String Regularities and Degenerate Strings

M. Sc. Thesis Defense

Md. Faizul Bari (100705050P)
Supervisor: Dr. M. Sohel Rahman

Department of Computer Science and Engineering
Bangladesh University of Engineering and Technology

Overview

- Problem Definition
- Basic Concepts
- Present State of the Problem
- Our Contributions
- Performance Comparison
- Motivation and Importance
- Conclusion

Overview

- Problem Definition
- Basic Concepts
- Present State of the Problem
- Our Contributions
- Performance Comparison
- Motivation and Importance
- Conclusion

Problem Definition

- The objective of this research is to devise novel algorithms for computing different kinds of regularities for degenerate strings.
- We mainly focus on computing the following data structures which contain information about repeated patterns in a string
  - Border array
  - Prefix array
  - Cover array

Problem Definition

- We are given a degenerate string x, of length n. We need to solve the following problems:
  - *Problem 1*: Computing the **prefix array** of x
  - *Problem 2*: Computing the **border array** of x
  - *Problem 3*: Computing the **cover array** of x

Overview

- Problem Definition
- Basic Concepts
- Present State of the Problem
- Our Contributions
- Performance Comparison
- Motivation and Importance
- Conclusion

Basic Concepts

- For a non-empty string, x = abbaccbbabbca

x =

a	b	b	a	c	c	b	b	a	b	b	c	a
1	2	3	4	5	6	7	8	9	10	11	12	13

  - **Length** of x is denoted by, $|x| = 13$
  - The ***i*-th symbol** of x is $x[i]$
    - e.g. here $x[5]$ = c and $x[9]$ = a

Basic Concepts

$$\overbrace{\text{abb}\underbrace{\text{accbbab}}_{w}\text{bca}}^{x} \qquad w = \text{accbbab}$$

- w is a **substring** of x and x is a **superstring** of w.

$$\overbrace{\underbrace{\text{abbac}}_{u}\text{cb}\underbrace{\text{babbca}}_{v}}^{x} \qquad \begin{array}{l} u = \text{abbac} \\ v = \text{babbca} \end{array}$$

- u is a **prefix** and v is a **suffix** of x.

Basic Concepts

$x =$

a	b	b	a	c	c	b	b	a	b	b	c	A
1	2	3	4	5	6	7	8	9	10	11	12	13

w

- Here $w = x[4...10]$
- So, $x[i...j]$ denotes the **substring** of x starting at position i and ending at j

Basic Concepts

- Given two strings x and y

$$x = \text{abbacaabc} \qquad\qquad y = \text{ccbabbcab}$$

$$xy = \text{abbacaabcccbabbcab}$$

- xy is called the **concatenation** of x and y.

- x^k denotes the concatenation of k copies of x.

Basic Concepts

- Given two strings x and y

$$x = \text{abbacaabc} \qquad\qquad y = \text{aabcbbcab}$$

- Where x has a suffix equal to a prefix of y we can get a new string by ovelapping x and y.

$$x \textit{ overlaps } y = \text{abbacaabcbbcab}$$

- This is called **superposition** of x and y.

Basic Concepts

- Border of x

$$x = \text{aabcabccbbacaabc}$$

  - Here “aabc” is a border of x, as it is both a prefix and a suffix of x.

- The border array, β of x is an array such that
  - for all $i \in \{1\ldots n\}$, $\beta[i]$ = length of the **longest proper border of** $x[1\ldots i]$.

Basic Concepts

- Cover of x

concatenation

x = aabaabaaaabaabaa

aabaa aabaa

aabaa aabaa

superposition

w = aabaa

- A substring w of x is a cover of x, if x can be constructed by **concatenation** or **superposition** of w.

Basic Concepts

- The Cover Array, γ of x, is a data structure used to store the length of the **longest proper cover of every prefix of *x*;**

- That is for all $i \in \{1...n\}$, $\gamma[i]$ = length of the longest proper cover of $x[1...i]$ or 0.

Basic Concepts

- The prefix array, Π of x , is a data structure used to store the length of the **longest prefix of every prefix of *x*;**

- That is for all for all $i \in \{1 \ldots n\}$, $\Pi[i]$ = length of the longest prefix of $x[1 \ldots i]$ or 0.

Example of prefix, border and cover arrays

Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
$x =$	a	b	a	a	b	a	b	a	a	b	a	a	b	a	b	a	a	b	a	b	a	b	a
$\Pi =$	0	0	1	3	0	6	0	1	8	0	1	3	0	8	0	1	3	0	3	0	3	0	1
$\beta =$	0	0	1	1	2	3	2	3	4	5	6	4	5	6	7	8	9	10	11	7	8	2	3
$\gamma =$	0	0	0	0	0	3	0	3	0	5	6	0	5	6	0	8	9	10	11	0	8	0	3

Mathematical representation

- For every prefix x[1 … i] of x the following sequences are monotonically decreasing to zero.
  - $\Pi[i], \Pi^2[i], \Pi^3[i], \ldots, \Pi^m[i]$; here $\Pi^m[i] = 0$
  - $\beta[i], \beta^2[i], \beta^3[i], \ldots, \beta^m[i]$; here $\beta^m[i] = 0$
  - $\gamma[i], \gamma^2[i], \gamma^3[i], \ldots, \gamma^m[i]$; here $\gamma^m[i] = 0$

Basic Concepts

Degenerate Strings:

- A degenerate string is a sequence

$T = T[1]T[2]...T[n]$, where $T[i] \subseteq \Sigma$ for all i, and Σ is a given alphabet of fixed size.

- If at any position in a degenerate string,

$|T[i]| = 1$, we call this a **solid symbol.** However, when $|T[i]| \geq 2$, we call this a **non-solid symbol.**

Basic Concepts

- Degenerate Strings:

$$
\begin{array}{r@{\,}cccccccccccccccc}
 & & & \mathrm{a} & & \mathrm{a} & & & & & & & & & \mathrm{a} & & \mathrm{b} \\
x = & \mathrm{a} & \mathrm{a} & \mathrm{b} & \mathrm{a} & \mathrm{c} & \mathrm{b} & \mathrm{c} & \mathrm{a} & \mathrm{a} & \mathrm{a} & \mathrm{b} & \mathrm{a} & \mathrm{c} & \mathrm{b} & \mathrm{a} & \mathrm{c} \\
 & & & \mathrm{c} & & & & & & & \mathrm{c} & & & & \mathrm{c} & &
\end{array}
$$

$$x = \mathrm{aa[abc]a[ac]bcaa[ac]bac[abc]a[bc]}$$

Basic Concepts

Matching in degenerate strings

- Given a degenerate string x, we say that
  - $x[i]$ matches $x[j]$ iff $x[i] \cap x[j] \neq \varphi$
  - $x[i]$ exactly matches $x[j]$ iff $x[i]$ and $x[j]$ are exactly equal.
  - Here $x[i], x[j] \subseteq \Sigma$

Example of prefix, border and cover arrays

Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
$x =$	a	b	a	a	b	a	b	a	a	b	a	a	b	a	b	a	a	b	a	b	a	b	a
$\Pi =$	0	0	1	3	0	6	0	1	8	0	1	3	0	8	0	1	3	0	3	0	3	0	1
$\beta =$	0	0	1	1	2	3	2	3	4	5	6	4	5	6	7	8	9	10	11	7	8	2	3
$\gamma =$	0	0	0	0	0	3	0	3	0	5	6	0	5	6	0	8	9	10	11	0	8	0	3

Mathematical representation

- For every prefix x[1 … i] of x the following sequences are monotonically decreasing to zero.
  - $\Pi[i], \Pi^2[i], \Pi^3[i], \ldots, \Pi^m[i]$; here $\Pi^m[i] = 0$
  - $\beta[i], \beta^2[i], \beta^3[i], \ldots, \beta^m[i]$; here $\beta^m[i] = 0$
  - $\gamma[i], \gamma^2[i], \gamma^3[i], \ldots, \gamma^m[i]$; here $\gamma^m[i] = 0$

In case of degenerate string

- These sequences in not valid for degenerate string.

- This can be easily shown by an example.

Border array of a degenerate string

Index	1	2	3	4	5
$x =$	a	b	a	[ab]	b
$\beta =$	0	0	1	2	2
				1	

Border and cover array of a degenerate string

Index	1	2	3	4	5	6
$x =$	a	b	a	[ab]	[ab]	a
$\beta =$	0	0	1	2	3	4
				1	2	3
					1	1
$\gamma =$	0	0	0	2	3	4
					2	3

Prefix array of a degenerate string

Index	1	2	3	4	5
$x =$	a	b	a	[ab]	b
$\Pi =$	0	0	2	2	0
$\beta =$	0	0	1	2	2
				1	

For a degenerate string

- Prefix array is linear in the size of *x*.
- Border and cover arrays can't be represented by a linear array. Both of them must be arrays of lists.
- The worst case space requirement for border and cover array in $O(n^2)$ where n is the length of x.

Overview

- Problem Definition
- Basic Concepts
- Present State of the Problem
- Our Contributions
- Performance Comparison
- Motivation and Importance
- Conclusion

Present State of the Problem

Regularities of conservative degenerate strings

- In a conservative degenerate string the number non-solid positions is bounded by a constant, λ.
- In [1], the authors investigated the regularities of conservative degenerate strings.
- The authors presented a $O(n\lambda)$ algorithms for finding
  - conservative covers (of length λ).
  - conservative seeds (of length λ).

Present State of the Problem

Regularities of conservative degenerate strings

- This algorithm can be extended to compute the cover array.
- But then we will have to run the algorithm for all possible cover lengths for every prefix of x.
- This would require $O(n^3)$ time and $O(n^2)$ space.

Present State of the Problem

Regularities on degenerate strings

- Antoniou et al. presented an $O(n \log n)$ algorithm to find the smallest cover of a degenerate string in [2].

- They showed that their algorithm can be easily extended to compute all the covers of x. The later algorithm runs in $O(n^2 \log n)$ time.

Present State of the Problem

Regularities on degenerate strings

- Antoniou's algorithm in [2], can also be extended to compute the cover array of x.
- This algorithm will also run in $O(n^2 \log n)$ time.
- This algorithm used uses a complex data structure , called the vEB tree.

Overview

- Problem Definition
- Basic Concepts
- Present State of the Problem
- Our Contributions
- Performance Comparison
- Motivation and Importance
- Conclusion

Our Contribution

- In this research we have devised the following new algorithms for degenerate strings:
    - *iCAb*: It uses border array and Aho-Corasick Automaton for computing all covers and the cover array.
    - *iCAp*: This algorithm computes the cover array from the prefix and border array of x.

The iCAb Algorithm

iCAb

- Finds all covers and the cover array of *x* using border array.
  - *Step 1:* Compute the border array of *x*.
  - *Step 2:* Using the Aho-Corasick pattern matching machine find out the borders that are also covers.

iCAb (STEP 1)

x = aa[abc]a[ac]bcaa[ac]bac[abc]a[bc]

Computer the border array of x

β	$(1,a)$	$(2,a)$		$(1,a)$	$(2,a)$	$(3,a)$	$(4,b)$	$(5,a)$			
		$(1,a)$			$(1,a)$	$(2,a)$	$(3,a)$				
						$(1,a)$	$(2,a)$	$(3,a)$	$(4,b)$	$(5,a)$	$(6,*)$
							$(1,a)$	$(2,a)$			
								$(1,a)$	$(2,a)$		
									$(1,a)$		
										$(1,a)$	$(2,a)$
											$(1,a)$

iCAb (STEP 2)

β	$(1,a)$	$(2,a)$		$(1,a)$	$(2,a)$	$(3,a)$	$(4,b)$	$(5,a)$			
		$(1,a)$			$(1,a)$	$(2,a)$	$(3,a)$				
						$(1,a)$	$(2,a)$	$(3,a)$	$(4,b)$	$(5,a)$	$(6,*)$
							$(1,a)$	$(2,a)$			
								$(1,a)$	$(2,a)$		
									$(1,a)$		
										$(1,a)$	$(2,a)$
											$(1,a)$

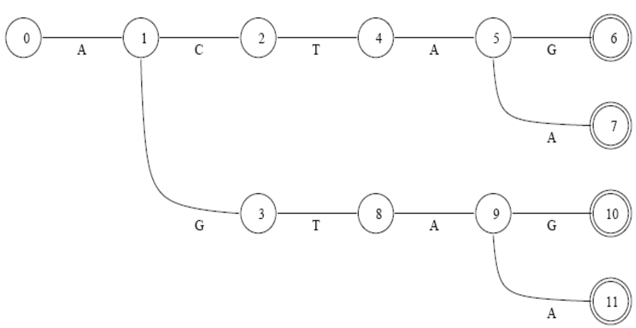
β	$(1,a)$	$(2,a)$		$(1,a)$	$(2,a)$	$(3,a)$	$(4,b)$	$(5,a)$			
		$(1,a)$			$(1,a)$	$(2,a)$	$(3,a)$				
						$(1,a)$	$(2,a)$	$(3,a)$	$(4,b)$	$(5,a)$	$(6,*)$
							$(1,a)$	$(2,a)$			
								$(1,a)$	$(2,a)$		
									$(1,a)$		
										$(1,a)$	$(2,a)$
											$(1,a)$

For Computing all the cover of x we only need the last entries of the border array.

iCAb (STEP 2)

β	$(1,a)$	$(2,a)$		$(1,a)$	$(2,a)$	$(3,a)$	$(4,b)$	$(5,a)$			
		$(1,a)$			$(1,a)$	$(2,a)$	$(3,a)$				
						$(1,a)$	$(2,a)$	$(3,a)$	$(4,b)$	$(5,a)$	$(6,*)$
							$(1,a)$	$(2,a)$			
								$(1,a)$	$(2,a)$		
									$(1,a)$		
										$(1,a)$	$(2,a)$
											$(1,a)$

Build an Aho-Corasick automaton with the dictionary containing the selected borders.



i	0	1	2	3	4	5	6	7	8	9	10	11
f(i)	0	0	0	0	0	1	3	1	0	1	3	1

Parse x through it to find out the borders that covers x.

iCAb (STEP 2)

β	$(1,a)$	$(2,a)$		$(1,a)$	$(2,a)$	$(3,a)$	$(4,b)$	$(5,a)$			
		$(1,a)$			$(1,a)$	$(2,a)$	$(3,a)$				
						$(1,a)$	$(2,a)$	$(3,a)$	$(4,b)$	$(5,a)$	$(6,*)$
							$(1,a)$	$(2,a)$			
								$(1,a)$	$(2,a)$		
									$(1,a)$		
										$(1,a)$	$(2,a)$
											$(1,a)$

β	$(1,a)$	$(2,a)$		$(1,a)$	$(2,a)$	$(3,a)$	$(4,b)$	$(5,a)$			
		$(1,a)$			$(1,a)$	$(2,a)$	$(3,a)$				
						$(1,a)$	$(2,a)$	$(3,a)$	$(4,b)$	$(5,a)$	$(6,*)$
							$(1,a)$	$(2,a)$			
								$(1,a)$	$(2,a)$		
									$(1,a)$		
										$(1,a)$	$(2,a)$
											$(1,a)$

For Computing the cover array of x we need to process all the entries of the border array.

iCAb (STEP 2)

β

$(1,a)\ (2,a)\quad (1,a)\ (2,a)\ (3,a)\ (4,b)\ (5,a)$

$(1,a)\quad (1,a)\ (2,a)\ (3,a)$

$(1,a)\ (2,a)\ (3,a)\ (4,b)\ (5,a)\ (6,*)$

$(1,a)\ (2,a)$

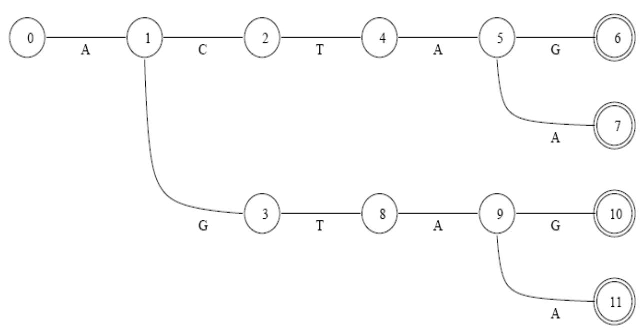
$(1,a)\ (2,a)$

$(1,a)$

$(1,a)\ (2,a)$

$(1,a)$

Build an Aho-Corasick automaton with the dictionary containing the selected borders.



i	0	1	2	3	4	5	6	7	8	9	10	11
f(i)	0	0	0	0	0	1	3	1	0	1	3	1

Parse x through it to find out the covers of x.

iCAb [Running Time Analysis]

- The algorithm runs in $O(nm)$ time where n is length of x and m is the number of borders.

- Using string combinatorics and probability analysis it can be proved that, *the expected number of borders of an degenerate string is bounded by a constant*.

iCAb [Running Time Analysis]

The possible equality cases are:

Symbol	Match To	Number of cases
$\sigma \in \{1, 2, \ldots, \alpha\}$	$\sigma \in \{1, 2, \ldots, \alpha\}$	α
$\sigma \in S,\ S \subseteq \Sigma$	$\sigma \in S,\ S \subseteq \Sigma,\ \lvert S\rvert > 1$	$\sum_{j=1}^{\alpha} \binom{\alpha}{j} \{2^{\alpha-j}(2^j - 1) - \binom{j}{1}\}$

Expected number of borders:

$$\sum_{k=1}^{n-1} \left(\frac{\alpha + \sum_{j=1}^{\alpha} \binom{\alpha}{j} \{2^{\alpha-j}(2^j - 1) - \binom{j}{1}\}}{2^{2\alpha}} \right)^k \leq \; 29.1746$$

So the running time reduces to *O(n)* on average.

iCAb

Finding all covers of an indeterminate string in $O(n)$ time on average

Md. Faizul Bari, M. Sohel Rahman, and Rifat Shahriyar

Department of Computer Science and Engineering
Bangladesh University of Engineering and Technology
Dhaka, Bangladesh
{faizulbari, msrahman, rifat}@cse.buet.ac.bd

Abstract. We study the problem of finding all the covers of an indeterminate string. An indeterminate string is a sequence $T = T[1]T[2]\dots T[n]$, where $T[i] \subseteq \Sigma$ for each i, and Σ is a given alphabet of fixed size. Here we describe an algorithm for finding all the covers of a string x. The algorithm is applicable for both regular and indeterminate strings. Our algorithm starts with the border array and uses pattern matching technique of the Aho-Corasick Automaton to compute all the covers of x from the border array. On average the algorithm requires $O(n)$ time to find out all the covers, where n is the length of x. finally, we extend our algorithm to compute the cover array of x in $O(n^2)$ time and $O(n)$ space complexity.

Key words: Indeterminate Strings, Covers, Cover Array, Aho-Corasick Automaton, String Regularities.

1 Introduction

Characterizing and finding regularities in strings are important problems in many areas of science. In molecular biology, repetitive elements in chromosomes determine the likelihood of certain diseases. In probability theory, regularities are important in the analysis of stochastic processes. In computer science, repetitive elements in strings are important in e.g. data compression, computational music analysis, coding, automata and formal language theory. As a result, in the last 20 years, string regularities have drawn a lot of attention from different disciplines of science.

The most common forms of string regularities are periods and repeats and there are several $O(n \log n)$ time algorithms for finding repetitions [6, 8], in a string x, where n is the length of x. Apostolico and Breslauer [4] gave an optimal $O(\log \log n)$-time parallel algorithm for finding all the repetitions of a string of length n. The preprocessing of the Knuth-Morris-Pratt algorithm [14] finds all periods of every prefix of x in linear time.

In many cases, it is desirable to relax the meaning of repetition. For instance, if we allow overlapping and concatenations of periods in a string we get the notion of *covers*. After periods and repeats, cover is the most popular form of regularities in strings. The idea of cover generalizes the idea of periods or repeats. A substring c of a string x is called a *cover* of x if and only if x can be constructed by concatenation and superposition of c. Another common string regularity is the *seed* of a string. A seed is an extended cover in the sense that it is a cover of a superstring of x.

Clearly, x is always a cover of itself. If a proper substring pc of x is also a cover of x, then pc is called a *proper cover* of x. For example, the string $x = abcababcabcabcab$ has covers x and $abcab$. Here, $abcab$ is a proper cover. A string that has a proper cover is called *coverable*; otherwise it is *superprimitive*. The notion of covers was introduced

- This algorithm was recently published in *The Prague Stringology Conference, 2009.*

The iCAp Algorithm

iCAp

- Step1: Finds the prefix array of x.

index	1	2	3	4	5	6	7	8
x	a	[ab]	b	b	a	[ab]	b	a
Π	0	3	0	0	3	2	0	1

  - *The prefix array contains non zero value only at positions which are equal to x[1]. First we find all such positions.*

  - *Then we try to extend each non-zero entry as far as possible*

iCAp

- For regular strings, there are several O(n) algorithm from computing the prefix array.
- But they all depend on the transitivity of matching.
- Degenerate string matching is non-transitive.
- So, no O(n) algorithm is possible for degenerate strings; as we have to match all possible pair of positions separately.
- This step requires $O(n^2)$ time and O(n) space.

iCAp

- Step 2: the prefix array is preprocessed so that the *range maxima queries* can be answered on this array in constant time per query.

- The preprocess in this step requires $O(n)$ time.

- So the running time of Step 2 is $O(n)$.

iCAp

- Step3: Finds the border array from the prefix array of x.

index	1	2	3	4	5	6	7	8
x	a	[ab]	b	b	a	[ab]	b	a
Π	0	3	0	0	3	2	0	1
β	0	1	2	3	1	2	3	1
						1	2	

Prefix array of a degenerate string

Index	1	2	3	4	5
$x =$	a	b	a	[ab]	b
$\Pi =$	0	0	2	2	0
$\beta =$	0	0	1	2	2
				1	

iCAp

- Step 3: Finds the border array from the prefix array of x.

index	1	2	3	4	5	6	7	8
x	a	[ab]	b	b	a	[ab]	b	a
Π	0	3	0	0	3	2	0	1
β	0	1	2	3	1	2	3	1
						1	2	

The border array can be computed from the prefix array. But the time and space complexity for computing and storing the border array is $O(n^2)$ in the worst case

iCAp

- Step3: Finds the cover array from the border and prefix array of *x*.

index	1	2	3	4	5	6	7	8
x	a	[ab]	b	b	a	[ab]	b	a
Π	0	3	0	0	3	2	0	1
β	0	1	2	3	1	2	3	1
						1	2	
γ	0	1	2	3	0	0	3	0

iCAp

- Now suppose string y is covered by the string *aba*.

index	1	2	3	4	5	6	7	8	9	10
	a	b	a		a	b	a			
y	a	b	a	b	a	b	a	a	b	a
			a	b	a			a	b	a
Π	0	0	5	0	3	0	1	3	0	1

iCAp

index	1	2	3	4	5	6	7	8
x	a	[ab]	b	b	a	[ab]	b	a
Π	0	3	0	0	3	2	0	1
β	0	1	2	3	1	2	3	1
						1	2	
γ	0	1	2	3	0	0	3	0

index	1	2	3	4	5	6	7	8
	a	[ab]	b		a	[ab]	b	
x	a	[ab]	b	b	a	[ab]	b	a
		a	[ab]	b				
γ	0	1	2	3	0	0	3	0

iCAp

- Step 4: Finds the cover array from the border and prefix array of *x*.

index	1	2	3	4	5	6	7	8
x	a	[ab]	b	b	a	[ab]	b	a
Π	0	3	0	0	3	2	0	1
β	0	1	2	3	1	2	3	1
						1	2	
γ	0	1	2	3	0	0	3	0

So we check the intervals sequentially to find the covers of x.

iCAp

- Step 4: Finds the cover array from the border and prefix array of *x*.
  - *We use the RMQ algorithm to find out the position of the maximum prefix length in each interval*
  - *We maintain another array which keeps track of the already covered portion of x. So we have no need to check a interval twice.*

iCAp

- Step 4: Finds the cover array from the border and prefix array of x.
  - So for finding a cover of length c, we will have to perform n/c RMQ queries in the worst case.
  - $n/1 + n/2 + n/3 + \ldots + 1$
  - Harmonic Series: $O(n \log n)$

iCAp [Running Time Analysis]

- Worst case running time of the steps are as follows:

Step of Algorithm	Running Time
Step 1	$O(n^2)$
Step 2	$O(n)$
Step 3	$O(n^2)$
Step 4	$O(nlogn)$

- So the overall running time of the algorithm is $O(n^2)$.

Overview

- Problem Definition
- Basic Concepts
- Present State of the Problem
- Our Contributions
- Performance Comparison
- Motivation and Importance
- Conclusion

Performance Comparison

- Computing all cover of x, where $|x| = n$

Algorithm	Running Time	Space Requirement
Conservative String Covering (too restricted)	$O(n^2)$	$O(n^2)$
Antoniou's [2]	$O(n^2 logn)$	$O(n^2)$
iCAb	$O(n^2)$ $O(n)$ average case	$O(n^2)$ $O(n)$ average case
iCAp	$O(n^2)$	$O(n^2)$

Performance Comparison

- Computing the cover array of x, where $|x| = n$

Algorithm	Running Time	Space Requirement
Conservative String Covering (too restricted)	$O(n^3)$	$O(n^2)$
Antoniou's [2]	$O(n^2 \log n)$	$O(n^2)$
iCAb	$O(n^2)$	$O(n^2)$
iCAp	$O(n^2)$	$O(n^2)$

Overview

- Problem Definition
- Basic Concepts
- Present State of the Problem
- Our Contributions
- Performance Comparison
- Motivation and Importance
- Conclusion

Motivation and importance

- Theoretical and Combinatorial point of view
- Computational biology
- Efficient algorithms for degenerate strings

Motivation: Theoretical

- Repeats
- Borders
- Prefixes
- Covers
- Seeds

Motivation: Computational biology

- Degenerate strings are very much applicable especially in the context of computational biology .
  - Errors in experimentations

1	gtatcaccgccagtggtat
2	ataccactggcggtgatac
3	tcaacaccgccagagataa
4	ttatctctggcggtgttga
5	ttatcaccgcagatggtta
6	taaccatctgcggtgataa
7	ctatcaccgcaagggataa
8	ttatcccttgcggtgatag
9	ctaacaccgtgcgtgttga
10	tcaacacgcacggtgttag
11	ttacctctggcggtgataa
12	ttatcaccgccagaggtaa

[acgt][act]a[act]c[at][ct][cgt][cgt][acgt][acg][acg][ag][atg]g[atg]t[atg][actg]

Motivation: Computational biology

- Tandem repeat => individual's inherited traits.
  - short nucleotide sequences
  - occur in adjacent or overlapping positions
- This type of repetition is exactly what is described by the cover array.

Motivation: Efficient Algorithm

- No efficient pattern matching algorithm for degenerate strings yet.
- Why?
  - Efficient algorithms on regular strings depends on regularities
    - KMP, failure function, Boyer-Moore
  - Absence of results on regularities?
- This has motivated researchers in stringology to study the regularities of degenerate strings with great interest in recent times.

Overview

- Problem Definition
- Basic Concepts
- Present State of the Problem
- Our Contributions
- Performance Comparison
- Motivation and Importance
- Conclusion

Conclusion

- Our Contribution:
  - Theoretical insight on different regularities for degenerate strings
  - The best algorithms so far for some regularities in degenerate strings

- Future Directions:
  - Efficient algorithms for degenerate strings?
  - Improvement of these algorithms

Questions?

Thank You

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 repetitive structures in indeterminate strings*. Proceedings of the 3rd IAPR International Conference on Pattern Recognition in Bioinformatics (PRIB 2008), 2008.