two algorithms both of which returns a degenerate string from a given cover array, if the cover array is valid. Our first algorithm infers a degenerate string on an unbounded alphabet satisfying the cover array and our second algorithm infers a degenerate string on a least size. Future research may be carried out for devising similar reconstruction algorithms for degenerate strings considering other data structures (e.g., seed array).

References

1. P. Antoniou, M. Crochemore, C. S. Iliopoulos, I. Jayasekera, and G. M. Landau: *Conservative string covering of indeterminate strings.* Proceedings of the Prague Stringology Conference 2008, 2008, pp. 108–115.
2. P. Antoniou, C. S. Iliopoulos, I. Jayasekera, and W. Rytter: *Computing epetitive structures in indeterminate strings.* Proceedings of the 3rd IAPR International Conference on Pattern Recognition in Bioinformatics (PRIB 2008), 2008, pp. 108–115.
3. A. Apostolico and A. Ehrenfeucht: *Efficient detection of quasiperiodicities in strings.* tcs, 119(2) 1993, pp. 247–265.
4. M. F. Bari, M. S. Rahman, and R. Shahriyar: *Finding all covers of an indeterminate string in $O(n)$ time on average*, in Proceedings of the Prague Stringology Conference 2009, J. Holub and J. Žďárek, eds., Czech Technical University in Prague, Czech Republic, 2009, pp. 263–271.
5. M. Crochemore, C. S. Iliopoulos, S. P. Pissis, and G. Tischler: *Cover array string reconstruction*, in CPM'10, 2010, pp. 251–259.
6. J.-P. Duval, T. Lecroq, and A. Lefebvre: *Border array on bounded alphabet.* Journal of Automata, Languages and Combinatorics, 2005, pp. 51–60.
7. F. Franek, S. Gao, W. Lu, P. J. Ryan, W. F. Smyth, Y. Sun, and L. Yang: *Verifying a border array in linear time.* J. Comb. Math. Comb. Comput, 42 2000, pp. 223–236.
8. J. Holub and W. F. Smyth: *Algorithms on indeterminate strings.* Miller, M., Park, K. (eds.): Proceedings of the 14th Australasian Workshop on Combinatorial Algorithms AWOCA'03, 2003, pp. 36–45.
9. J. Holub, W. F. Smyth, and S. Wang: *Fast pattern-matching on indeterminate strings.* Journal of Discrete Algorithms, 6(1) 2008, pp. 37–50.
10. C. S. Iliopoulos, M. Mohamed, L. Mouchard, K. G. Perdikuri, W. F. Smyth, and A. K. Tsakalidis: *String regularities with don't cares.* Nordic Journal of Computing, 10(1) 2003, pp. 40–51.
11. C. S. Iliopoulos, L. Mouchard, and M. S. Rahman: *A new approach to pattern matching in degenerate DNA/RNA sequences and distributed pattern matching.* Mathematics in Computer Science, 1(4) 2008, pp. 557–569.
12. T. M. Moosa, S. Nazeen, M. S. Rahman, and R. Reaz: *Linear time inference of strings from cover arrays using a binary alphabet – (extended abstract)*, in WALCOM'12, 2012, pp. 160–172.
13. S. Nazeen, M. S. Rahman, and R. Reaz: *Indeterminate string inference algorithms.* J. Discrete Algorithms, 2012, pp. 23–34.
14. W. F. Smyth and S. Wang: *New perspectives on the prefix array*, in SPIRE, A. Amir, A. Turpin, and A. Moffat, eds., vol. 5280 of Lecture Notes in Computer Science, Springer, 2008, pp. 133–143.
15. W. F. Smyth and S. Wang: *An adaptive hybrid pattern-matching algorithm on indeterminate strings.* Int. J. Found. Comput. Sci., 2009, pp. 985–1004.