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, $x = \text{abbaccbbabbca}$

- **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

- \( w \) is a **substring** of \( x \) and \( x \) is a **superstring** of \( w \).

\[
\begin{align*}
  x & = \text{abbacccbbabbbca} \\
  w & = \text{accbbbab} \\
  u & = \text{bbac} \\
  v & = \text{babbca}
\end{align*}
\]

- \( u \) is a **prefix** and \( v \) is a **suffix** of \( x \).
Basic Concepts

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 $y$

  $$x = \text{abbacaabc} \quad y = \text{ccbabbbcab}$$

  $$xy = \text{abbacaabc}c\text{ccbabbbcab}$$

- $xy$ 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 $y$

  $x = \text{abbacaabc}$ \quad $y = \text{aabcbbbcab}$

• Where $x$ has a suffix equal to a prefix of $y$ we can get a new string by overlapping $x$ and $y$.

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

• **Cover** of $x$

  $x = \text{aabaabaa}\text{aabaabaa}$
  $\text{aabaa} \text{aabaa} \text{aabaa} $
  $\text{aabaa} \text{aabaa}$

  $w = \text{aabaa}$

  **concatenation**

  **superposition**

• 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**, $\Pi$ 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 $i \in \{1...n\}$, $\Pi[i] = \text{length of the longest prefix of } x[1...i]$ 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 | 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 $x[1 \ldots i]$ 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 \( \Sigma \) 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:

\[ x = aabacbcaaabacbac \]

\[ x = aa[abc]a[ac]bcaaac[ac]bac[abc]a[bc] \]
Basic Concepts

Matching in degenerate strings

• Given a degenerate string $x$, we say that

  ▫ $x[i]$ matches $x[j]$ iff $x[i] \cap x[j] \neq \emptyset$

  ▫ $x[i]$ 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

<table>
<thead>
<tr>
<th>Index</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
<th>18</th>
<th>19</th>
<th>20</th>
<th>21</th>
<th>22</th>
<th>23</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>(\Pi)</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>6</td>
<td>0</td>
<td>1</td>
<td>8</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>8</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>(\beta)</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>7</td>
<td>8</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>(\gamma)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>5</td>
<td>6</td>
<td>0</td>
<td>5</td>
<td>6</td>
<td>0</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>0</td>
<td>8</td>
<td>0</td>
<td>3</td>
</tr>
</tbody>
</table>
Mathematical representation

- For every prefix \( x[1 \ldots i] \) 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

<table>
<thead>
<tr>
<th>Index</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x = )</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>[ab]</td>
<td>b</td>
</tr>
<tr>
<td>( \beta = )</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
Border and cover array of a degenerate string

<table>
<thead>
<tr>
<th>Index</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x = $</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>[ab]</td>
<td>[ab]</td>
<td>a</td>
</tr>
<tr>
<td>$\beta = $</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>$\gamma = $</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>
Prefix array of a degenerate string

<table>
<thead>
<tr>
<th>Index</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x =$</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>[ab]</td>
<td>b</td>
</tr>
<tr>
<td>$\Pi =$</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>$\beta =$</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>
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, $\lambda$.
• 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 $\lambda$).
  ▫ conservative seeds (of length $\lambda$).
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:

  • $iCA_{Ab}$: It uses border array and Aho-Corasick Automaton for computing all covers and the cover array.

  • $iCA_{Ap}$: 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 \( x \)

\[
| \beta | (1, a) (2, a) (1, a) (2, a) (3, a) (4, b) (5, a) (1, a) (2, a) (3, a) (4, b) (5, a) (6, *) (1, a) (2, a) (1, a) (2, a) (1, a) |
\]
For Computing all the cover of $x$ we only need the last entries of the border array.
iCAb (STEP 2)

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

Parse $x$ through it to find out the borders that covers $x$. 
iCAb (STEP 2)

For computing the cover array of $x$ we need to process all the entries of the border array.
iCAb (STEP 2)

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

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.
The possible equality cases are:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Match To</th>
<th>Number of cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\sigma \in {1, 2, \ldots, \alpha}$</td>
<td>$\sigma \in {1, 2, \ldots, \alpha}$</td>
<td>$\alpha$</td>
</tr>
<tr>
<td>$\sigma \in S, S \subseteq \Sigma$</td>
<td>$\sigma \in S, S \subseteq \Sigma,</td>
<td>S</td>
</tr>
</tbody>
</table>

Expected number of borders:

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

So the running time reduces to $O(n)$ on average.
This algorithm was recently published in The Prague Stringology Conference, 2009.
The iCAP Algorithm
iCAP

• **Step 1**: Finds the prefix array of $x$.

<table>
<thead>
<tr>
<th>index</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>a</td>
<td>[ab]</td>
<td>b</td>
<td>b</td>
<td>a</td>
<td>[ab]</td>
<td>b</td>
<td>a</td>
</tr>
<tr>
<td>Π</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

- 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
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**

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

<table>
<thead>
<tr>
<th>index</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td>a</td>
<td>[ab]</td>
<td>b</td>
<td>b</td>
<td>a</td>
<td>[ab]</td>
<td>b</td>
<td>a</td>
</tr>
<tr>
<td>$\Pi$</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$\beta$</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>
Prefix array of a degenerate string

<table>
<thead>
<tr>
<th>Index</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x =$</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>[ab]</td>
<td>b</td>
</tr>
<tr>
<td>$\Pi =$</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>$\beta =$</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>
iCAP

• **Step 3**: Finds the border array from the prefix array of $x$.

<table>
<thead>
<tr>
<th>index</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td>a</td>
<td>[ab]</td>
<td>b</td>
<td>b</td>
<td>a</td>
<td>[ab]</td>
<td>b</td>
<td>a</td>
</tr>
<tr>
<td>$\Pi$</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$\beta$</td>
<td>0</td>
<td>1 2 3</td>
<td>1 2 3</td>
<td>1 2 3</td>
<td>1</td>
<td>2 3</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

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

- **Step 3**: Finds the cover array from the border and prefix array of $x$.

<table>
<thead>
<tr>
<th>index</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td>a</td>
<td>[ab]</td>
<td>b</td>
<td>b</td>
<td>a</td>
<td>[ab]</td>
<td>b</td>
<td>a</td>
</tr>
<tr>
<td>$\Pi$</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$\beta$</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>$\gamma$</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>
iCAP

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

<table>
<thead>
<tr>
<th>index</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>a</td>
<td>b</td>
<td>a</td>
<td></td>
<td>b</td>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$y$</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>a</td>
<td>b</td>
<td>a</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>a</td>
<td>b</td>
<td>a</td>
<td></td>
<td>a</td>
<td>b</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>$\Pi$</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
### iCAP

<table>
<thead>
<tr>
<th>index</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>x</td>
<td>a</td>
<td>[ab]</td>
<td>b</td>
<td>b</td>
<td>a</td>
<td>[ab]</td>
<td>b</td>
<td>a</td>
</tr>
<tr>
<td>Π</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>β</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>γ</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

### Extended Table

<table>
<thead>
<tr>
<th>index</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>a</td>
<td>[ab]</td>
<td>b</td>
<td>a</td>
<td>[ab]</td>
<td>b</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>a</td>
<td>[ab]</td>
<td>b</td>
<td>b</td>
<td>a</td>
<td>[ab]</td>
<td>b</td>
<td>a</td>
</tr>
<tr>
<td>γ</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>
iCAP

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

<table>
<thead>
<tr>
<th>index</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>a</td>
<td>[ab]</td>
<td>b</td>
<td>b</td>
<td>a</td>
<td>[ab]</td>
<td>b</td>
<td>a</td>
</tr>
<tr>
<td>$\Pi$</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$\beta$</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>$\gamma$</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>0</td>
</tr>
</tbody>
</table>

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 an 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:

<table>
<thead>
<tr>
<th>Step of Algorithm</th>
<th>Running Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step 1</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Step 2</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Step 3</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Step 4</td>
<td>$O(n\log n)$</td>
</tr>
</tbody>
</table>

- 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 $|x| = n$

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Running Time</th>
<th>Space Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conservative String Covering (too restricted)</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>Antoniou’s [2]</td>
<td>$O(n^2 \log n)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>iCAb</td>
<td>$O(n^2)$ $O(n)$ average case</td>
<td>$O(n^2)$ $O(n)$ average case</td>
</tr>
<tr>
<td>iCAp</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
</tbody>
</table>
Performance Comparison

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

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Running Time</th>
<th>Space Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conservative String Covering</td>
<td>$O(n^3)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>(too restricted)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Antoniou’s [2]</td>
<td>$O(n^2 \log n)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>iCAb</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>iCAp</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
</tr>
</tbody>
</table>
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
gt at caccgccagt ggt at
at accact ggcgggt gat ac
t caacaccgccagagat aa
t t at ct ct ggcgggt gt t ga
t t at caccgcagat ggt t a
t aaccat ct gcggt gat aa
t at caccgcaagggat aa
t t at ccct t gcggt gat ag
t ct aacaccgt gcgt gt t ga
t caacacgcacggt gt t ag
 tt acct ct ggcgggt gat aa
 tt at caccgccagaggt aa
Motivation: Computational biology

- Tandem repeat $\Rightarrow$ 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
