

# Linear Algorithm for Conservative Degenerate Pattern Matching

Maxime Crochemore<sup>1</sup>, Costas S. Iliopoulos<sup>1</sup>, Ritu Kundu<sup>1</sup>,  
Manal Mohamed<sup>1</sup> and Fatima Vayani<sup>1</sup>

<sup>1</sup>Department of Informatics, King's College London

July 16, 2018

## Abstract

A *degenerate symbol*  $\tilde{x}$  over an alphabet  $\Sigma$  is a non-empty subset of  $\Sigma$ , and a sequence of such symbols is a *degenerate string*. A degenerate string is said to be *conservative* if its number of non-solid symbols is upper-bounded by a fixed positive constant  $k$ . We consider here the matching problem of conservative degenerate strings and present the first linear-time algorithm that can find, for given degenerate strings  $\tilde{P}$  and  $\tilde{T}$  of total length  $n$  containing  $k$  non-solid symbols in total, the occurrences of  $\tilde{P}$  in  $\tilde{T}$  in  $O(nk)$  time.

## 1 Introduction

*Degenerate*, or *indeterminate*, strings are found in Biology, Musicology and Cryptography. They are defined by the occurrence of one or more positions which are represented by sets of symbols. In *conservative degenerate strings*, the number of such occurrences is bounded by  $k$ . In music, single notes may match chords. In encrypted and biological sequences, a position in one string may match exactly with various symbols in other strings.

Previous algorithmic research of degenerate strings has been focused on pattern matching. Pattern matching in degenerate strings is particularly relevant in the context of coding biological sequences. Due to the degeneracy of

the genetic code, two dissimilar DNA sequences can be translated into two identical protein sequences. Without taking this degeneracy into account, many associations between biological entities can be overlooked. For example, the following six DNA codons are all translated into the amino acid Leucine: *TTA*, *TTG*, *CTT*, *CTC*, *CTA* and *CTG*. This example highlights the significance of solving problems relating to degeneracy in strings. In fact, special symbols to represent sets of DNA symbols have long been established by the IUPAC-IUBMB Biochemical Nomenclature Committee [1]. For example, *R* represents any purine (*A* or *G*), *Y* represents any pyrimidine (*C*, *T* or *U*) and *N* represents any nucleic acid. An example of practical implications of such research is in the design of primers for cloning DNA sequences using PCR (Polymerase Chain Reaction). Degenerate primers are used when their design is based on protein sequences, which can be reverse-translated to  $n^k$  different sequences, where  $n$  is the length of the sequence.

This paper introduces an algorithm which is a significant improvement from those published previously. The first significant contribution for the problem of pattern matching of degenerate strings was in 1974 [2], and was later improved [3]. Later still, faster algorithms for the same problem were proposed [4, 5]. Since, many practical methods have been suggested [6–8], as well as variations of the problem considered. For example, a non-practical generalised string matching algorithm was introduced by Abrahamson in 1987 [9]. Most recently, Crochemore *et al.* [10] reported an algorithm to find the shortest solid cover in a degenerate string with time complexity  $O(2^k)$ . We report here a major improvement in time:  $O(kn)$ . Further to the problem of pattern matching, the linear algorithm reported here can be applied to many different problems, including finding cover and prefix arrays.

The rest of the paper is organised in the following format: The next section introduces the vocabulary and the notions that will be used in this paper. Section 3 formally defines the problem and presents the algorithm we have proposed. The algorithm is analysed in Section 4 and finally, Section 5 concludes the paper.

## 2 Preliminaries

To provide an overview of our results we begin with a few definitions, generally following [8, 10]. An *alphabet*  $\Sigma$  is a non-empty finite set of symbols of size  $|\Sigma|$ . A *string* over a given alphabet is a finite sequence of symbols. The *length* of a string  $x$  is denoted by  $|x|$ . The *empty string* is denoted by  $\varepsilon$ . The set of all strings over an alphabet  $\Sigma$  (including empty string  $\varepsilon$ ) is denoted by  $\Sigma^*$ .

A *degenerate symbol*  $\tilde{x}$  over an alphabet  $\Sigma$  is a non-empty subset of  $\Sigma$ , i.e.,  $\tilde{x} \subseteq \Sigma$  and  $\tilde{x} \neq \emptyset$ .  $|\tilde{x}|$  denotes the size of the set and we have  $1 \leq |\tilde{x}| \leq |\Sigma|$ . A finite sequence  $\tilde{X} = \tilde{x}_1\tilde{x}_2 \dots \tilde{x}_n$  is said to be a *degenerate string* if  $\tilde{x}_i$  is a degenerate symbol for each  $i$  from 1 to  $n$ . In other words, a *degenerate string* is built over the potential  $2^{|\Sigma|} - 1$  non-empty sets of letters belonging to  $\Sigma$ . The number of the degenerate symbols,  $n$  here, in a degenerate string  $\tilde{X}$  is its *length*, denoted as  $|\tilde{X}|$ . For example,  $\tilde{X} = [a, b][a][c][b, c][a][a, b, c]$  is a degenerate string of length 6 over  $\Sigma = [a, b, c]$ . If  $|\tilde{x}_i| = 1$ , that is,  $\tilde{x}_i$  represents a single symbol of  $\Sigma$ , we say that  $\tilde{x}_i$  is a *solid symbol* and  $i$  is a *solid position*. Otherwise  $\tilde{x}_i$  and  $i$  are said to be *non-solid symbol* and *non-solid position* respectively. For convenience we often write  $\tilde{x}_i = c$  ( $c \in \Sigma$ ), instead of  $\tilde{x}_i = [c]$ , in case of solid symbols. Consequently, the degenerate string  $\tilde{X}$  mentioned in the example previously will be written as  $[a, b]ac[b, c]a[a, b, c]$ . A string containing only solid symbols will be called a *solid string*. Also as a convention, capital letters will be used to denote strings while small letters will be used for representing symbols. Furthermore, the degeneracy will be indicated by a *tilde*, for example,  $\tilde{X}$  denotes a *degenerate string* while a plain letter like  $X$  represents a *solid string*. The empty degenerate string is denoted by  $\tilde{\varepsilon}$ .

A *conservative degenerate string* is a degenerate string where its number of non-solid symbols is upper-bounded by a fixed positive constant  $k$ . The concatenation of degenerate strings  $\tilde{X}$  and  $\tilde{Y}$  is  $\tilde{X}\tilde{Y}$ . A degenerate string  $\tilde{V}$  is a *substring* (resp. *prefix*, *suffix*) of a degenerate string  $\tilde{X}$  if  $\tilde{X} = \tilde{U}\tilde{V}\tilde{W}$  (resp.  $\tilde{X} = \tilde{V}\tilde{W}$ ,  $\tilde{X} = \tilde{U}\tilde{V}$ ) for some degenerate strings  $\tilde{U}$  and  $\tilde{W}$ . By  $\tilde{X}[i..j]$ , we represent a substring  $\tilde{x}_i\tilde{x}_{i+1} \dots \tilde{x}_j$  of  $\tilde{x}$ .

For degenerate strings, the notion of symbol equality is extended to single-symbol *match* between two degenerate symbols in the following way. Two degenerate symbols  $\tilde{x}$  and  $\tilde{y}$  are said to *match* (represented as  $\tilde{x} \approx \tilde{y}$ ) if

$\tilde{x} \cap \tilde{y} \neq \emptyset$ . Extending this notion to degenerate strings, we say that two degenerate strings  $\tilde{X}$  and  $\tilde{Y}$  *match* (denoted as  $\tilde{X} \approx \tilde{Y}$ ) if  $|\tilde{X}| = |\tilde{Y}|$  and corresponding symbols in  $\tilde{X}$  and  $\tilde{Y}$  match, i.e., for each  $i = 1, \dots, |\tilde{X}|$  we have  $\tilde{x}_i \approx \tilde{y}_i$ . Note that the relation  $\approx$  is not transitive. A degenerate string  $\tilde{X}$  is said to *occur* at position  $i$  in another degenerate (resp. solid) string  $\tilde{Y}$  (resp.  $Y$ ) if  $\tilde{X} \approx \tilde{Y}[i..i + |\tilde{X}| - 1]$  (resp.  $\tilde{X} \approx Y[i..i + |\tilde{X}| - 1]$ ).

### 3 Conservative Degenerate String Matching

**Problem 1.** Given a conservative degenerate pattern  $\tilde{P}$  with  $k$  non-solid symbols, and a solid text  $T$ , find all positions in  $T$  at which  $\tilde{P}$  occurs.

**Example 1.** We consider a degenerate pattern,  $\tilde{P} = a[bc]da[bd]$  with  $k = 2$  and a text,  $T = dacdabdadcabdac$ . Table 1 shows that  $\tilde{P}$  occurs in  $T$  at positions 2 and 5.

Table 1: Occurrence of  $\tilde{P}$  in  $T$

$i$	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$t$	$d$	$a$	$c$	$d$	$a$	$b$	$d$	$a$	$d$	$c$	$a$	$b$	$d$	$a$	$c$
Matches		$a$	$[bc]$	$d$	$a$	$[bd]$									
					$a$	$[bc]$	$d$	$a$	$[bd]$						

For convenience, we compute a table  $Pre[k, |\Sigma|]$  such that for each non-solid position  $i$  ( $1 \leq i \leq k$ ) and each letter  $a \in \Sigma$ , we have  $Pre[i, a] = 1$  if  $a \in \tilde{P}[i]$  and 0 otherwise. After such  $O(k|\Sigma|)$ -time preprocessing, we can check in  $O(1)$  time whether a non-solid position in  $\tilde{P}$  matches a position in  $T$  or not.

### An Outline of Our Approach

Our algorithm to solve Problem 1 is built on the top of an adapted version of the sequential algorithm presented by Landau and Vishkin to find all occurrences of a (solid) pattern  $P$  of length  $m$  in a (solid) text  $T$  of length  $n$  with at most  $e$  differences each [11], where a difference can be due to either a mismatch between the corresponding characters of the text and the pattern, or a superfluous character in the text, or a superfluous character in the pattern. The modification required for our strategy is to treat only

mismatches as the differences in Landau and Vishkin's algorithm. On the lines of the original Landau and Vishkin's algorithm, the modified one works in the following two steps .

Step 1: Compute the suffix tree of the string obtained after concatenating the text, the pattern and a character  $\#$  which is not present in  $\Sigma \cup \Lambda$ , i.e.  $TP\#$ ; using the serial algorithm of Weiner [12].

Step 2: Let  $Mismatch_{i,j}$  be the position in the pattern at which we have  $j^{th}$  mismatch (when defined) between  $T[i+1..i+m]$  and  $P[1..m]$ . In other words,  $Mismatch_{i,j} = f$  represents  $j^{th}$  mismatch from left to right and implies that  $t_{i+f} \neq p_f$ . In this step, we find  $Mismatch_{i,j}$  for each  $i$  and  $j$  such that  $0 \leq i \leq n-m$  and  $1 \leq j \leq c+1$  where  $c$  denotes the maximum of the two :  $e$  and the total number of mismatches between  $T[i+1..i+m]$  and  $P[1..m]$ . If some  $Mismatch_{i,j} = m+1$ , it signifies that there is an occurrence of the pattern in the text, starting at  $t[i+1]$ , with at most  $e$  mismatches.  $Mismatch_{i,j}$  can be computed from  $Mismatch_{i,j-1}$  as follows :

Let  $LCA_{si,sj}$  be the lowest common ancestor (in short LCA) of the leaves of the suffixes  $T[si+1, n]$  and  $P[sj+1]$  in the suffix tree and  $|LCA_{si,sj}|$  denotes its length.  $Mismatch_{i,j-1} = f$  implies that  $T[i+1..i+f]$  and  $P[1..f]$  is matched with  $j-1$  mismatches. We want to find the largest  $q$  such that  $T[i+f+1..i+f+q] = P[f+1..f+q]$  and  $t_{i+q+1} \neq p_{q+1}$ , so that  $Mismatch_{i,j} = q+1$ . The desired  $q$  is same as length of  $LCA_{i+f,f}$ . Thus,  $Mismatch_{i,j} = f + |LCA_{i+f,f}|$ .

Pseudocode for our approach is given as Algorithm 1. It works in the following three stages :

### STAGE 1: Substitute

In the first stage, each of the non-solid symbols occurring in the given degenerate pattern is replaced by a unique symbol which is not present in  $\Sigma$ .  $\Lambda$  represents the set of these unique symbols i.e.  $\{\lambda_i\}$  such that  $0 < i \leq k$ . It is to be noted that the pattern,  $p_\lambda$ , obtained by such a substitution will be a solid string. For example,  $P_\lambda$  obtained from  $\tilde{P}$  in Example 1 is given in Table 2.

**Definition 3.1.** We define  $\lambda$  positions as the positions in  $P_\lambda$  which contain  $\{\lambda_i\} \in \Lambda$ . Note that these are same as the non-solid positions in  $\tilde{P}$ .

Table 2: [STAGE 1: Substitute]  $P_\lambda$  obtained from  $\tilde{P}$

$\tilde{P}$	$a$	$[bc]$	$d$	$a$	$[bd]$
$P_\lambda$	$a$	$\lambda_1$	$d$	$a$	$\lambda_2$

## STAGE 2: Approximate Pattern Search

The next stage comprises of using modified Landau and Vishkin's algorithm to search pattern  $P_\lambda$  (solid) in text  $T$  (solid) with at most  $k$  mismatches in each occurrence. First, a suffix tree for the (solid) string  $TP_\lambda$  is constructed. Then, LCA queries on this suffix tree are used to compute  $Mismatch_{i,j}$  for each  $i$  and  $j$  such that  $0 \leq i \leq n - m$  and  $1 \leq j \leq k + 1$ . As explained in Remark 3.1,  $j$  will vary up to  $k + 1$  in  $P_\lambda$ 's case. Every  $i$ , such that  $Mismatch_{i,k+1} = m + 1$ , marks the beginning of an occurrence of  $P_\lambda$  in  $T$  (at  $i + 1$ ) and thus added to the set *ApproximateMatch*.

Figure 1 demonstrates the suffix tree for the string obtained from concatenating  $T$  from Example 1 and  $P_\lambda$  from the previous step, i.e  $TP_\lambda$  which is *dacdabdadcabdacal<sub>1</sub>dal<sub>2</sub>#*. Note that each node of the suffix tree is stored as a pair (start, length) that represents the contiguous substring  $S[start + 1..start + length]$ . In addition, a leaf node indicates the suffix it represents. A leaf node showing  $i$  denotes a suffix  $S[i + 1..|S|]$ . Table 3 shows the resultant  $Mismatch[0..n - m, 1..k + 1]$  array. This table provides the positions in  $P_\lambda$  where it mismatches with the corresponding character in  $T$ . For example,  $Mismatch[7, 1] = 2$  denotes that the first mismatch between  $T[8, 12]$  and  $P_\lambda$  occurs at position 2 in  $P_\lambda$  and rightly so as  $t[8 + 2] = t[10]$  (i.e.  $c$ ) does not match with  $p_\lambda[2]$  (i.e.  $\lambda_1$ ). As  $P_\lambda$  occurs in  $T$  with at most 2 mismatches at locations 2, 5 and 11 (rows 1, 4 and 10 contain 6, i.e.  $m + 1$ ), *ApproximateMatch* = [1, 4, 10].

**Remark 3.1.** *There will always be a mismatch between  $P_\lambda$  and  $T$  at  $\lambda$  positions as each of the  $\lambda_i \in \Lambda$  is unique and does not occur in  $\Sigma$  and hence in  $T$ . As there are  $k$   $\lambda$  positions, at least  $k$  mismatches are bound to be there for each position  $i$  in the text starting at which the pattern is being matched against. More explicitly, each occurrence recorded in *ApproximateMatch* has  $k$  mismatches exactly.*

Figure 1: [STAGE 2: APPROXIMATE PATTERN MATCH] Suffix Tree for  $TP_\lambda\#$

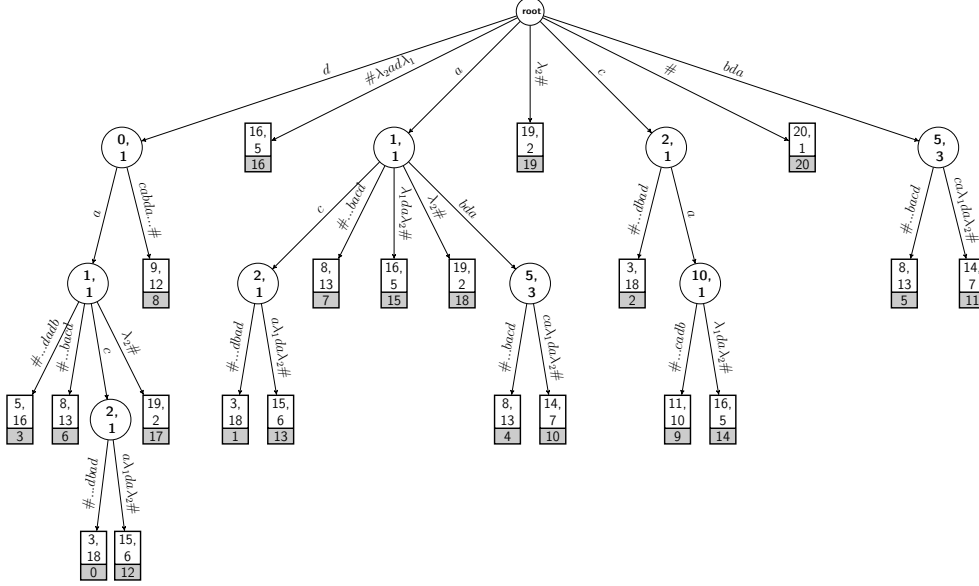


Table 3: [STAGE 2: Approximate Pattern Search] *Mismatch* array

$j \downarrow i \rightarrow$	0	1	2	3	4	5	6	7	8	9	10
1	1	2	1	1	2	1	1	2	1	1	2
2	2	5	2	2	5	2	2	3	2	2	5
3	3	6	3	3	6	3	4	5	3	3	6

### STAGE 3: Filter

An occurrence in *ApproximateMatch* reports a mismatch at a  $\lambda$  position even if there is a match at the corresponding non-solid position in reality. For example, if some  $\lambda_i$  has been substituted at a non-solid position containing, say  $[b, c]$ , and the corresponding symbol in  $T$  is  $c$ , clearly it is a match but that position will be recorded as a ‘mismatch’ in array *Mismatch* because  $\lambda_i$  does not match with  $c$ . Thus, a mismatch of all the  $k$  mismatches, found in an occurrence of solid  $P_\lambda$  in  $T$  identified by *ApproximateMatch* in the preceding step, can be seen as either *real* or *fake* when considered with respect to the match of the degenerate pattern  $\tilde{P}$  and  $T$ .

**Definition 3.2.** A mismatch at a position, say  $e = \text{Mismatch}[i, j]$ , is *real*

if the corresponding symbols in the degenerate pattern  $\tilde{P}$  and the text  $T$  mismatch, i.e.  $t[i + e] \not\approx \tilde{p}[e]$ . Otherwise, the mismatch is *fake*.

**Remark 3.2.** *A mismatch at a solid position will always be real while one at a  $\lambda$  position can either be real or fake.*

**Definition 3.3.** An *approximate occurrence* is an occurrence of  $P_\lambda$  in  $T$  with  $k$  mismatches whereas an occurrence of  $\tilde{P}$  in  $T$  with exact match is called an *exact occurrence*.

**Remark 3.3.** *It follows from Remarks 3.1 and 3.2 that if there is a mismatch even at a single solid position, total number of mismatches will exceed  $k$  and such an occurrence will not figure as an approximate occurrence. Conversely, an approximate occurrence will have mismatches only at  $\lambda$  positions.*

For each location  $i$  in the text where an *approximate occurrence* of  $P_\lambda$  has been found ( $i \in \text{ApproximateMatch}$ ), each position of mismatch ( $\lambda$  positions) in the pattern is checked for whether the mismatch is *real* or not. If an approximate occurrence of pattern  $P_\lambda$  in text  $T$  contains a real mismatch, it can be observed that it cannot represent an exact occurrence of  $\tilde{P}$  whereas the approximate occurrence containing only fake mismatches will be same as an exact occurrence. The set of all such exact occurrences is the solution to our Problem 1. This step, therefore, filters out and discards the approximate occurrences with real errors.

Table 4 elucidates this stage for the example being considered. With values given by  $\text{ApproximateMatch} = [1, 4, 10]$  from the previous stage, we test each  $\lambda$  position from  $\Lambda = [2, 5]$  to check if the mismatch is real or fake. At first  $\lambda$  position (i.e. 2),  $t[1 + 2]$  (i.e.  $c$ ) matches  $\tilde{p}[2]$  (i.e.  $[b, c]$ ), thus the mismatch is fake. The mismatch for the second  $\lambda$  position (i.e. 5) is also fake owing to the fact that  $t[6] \approx \tilde{p}[5]$ . Therefore, location 2 is recorded as an occurrence of exact match of  $\tilde{P}$  in  $T$ . Similar is the case of location 5 (i.e. value 4). But for value 10, even if the first mismatch is fake ( $t[12]$  (i.e.  $b$ )  $\approx \tilde{p}[2]$  (i.e.  $[b, c]$ )), the fact that  $t[15]$  (i.e.  $c$ )  $\not\approx \tilde{p}[5]$  (i.e.  $[b, d]$ ) makes the second mismatch real. Therefore, location 10 is discarded. And thus the correct solution to Example 1 is obtained.



Table 4: [STAGE 3: Filter] Checking Mismatches in Approximate Occurrences of  $\tilde{P}$  in  $T$

$i$	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$T$	$d$	$a$	$c$	$d$	$a$	$b$	$d$	$a$	$d$	$c$	$a$	$b$	$d$	$a$	$c$
Approximate Occurences	$a$		$\lambda_1$	$d$	$a$	$\lambda_2$									
			$\downarrow$			$\downarrow$									
			$[bc]$			$[bd]$									
			$\downarrow$			$\downarrow$									
			Fake			Fake									
				$a$	$\lambda_1$	$d$	$a$	$\lambda_2$							
						$\downarrow$			$\downarrow$						
						$[bc]$			$[bd]$						
						$\downarrow$			$\downarrow$						
						Fake			Fake						
										$a$	$\lambda_1$	$d$	$a$	$\lambda_2$	
											$\downarrow$			$\downarrow$	
											$[bc]$			$[bd]$	
											$\downarrow$			$\downarrow$	
											Fake			Real	

## 4 Algorithm Analysis

**Theorem 4.1.** *Algorithm 1 correctly computes all occurrences of  $\tilde{P}$  in  $T$  in  $O(kn)$  time complexity.*

*Proof.* Landau and Vishkin's algorithm correctly finds all occurrences of  $P_\lambda$  in  $T$  with at most  $k$  mismatches in  $O(kn)$  time for a fixed alphabet.  $P_\lambda$  differs from  $\tilde{P}$  only at the  $\lambda$  positions which are equal to  $k$  in number. In addition, each of the  $\lambda$  positions causes a mismatch. Notably, an exact occurrence of  $\tilde{P}$  in  $T$  will be given by an approximate occurrence of  $P_\lambda$  in  $T$  with mismatches only at  $\lambda$  positions and all of these mismatches must be fake. All such occurrences where mismatches occur only at  $k$   $\lambda$  positions are guaranteed to be captured by the approximate occurrences given in *ApproximateMatch*. Also, as a consequence of Remark 3.3, an approximate occurrence (for which number of mismatches are at most  $k$ ) will never have a mismatch at any solid position. The filtering stage checks each of the mismatches in an approximate occurrence and if all of these mismatches are found to be fake, we have an exact occurrence. Thus, at the end of the filtering stage, we have all the occurrences of an exact match only.

The substitution stage can be performed in  $O(n)$  time. As mentioned previously, the approximate pattern-search stage using modified Landau and Vishkin's algorithm computes *ApproximateMatch* in  $O(kn)$  time for a fixed

---

**Algorithm 1** Conservative Degenerate String Matching Algorithm

---

**Input:** Pattern  $\tilde{P}$  of length  $m$ ,

Text  $T$  of length  $n$ ,

Number of non-solid symbols  $k$

**Output:** The set of indices of  $T$  where  $\tilde{P}$  occurs in  $T$

▷ **Substitute:**

1:  $\Lambda \leftarrow \{\lambda_i \mid \lambda_i \notin \Sigma \text{ and } 0 < i \leq k\}$

2:  $P_\Lambda \leftarrow$  string obtained after substituting  $i^{th}$  non-solid symbol in  $\tilde{P}$  with  $\lambda_i$  in  $\Lambda \forall i$  such that  $0 < i \leq k$

▷ **Approximate Pattern Search:**

3: Build Suffix Tree for the string  $TP_\Lambda\#$

4:  $ApproximateMatch \leftarrow \emptyset$

5: **for**  $i \leftarrow 0$  **to**  $n - m$  **do**

6:      $f \leftarrow 0$

7:     **for**  $j \leftarrow 1$  **to**  $k + 1$  **do**

8:          $Mismatch[i, j] = f + |LCA_{i+f, f}|$

9:          $f \leftarrow Mismatch[i, j]$

10:     **end for**

11:     **if**  $Mismatch[i, k + 1] = m + 1$  **then**     \*\*\* approximate occurrence found \*\*\*

12:         Add  $i$  to  $ApproximateMatch$

13:     **end if**

14: **end for**

▷ **Filter:**

15:  $Occ \leftarrow \emptyset$

16: **for each**  $i \in ApproximateMatch$  **do**

17:      $flagAllFake \leftarrow \text{true}$

18:     **for each**  $e \in \Lambda$  **do**

19:         **if**  $t[i + e] \not\approx \tilde{p}[e]$  **then**

20:              $flagAllFake \leftarrow \text{false}$

21:             **Break**     \*\*\* real mismatch \*\*\*

22:         **end if**

23:     **end for**

24:     **if**  $flagAllFake$  **then**     \*\*\* all fake mismatches \*\*\*

25:         Add  $i + 1$  to  $Occ$      \*\*\* exact occurrence found \*\*\*

26:     **end if**

27: **end for**

28: **return**  $Occ$

---

sized alphabet as the suffix tree is constructed in linear time with respect to the size of the input string  $(n + m)$  and computation of  $Mismatch$  array

(lines 5 to 14) takes  $O(kn)$  time. The filtering stage, in the worst case (*ApproximateMatch* contains 0 to  $n - m$ ), needs to process each location in  $T$  and to check whether mismatch at every  $\lambda$  position is real or fake. This check can be performed in constant time after  $O(k|\Sigma|)$ -time pre-processing as mentioned earlier, which yields  $O(kn)$  time requirements for this stage. Thus, in  $O(k|\Sigma| + n + kn + kn) = O(kn)$  time Algorithm 1 correctly computes all occurrences of  $\tilde{P}$  in  $T$ .  $\square$

**Corollary.** *Given degenerate strings  $\tilde{P}$  and  $\tilde{T}$  of total length  $n$  containing  $k$  non-solid symbols in total, one can compute occurrences of  $\tilde{P}$  in  $\tilde{T}$  in  $O(nk)$  time.*

## 5 Conclusion

In this paper, we studied the matching problem of conservative degenerate strings and presented an efficient algorithm that can find, for given degenerate strings  $\tilde{P}$  and  $\tilde{T}$  of total length  $n$  containing  $k$  non-solid symbols in total, the occurrences of  $\tilde{P}$  in  $\tilde{T}$  in  $O(nk)$  time, i.e. linear to the size of the input. In particular, we used the novel technique of substituting the non-solid symbols in the given degenerate strings with unique solid symbols, which let us make use of the efficient approximate pattern search solution for solid strings to get an efficient solution for degenerate strings. It would be interesting to see how well the presented algorithm behaves in practice and to apply it to solve a vast number of problems like prefix/border array, suffix trees, covers, repetitions, seeds, decomposition etc.

## References

- [1] Athel Cornish-Bowden. Iupac-iub symbols for nucleotide nomenclature. *Nucleic Acids Research*, 13:3021–3030, 1985.
- [2] Michael J Fischer and Michael S Paterson. String-matching and other products. Technical report, DTIC Document, 1974.
- [3] S Muthukrishnan and Krishna Palem. Non-standard stringology: Algorithms and complexity. In *Proceedings of the twenty-sixth annual ACM symposium on Theory of computing*, pages 770–779. ACM, 1994.

- [4] Piotr Indyk. Faster algorithms for string matching problems: Matching the convolution bound. In *Foundations of Computer Science, 1998. Proceedings. 39th Annual Symposium on*, pages 166–173. IEEE, 1998.
- [5] Adam Kalai. Efficient pattern-matching with don’t cares. In *SODA*, volume 2, pages 655–656, 2002.
- [6] Jan Holub, William F Smyth, and Shu Wang. Fast pattern-matching on indeterminate strings. *Journal of Discrete Algorithms*, 6(1):37–50, 2008.
- [7] William F Smyth and Shu Wang. An adaptive hybrid pattern-matching algorithm on indeterminate strings. *International Journal of Foundations of Computer Science*, 20(06):985–1004, 2009.
- [8] M. Sohel Rahman, Costas S. Iliopoulos, and Laurent Mouchard. Pattern matching in degenerate dna/rna sequences. In M. Kaykobad and Md. Saidur Rahman, editors, *Proceedings of the Workshop on Algorithms and Computation*, pages 109–120, Dhaka, Bangladesh, February 2007. Bangladesh Academy of Sciences. ISBN 984-300-000010-3.
- [9] Karl R. Abrahamson. Generalized string matching. *SIAM J. Comput.*, 16(6):1039–1051, 1987. doi: 10.1137/0216067. URL <http://dx.doi.org/10.1137/0216067>.
- [10] Maxime Crochemore, Costas S. Iliopoulos, Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, and Tomasz Walen. Covering problems for partial words and for indeterminate strings. In *ISSAC*, 2014.
- [11] G. M. Landau and U. Vishkin. Fast parallel and serial approximate string matching. *J. Algorithms*, 10(2):157–169, June 1989. ISSN 0196-6774. doi: 10.1016/0196-6774(89)90010-2. URL [http://dx.doi.org/10.1016/0196-6774\(89\)90010-2](http://dx.doi.org/10.1016/0196-6774(89)90010-2).
- [12] Peter Weiner. Linear pattern matching algorithms. In *Proceedings of the 14th Annual Symposium on Switching and Automata Theory (Swat 1973)*, SWAT ’73, pages 1–11, Washington, DC, USA, 1973. IEEE Computer Society. doi: 10.1109/SWAT.1973.13. URL <http://dx.doi.org/10.1109/SWAT.1973.13>.