Efficient Computation of Clustered-Clumps in Degenerate Strings

Costas Iliopoulos, Ritu Kundu, Manal Mohamed

Efficient Computation of Clustered-Clumps in Degenerate Strings *

Costas S. Iliopoulos, Ritu Kundu, and Manal Mohamed

Department of Informatics,
King's College London
London WC2R 2LS, United Kingdom
{costas.iliopoulos, ritu.kundu, manal.mohamed}@kcl.ac.uk

Abstract. Given a finite set of patterns, a clustered-clump is a maximal overlapping set of occurrences of such patterns. Several solutions have been presented for identifying clustered-clumps based on statistical, probabilistic, and most recently, formal language theory techniques. Here, motivated by applications in molecular biology and computer vision, we present efficient algorithms, using String Algorithm techniques, to identify clustered-clumps in a given text. The proposed algorithms compute in $\mathcal{O}(n+m)$ time the occurrences of all clustered-clumps for a given set of degenerate patterns $\tilde{\mathcal{P}}$ and/or degenerate text $\tilde{T}$ of total lengths m and n, respectively; such that the total number of non-solid symbols in $\tilde{\mathcal{P}}$ and $\tilde{T}$ is bounded by a fixed positive integer d.

Keywords: conservative degenerate string; pattern; overlapping occurrences; clustered-clump

1 Introduction

The ability to identify and compute various repeated patterns in strings is known to play a central role in many aspects of computer science fields including data compression, computer vision, computer-assisted music analysis and molecular biology. One of the most fundamental questions arising in such studies is to locate the *regions/windows* of overlapping occurrences of patterns in a given longer string named as text. This question is of particular interest in molecular biology, e.g. finding patterns with unexpectedly high or low frequencies and gene recognition.

In this paper, we consider a recently studied problem of computing clumps in a given text [10, 2]. In particular, given a finite set of patterns $\mathcal{P}$, we compute all factors in a given text T such that each factor is composed of the maximal overlapping occurrences of patterns from $\mathcal{P}$; these will be referred to as clustered-clumps hereafter. Such findings may be utilised, for example, for gene prediction, that is to find genes within a genome, based on the occurrences of specific DNA sequence motifs before or after them. Examples of such motifs include gene

* This research is partially supported by The Leverhulme Trust

promoters; start and stop codons; and poly(A) tails. An example of overlapping motifs, specifically, recognition sites to which proteins bind, is presented in [6].

In molecular biology (where sequences are considered as stings over fixed size alphabet Σ) if the specific nature of biological data is to be accommodated, it is required to allow some positions in the sequence to contain, instead of a single letter from Σ, a subset of Σ. Such *degenerate (indeterminate)* symbols can be interpreted as information that the exact letter at the given position is not known, but is suspected to be one of the specified letters.

Other than the aforementioned applications in genomics, identification of clustered-clumps in *degenerate* data also finds applications in areas such as computer vision or image processing. One such application can be the matching and retrieval of roughly aligned images containing the same scene, except nuances of some regions, and allowing for transformations like shifting, scaling, rotation etc.

Several solutions have been presented for identifying clustered-clumps based on statistical, probabilistic, and most recently, formal language theory techniques [10, 2]. To the best of our knowledge, no solution heretofore explores the problem accounting for *degeneracy* in data. Here, we present efficient algorithms, using String Algorithm techniques, to identify clustered-clumps in a given text. Our solution considers *degenerate* strings arising from the nature of real data. The proposed algorithms compute in $\mathcal{O}(n+m)$ time the occurrences of all clustered-clumps for a given set of degenerate patterns $\tilde{\mathcal{P}}$ and/or degenerate text $\tilde{T}$ of total lengths m and n, respectively; such that the total number of non-solid symbols in $\tilde{\mathcal{P}}$ and $\tilde{T}$ is bounded by a fixed positive integer d.

The rest of the paper is organised in the following format: The next section introduces the vocabulary and the notions that will be used throughout paper. The algorithmic tools and data-structures required to build the solutions have been described in Section 3. Section 4 formally defines the problem and its variations along with presenting and analysing the algorithms. Finally, Section 5 concludes the paper.

2 Terminology and Technical Background

We begin with basic definitions and notation. We think of a *string* X of *length* n as an array $X[1\,..\,n]$, where every $X[i]$, $1 \leq i \leq n$, is a *letter* drawn from some fixed *alphabet* Σ of size $|\Sigma| = \mathcal{O}(1)$. The *empty string* is denoted by ε. The set of all strings over Σ (including the empty string ε) is denoted by Σ^*. A string Y is a *factor* of a string X if there exist two strings U and V, such that $X = UYV$. Hence, we say that there is an *occurrence* of Y in X, or, simply, that Y *occurs in* X. Consider the strings X, Y, U, and V, such that $X = UYV$. If $U = \varepsilon$, then Y is a *prefix* of X. If $V = \varepsilon$, then Y is a *suffix* of X.

A *degenerate symbol* $\tilde{\sigma}$ over an alphabet Σ is a non-empty subset of Σ, i.e., $\tilde{\sigma} \subseteq \Sigma$ and $\tilde{\sigma} \neq \emptyset$. $|\tilde{\sigma}|$ denotes the size of the set and we have $1 \leq |\tilde{\sigma}| \leq |\Sigma|$. A *degenerate string* is built over the potential $2^{|\Sigma|} - 1$ non-empty sets of letters belonging to Σ. In other words, a degenerate string $\tilde{X} = \tilde{X}[1\,..\,n]$, is a

string such that every $\tilde{X}[i]$ is a degenerate symbol, $1 \leq i \leq n$. For example, $\tilde{X} = \{\texttt{a,b}\}\{\texttt{a}\}\{\texttt{c}\}\{\texttt{b,c}\}\{\texttt{a}\}\{\texttt{a,b,c}\}$ is a degenerate string of length 6 over $\Sigma = \{\texttt{a,b,c}\}$. If $|\tilde{X}[i]| = 1$, that is, $\tilde{X}[i]$ is a single letter of Σ, then 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 a *non-solid symbol* and a *non-solid position*, respectively. For convenience, we often write $\tilde{X}[i] = \sigma$ ($\sigma \in \Sigma$), instead of $\tilde{X}[i] = \{\sigma\}$ in case of solid symbols. Consequently, the degenerate string $\tilde{X}$ mentioned previously will be written as $\tilde{X} = \{\texttt{a,b}\}\texttt{ac}\{\texttt{b,c}\}\texttt{a}\{\texttt{a,b,c}\}$. A string containing only solid symbols will be called a *solid string*. A *conservative degenerate string* is a degenerate string where its number of non-solid symbols is upper-bounded by a fixed positive constant.

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 $\tilde{\alpha_1}$ and $\tilde{\alpha_2}$ are said to *match* (represented as $\tilde{\alpha_1} \approx \tilde{\alpha_2}$) if $\tilde{\alpha_1} \cap \tilde{\alpha_2} \neq \emptyset$. 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 $\tilde{X}[i] \approx \tilde{Y}[i]$, for $i = 1, \cdots, |\tilde{X}|$. Note that the relation $\approx$ is not transitive. A degenerate string $\tilde{Y}$ is said to *occur* at position i in another degenerate (resp. solid) string $\tilde{X}$ (resp. X) if $\tilde{Y} \approx \tilde{X}[i \,..\, i + |\tilde{Y}| - 1]$ (resp. $\tilde{Y} \approx X[i \,..\, i + |\tilde{Y}| - 1]$). Note that for a fixed-sized alphabet, the matching relation can be implemented in $\mathcal{O}(1)$ time if degenerate symbols are represented by bit-vectors of size $|\Sigma|$.

A set of strings $\mathcal{P} = \{P_1, \cdots, P_r\}$ is *reduced* if no P_i is factor of a P_j with $i \neq j$. For instance, the set $\{\texttt{aa,aba}\}$ is reduced whereas the sets $\{\texttt{aa,aab}\}$, $\{\texttt{aa,baa}\}$, and $\{\texttt{aa,baab}\}$ are non-reduced.
In [1], a *clustered-clump* of a given reduced set of strings $\mathcal{P} = \{P_1, \cdots, P_r\}$, where each P_i of length at least 2, is defined as follows:

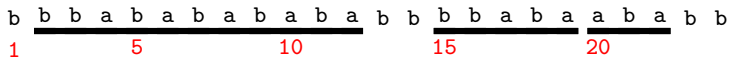
Definition 1 (Clustered-Clump). *A clustered-clump of a given reduced set of strings $\mathcal{P} = \{P_1, \cdots, P_r\}$ is a string W such that any two consecutive positions in W are covered by the same occurrence in W of a string $P \in \mathcal{P}$. The position i of the string W is covered by a string P if $P = W[j \,..\, j + |P| - 1]$ for some $j \in \{1, \cdots, |W| - |P| + 1\}$ and $j \leq i \leq j + |P| - 1$. More formally, W is a clustered-clump for the set $\mathcal{P}$ such that*

$$\forall i \in \{1, \cdots, |W|\} \;\; \exists P \in \mathcal{P}, \exists j \in Pos_W(P) \text{ such that } j \leq i \leq j + |P| - 1,$$

where $Pos_W(P)$ is the set of positions of occurrences of P in W.

For a given text (string) T, a factor W is a clustered-clump if it is *maximal* in the sense that there exists no occurrence of the set $\mathcal{P}$ in T that overlaps W without being a factor of it.

Example 1. Consider the set $\mathcal{P} = \{\texttt{aba}, \texttt{bba}\}$ and the text $T = \texttt{bbbabababababb}$ $\texttt{bbabaababb}$, we have the following clumps underlined:



Notice that the factor $\texttt{ababa}$ at position 6 is not a clustered-clump since it is not maximal. Also, the factor $\texttt{bbabaaba}$ at position 15 does not form a single

clustered-clump, because its two-letter factor `aa` is not covered by an occurrence of either `aba` or `bba`.

3 Algorithmic Tools

In the following we present two fundamental data structures supporting a wide variety of string matching algorithms. Both data structures are used in the proposed algorithms presented in Section 4.

Suffix Tree:

The *suffix tree* $\mathcal{S}(X)$ of a non-empty string X of length n is a compact trie representing all the suffixes of X such that $\mathcal{S}(X)$ has n leaves, labelled from 1 to n. Additionally, each edge is labelled with a factor of X. For any $i, 1 \leq i \leq n$, the concatenation of the edges' labels on the path from the root of $\mathcal{S}(X)$ to leaf i is precisely the suffix $X[i\,..\,n]$. For any two suffixes $U = X[i\,..\,n]$ and $V = X[j\,..\,n]$ of X, if W is the longest common prefix of U and V, then the path in $\mathcal{S}(X)$ corresponding to W is the same for U and V. For a general introduction of suffix trees, see [3].

The construction of the suffix tree $\mathcal{S}(X)$ of the input string X takes $\mathcal{O}(n)$ time and space, for string over a fixed-sized alphabet [12, 8, 11]. Once the suffix tree of a given string (called text) has been constructed, it can be used to support queries that return the occurrences of a given string (called pattern) in time linear in the length of the pattern.

Aho-Corasic Automaton:

The *Aho-Corasic automaton* of a set of strings $\mathcal{P}$, denoted $\mathcal{A}(\mathcal{P})$, is the minimal partial deterministic finite automaton accepting the set of all strings having a string of $\mathcal{P}$ as a suffix (see [5, Section 7.1] for more description and for efficient construction); an example is given in Figure 1. This data structure has an *initial* state, denoted s_0, and a *transition function* represented by the edges in the figure. A state is marked as terminal if the string it represents is in the set $\mathcal{P}$; note that all the leaves are terminal states. Let 'goto' denote the transition function, then the *suffix-link*, represented by the dotted line; is defined as follows: For a given non empty string X such that $s_i = \text{goto}(s_0, X)$, the suffix-link of state s_i points at $s_j = \text{goto}(s_0, X')$, where X' is the longest suffix of X such that $s_i \neq s_j$.

The construction of the suffix automaton $\mathcal{A}(\mathcal{P})$ together with the suffix-links can be done in linear time and space [3] independent of the alphabet size. Note that the transition function can be implemented in $\mathcal{O}(1)$ time for a fixed size alphabet.

Once the automaton $\mathcal{A}(\mathcal{P})$ has been constructed, searching a text T for occurrences of the patterns in $\mathcal{P}$ can be realized in time linear in the length of T; such a problem is known as the dictionary matching problem. The matching

involves the Aho-Corasic automaton scanning the text, reading every letter exactly once. If the automaton is in state s_i and reads letter α of the text, it moves to state $s_j = \text{goto}(s_i, \alpha)$ if defined, otherwise, it moves to the nearest s_k such that $s_k = \text{goto}(s_j, \alpha)$ is defined and s_j is the state identified by the following of suffix-links starting from s_i. Also, if the automaton encounters a terminal state, it outputs an occurrence(s) of one or more patterns. Note that if $\mathcal{P}$ is reduced then at most one pattern from $\mathcal{P}$ occurs at each position of the text. In the rest of the paper, we assume that $\mathcal{P}$ is a reduced set.

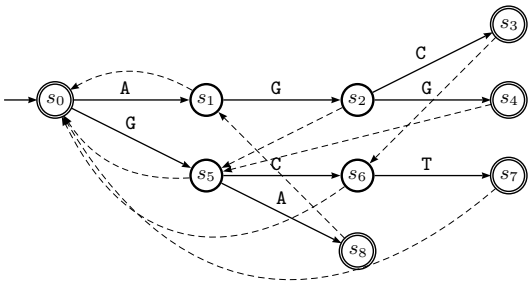


Fig. 1. Aho-Corasic automaton of set $\mathcal{P} = \{\texttt{AGC,AGG,GCT,GA}\}$.

4 Clustered-Clumps Algorithms

The Clustered-Clump problem is formally defined as follows:

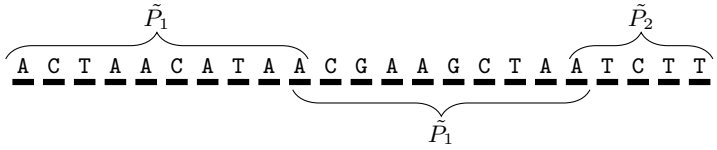
FINDING CLUSTERED-CLUMPS
Input: A text T of length n and a set of patterns $\mathcal{P} = \{P_1, \cdots, P_r\}$, such that $m = \sum_{1 \leq i \leq r} |P_i|$.
Output: All clustered-clumps in T.

While the above problem can efficiently be solved using any standard dictionary matching algorithm, extending the definition to include degenerate strings makes the problem more interesting, challenging, and useful for practical applications. In the following, we reformulate the definition to generate three variations of the problem - the patterns are degenerate (but the text is solid), the text is degenerate (but the patterns are solid), and both the text and the patterns are degenerate. Further, we assume that the number of non-solid symbols in either the text or the set of patterns is bounded by a given constant, i.e. we will deal with conservative degenerate strings.

4.1 Problem 1: Solid Text & Degenerate Patterns

PROBLEM 1: FINDING CLUSTERED-CLUMPS IN SOLID TEXT GIVEN DEGENERATE PATTERNS
Input: A text T of length n, a set of conservative degenerate patterns $\tilde{\mathcal{P}} = \{\tilde{P}_1, \cdots, \tilde{P}_r\}$, and integers d and m, such that total number of non-solid symbols in $\tilde{\mathcal{P}} \leq d$, and $m = \sum_{1 \leq i \leq r} |\tilde{P}_i|$.
Output: All clustered-clumps in T.

Example 2. Consider the text $T =$ `TTCGACTAACATAACGAAGCTAATCTTAAC` and the set of degenerate patterns $\tilde{\mathcal{P}}=$ {`AC{T,G}AA{C,G}{A,C,G}TAA`, `AT{C,G}TT`}, a clustered-clump occurring at position 5 has been shown below:



The solution we propose for this problem is based on the idea used in [9]. Each degenerate pattern $\tilde{P}_i \in \tilde{\mathcal{P}}$ can be seen as consisting of solid subpatterns, interspersed with *non-solid regions*. Let $P_{i,j}$ be a solid subpattern of $\tilde{P}_i$, $1 \leq i \leq r$, $1 \leq j \leq sub(i)$, where $sub(i)$ denotes the number of solid subpatterns in $\tilde{P}_i$. Additionally, let $\Re^i_{j-1,j}$ represents a *non-solid region* between the subpatterns $P_{i,j-1}$ and $P_{i,j}$. In other words, a pattern $\tilde{P}_i$ can be viewed as:

$$\tilde{P}_i = P_{i,1} \quad \Re^i_{1,2} \quad P_{i,2} \quad \Re^i_{2,3} \quad P_{i,3} \quad .. \quad P_{i,sub(i)}$$

Note that if the pattern $\tilde{P}_i$ ends with a non-solid symbol(s), then the last *non-solid region* is represented as $\Re^i_{sub(i),\infty}$.

The following steps outline our solution:
Step 1: Split: In this step, each degenerate pattern in $\tilde{\mathcal{P}}$ is split into its component subpatterns; we call the set of all these solid subpatterns so obtained $\mathcal{P}$. Effectively, we are breaking every degenerate pattern into subpatterns by chopping out *non-solid regions* so that each of the subpatterns is solid.

Example 3. Suppose $\tilde{\mathcal{P}}=$ {`AC{T,G}AA{C,G}{A,C,G}TAA`, `AT{C,G}TT`} as in Example 2. Here $r = 2$. Splitting $\tilde{P}_1$ results in $P_{1,1} =$ `AC`, $P_{1,2} =$ `AA`, $P_{1,3} =$ `TAA`, while splitting $\tilde{P}_2$ results in $P_{2,1} =$ `AT` and $P_{2,2} =$ `TT`. Then $\mathcal{P} =$ {`AC,AA,TAA,AT,TT`}, $sub(1) = 3$, and $sub(2) = 2$. Here, $\Re^1_{1,2} =$ {`T,G`}, $\Re^1_{2,3} =$ {`C,G`}{`A,C,G`}, and $\Re^2_{1,2}$ = {`C,G`}.

Step 2: Find occurrences of solid subpatterns in T: We next build the Aho-Corasick automaton of the set $\mathcal{P}$; denoted by $\mathcal{A}(\mathcal{P})$. Using the automaton, we compute all the occurrences of the solid subpatterns in the text T.

The occurrences of the subpatterns of the set $\mathcal{P}$ are maintained using a boolean matrix *Valid* of size $|\mathcal{P}| \times n$ such that we can test in constant time whether or not a specific solid subpattern occurs at a given text position. If an occurrence of $P_{i,j}$ for which $\Re^i_{j-1,j}$ exists is found at a position (say k), then we need to check:

1. Whether $P_{i,j-1}$ (if $j > 1$) occurs in the corresponding position (i.e. $k - (|P_{i,j-1}| + |\,\Re^i_{j-1,j}\,|)$ in T.
2. Whether the non-solid symbols in $\Re^i_{j-1,j}$ match the corresponding positions in T. If $j = sub(i)$, then the non-solid region $\Re^i_{sub(i),\infty}$ is also tested for matching.

If both conditions are true, then an occurrence of $P_{i,j}$ is marked `true` in the matrix *Valid*. Notice that proceeding in this way, an occurrence marked `true` for $P_{i,sub(i)}$ corresponds to an occurrence of the degenerate pattern $\tilde{P}_i$ in T.

Step 3: Compute the locations of clustered-clumps: Using the information about the occurrences of the degenerate patterns in the text, we populate an array *LongestOcc* of size n that stores the length of the longest pattern occurring at each position of the text. It is easy to see that simple calculations done in a single scan of this array can report the positions of all the clustered-clumps in T; see Function 1 below for more details.

Function 1 `ComputeClusteredClumps(`*LongestOcc*, n`)`

input : *LongestOcc* is the array storing the length of the longest pattern occurring at each position of the text
integer n represents length of the text.

output: A set of all pairs (i, l) such that a maximal clustered clump of length l occurs at position i in the text.

```
R ← Φ;
start ← last ← 1;
for u ← 1 to n do                                   // Scan LongestOcc
    if LongestOcc[u] = 0 then
        if last < u then
            start ← last ← u + 1;
    else if u + LongestOcc[u] − 1 > last then
        last ← u + LongestOcc[u] − 1;
    if u = last then
        if last − start > 0 then
            Add (start, last − start + 1) to R;
        start ← last ← last + 1;
return R;
```

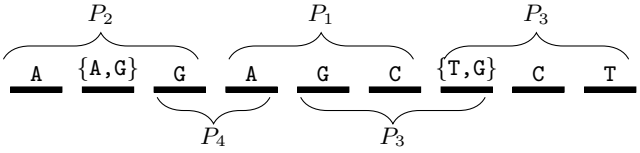
Running Time Analysis: Computing both $\mathcal{P}$ and $\mathcal{A}(\mathcal{P})$ takes $\mathcal{O}(m)$ time, while $\mathcal{O}(n)$ time is required for finding all the occurrences of all the solid subpat-

terns using the Aho-Corasick automaton. Subsequent symbol-by-symbol matching of *non-solid regions* is bounded by $\mathcal{O}(d)$ in the worst case for each position in the text, implying that overall $\mathcal{O}(dn)$ time is required. Computations in the last step can be done in $\mathcal{O}(n)$ time. Thus, the solution finds all the clustered-clumps in the text in $\mathcal{O}(n+m)$ time for constant d.

4.2 Problem 2: Degenerate Text & Solid Patterns

PROBLEM 2: FINDING CLUSTERED-CLUMPS IN DEGENERATE TEXT GIVEN SOLID PATTERNS
Input: A conservative degenerate text $\tilde{T}$ of length n, a set of patterns $\mathcal{P} = \{P_1, \cdots, P_r\}$, and integers d and m, such that the total number of non-solid symbols in $\tilde{T} \leq d$, and $m = \sum_{1 \leq i \leq r} |P_i|$.
Output: All clustered-clumps in $\tilde{T}$.

Example 4. Consider the text $T =$ `CATTA{A,G}GAGC{T,G}CTTTA` and the set of patterns $\tilde{\mathcal{P}}=$ {`AGC,AGG,GCT,GA`}, a clustered-clump occurring at position 5 has been shown below:



Our solution for Problem 2 is built using the recently developed algorithm described in [4] that solves, in linear time, pattern matching problem in conservative degenerate text (applying an adapted version of Landau and Vishkin's [7] algorithm for approximate pattern matching). Please refer to [4] for full details, but for the sake of completeness, a brief description has been provided in the following steps:
Step1: Substitute: In this step, each of the non-solid symbols occurring in the given degenerate text is replaced by a unique symbol which is not present in Σ. Let Λ be the set of these unique symbols i.e. $\Lambda = \{\lambda_i\}$ such that $0 < i \leq d$. It is to be noted that the text, T_λ, obtained by such a substitution will be a solid string. For example, if $T =$ `CATTA{A,G}GAGC{T,G}CTTTA` as in Example 4 then $T_\lambda =$ `CATTA`λ_1`GAGC`λ_2`CTTTA`; here $\Lambda = \{\lambda_1, \lambda_2\}$.
Step 2: Find occurrences of solid patterns in T_λ: We concatenate the text T_λ and the patterns as follows:

$$\overline{T} = T_\lambda P_1 \#_1 .. P_r \#_r,$$

where each delimiting symbol $\#_i$, $1 \leq i \leq r$ is a unique symbol that is not present in $\Sigma \cup \Lambda$. Next, the suffix tree $\mathcal{S}(\overline{T})$ of $\overline{T}$ is constructed. As detailed in

[4], checking whether a pattern P_i occurs at a certain position in T is realized by at most d longest common ancestor (LCA) queries on $\mathcal{S}(\overline{T})$.
Step 3: Compute the positions of clustered-clumps: We proceed in similar fashion as in Step 3 of Problem 1, reporting the positions of all clustered-clumps in $\tilde{T}$ using Function 1.

Running Time Analysis: The substitution and the concatenation steps to obtain T_λ and $\overline{T}$, and later constructing the suffix tree $\mathcal{S}(\overline{T})$ can be performed in $\mathcal{O}(n+m)$ time. Finding all the occurrences of all the patterns in $\mathcal{P}$ takes $\mathcal{O}(dn)$ [4]. Thus, for constant d, the solution computes all the clustered-clumps in the text in $\mathcal{O}(n+m)$ time.

4.3 Case 3: Degenerate Text & Degenerate Patterns

Problem 3: Finding Clustered-Clumps in Degenerate Text given Degenerate Patterns
Input: A conservative degenerate text $\tilde{T}$ of length n, a set of conservative degenerate patterns $\tilde{\mathcal{P}} = \{\tilde{P}_1, \cdots, \tilde{P}_r\}$, and integers d and m, such that total number of non-solid symbols in both $\tilde{T}$ and $\tilde{\mathcal{P}} \leq d$ and $m = \sum_{1 \leq i \leq r} |\tilde{P}_i|$.
Output: All clustered-clumps in $\tilde{T}$.

The solution for Problem 3 is achieved by combining the solutions for both Problem 1 and Problem 2. Let d_T and $d_{\mathcal{P}}$ be the number of non-solid symbols in $\tilde{T}$ and $\tilde{\mathcal{P}}$, respectively, such that $d = d_T + d_{\mathcal{P}}$. In the following, we outline our solution for this general case:
Step1: Substitute: Replace each non-solid symbol in $\tilde{T}$ with a unique symbol λ_i; $0 < i \leq d_T$, to obtain a solid text T_λ.
Step 2: Split: Split each of the degenerate patterns in $\tilde{\mathcal{P}}$ into its component subpatterns to obtain the set of solid subpatterns $\mathcal{P}$.
Step 3: Find occurrences of solid subpatterns: Construct the suffix tree $(\mathcal{S}(\overline{T}))$ for $\overline{T}$ (string obtained by appending T_λ with all the subpatterns in $\mathcal{P}$ delimited by unique symbols $\#_i$, $1 \leq i \leq d_{\mathcal{P}} + r$). Find all the occurrences of each of the subpatterns in $\mathcal{P}$ by LCA queries on $\mathcal{S}(\overline{T})$ (as in Step 2 of Problem 2). Similar to Step 2 in Problem 1, maintain a boolean matrix *Valid* such that an occurrence marked `true` for $P_{i,sub(i)}$ corresponds to an occurrence of the degenerate pattern $\tilde{P}_i$ in $\tilde{T}$.
Step 4: Compute the locations of clustered-clumps: Use Function 1 to report the positions of all clustered-clumps in $\tilde{T}$.

Running Time Analysis: As the solution makes use of the steps of the solutions for problems 1 and 2, both of which have been shown to be bounded by $\mathcal{O}(n+m)$ time, thus, Problem 3 can overall be solved in $\mathcal{O}(n+m)$ time.

5 Conclusion

In this paper, we studied the problem of identifying clustered-clumps in conservative degenerate strings and presented $\mathcal{O}(n+m)$-time algorithms that compute the occurrences of all clustered-clumps for a given set of degenerate patterns $\tilde{\mathcal{P}}$ or/and a degenerate text $\tilde{T}$ of total lengths m and n, respectively; such that the total number of non-solid symbols in $\tilde{\mathcal{P}}$ and $\tilde{T}$ is bounded by a given constant d. The presented algorithms are promising for applications in genomics and computer vision. We intend to conduct larger-scale experiments, using genomic as well as digitized-images datasets. Furthermore, other domains that involve web-mining applications may find the presented solutions interesting and beneficial.

References

1. Bassino, F., Clément, J., Fayolle, J., Nicodème, P.: Constructions for clumps statistics. CoRR abs/0804.3671 (2008)
2. Boeva, V., Clment, J., Rgnier, M., Vandenbogaert, M.: Assessing the significance of sets of words. In: Apostolico, A., Crochemore, M., Park, K. (eds.) CPM. Lecture Notes in Computer Science, vol. 3537, pp. 358–370. Springer (2005)
3. Crochemore, M., Hancart, C., Lecroq, T.: Algorithms on Strings. Cambridge University Press (2007), 392 pages
4. Crochemore, M., Iliopoulos, C.S., Kundu, R., Mohamed, M., Vayani, F.: Linear algorithm for conservative degenerate pattern matching. Engineering Applications of Artificial Intelligence 51, 109 – 114 (2016)
5. Crochemore, M., Rytter, W.: Text Algorithms. Oxford University Press (1994)
6. Kvietikova, I., Wenger, R.H., Marti, H.H., Gassmann, M.: The transcription factors ATF-1 and CREB-1 bind constitutively to the hypoxia-inducible factor-1 (HIF-1) DNA recognition site. Nucleic Acids Research 23(22), 4542–4550 (1995)
7. Landau, G.M., Vishkin, U.: Fast parallel and serial approximate string matching. J. Algorithms 10(2), 157–169 (Jun 1989)
8. McCreight, E.M.: A space-economical suffix tree construction algorithm. Journal of the ACM (JACM) 23(2), 262–272 (1976)
9. Rahman, M.S., Iliopoulos, C.S.: Pattern matching algorithms with don't cares. In: van Leeuwen, J., Italiano, G.F., van der Hoek, W., Meinel, C., Sack, H., Plasil, F., Bielikova, M. (eds.) Proceedings of the 33rd International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM07). pp. 116–126. Institute of Computer Science AS CR, Prague (2007)
10. Régnier, M.: A unified approach to word statistics. In: Proceedings of the Second Annual International Conference on Computational Molecular Biology. pp. 207–213. RECOMB, ACM, New York, USA (1998)
11. Ukkonen, E.: On-line construction of suffix trees. Algorithmica 14(3), 249–260 (1995)
12. Weiner, P.: Linear pattern matching algorithms. In: Proceedings of the 14th IEEE Annual Symposium on Switching and Automata Theory. pp. 1–11. Institute of Electrical Electronics Engineer (1973)