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

Md. Faizul Bari M. Sohel Rahman Rifat Shahriyar

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

The Prague Stringology Conference 2009

Outline

1. Introduction
   - Definitions
   - The Problems That We Studied
   - Previous Works
2. Our Contributions
   - Main Results
   - Our Algorithm

Outline

Basic Definitions

For a strings $x = uwv$:

- $|x|$ is the **length** of x
- ϵ is the **empty** string
- $x[i]$ is the ***i*-th symbol** of x
- w is a **substring** of x and x is a **superstring** of w
- $u(v)$ is a **prefix (suffix)** of x
- $x[i \ldots j]$ denotes the substring of x starting at position i and ending at j

Basic Definitions

For strings $x = x[1 \dots n]$ and $y = y[1 \dots m]$:

- xy denotes the **concatenation** of strings x and y.
- x^k denotes the concatenation of k copies of x.
- If $x[n - i + 1 \dots n] = y[1 \dots i]$ for some $i \geq 1$, the string $x[1 \dots n]y[i + 1 \dots m]$ is a **superposition** of x and y. We also say that x overlaps y.

Border and Border Array

Border and Border Array:

- A **border** u of x is a prefix of x that is also a suffix of x.
- That is $u = x[1 \ldots b]$ = $x[n - b + 1 \ldots n]$ for some $b \in \{0 \ldots n - 1\}$.
- The border array of x is an array β such that for all $i \in \{1 \ldots n\}$, $\beta[i]$ = length of the **longest border** of $x[1 \ldots i]$.

Cover and Cover Array

Cover and Cover Array:

- A substring w of x is called a *cover* of x, if x can be constructed by **concatenating** or **overlapping** copies of w. We also say that w covers x.
- For example, if x = *ababaaba*, then *aba* and x are covers of x.
- The *cover array* γ, is a data structure used to store the length of the **longest proper cover** of every prefix of x;
- That is for all $i \epsilon 1 \ldots$n, $\gamma[i]$ = length of the longest proper cover of $x[1 \ldots i]$ or 0.

Indeterminate Strings

Indeterminate Strings:

- An indeterminate string is a sequence $T = T[1]T[2]\ldots T[n]$, where $T[i] \subseteq \Sigma$ for each i, and Σ is a given alphabet of fixed size.
- If at any position in an indeterminate string, $|T[i]| = 1$, we call this a **solid symbol**. However, when $|T[i]| \geq 1$, we call this a **non-solid symbol**.
- In an indeterminate string a non-solid position can contain up to $|\Sigma|$ symbols.

Outline

1. Introduction
   - Definitions
   - The Problems That We Studied
   - Previous Works
2. Our Contributions
   - Main Results
   - Our Algorithm

The Problems That We Studied

The problems that we studied here are

- Finding all the covers of an indeterminate string
- Finding the cover array of an indeterminate string

The Problems That We Studied

The problems that we studied here are

- Finding all the covers of an indeterminate string
- Finding the cover array of an indeterminate string

Outline

1. Introduction
   - Definitions
   - The Problems That We Studied
   - Previous Works
2. Our Contributions
   - Main Results
   - Our Algorithm

Previous Works

Regularities of *conservative* indeterminate strings

- In [1], the authors investigated the regularities of *conservative* indeterminate strings.
- In a conservative indeterminate string the number indeterminate positions is bounded by a constant.
- The authors presented algorithms for finding
  - The smallest *conservative cover* (number of indeterminate position in the cover is bounded by a given constant)
  - λ-conservative covers (conservative covers having a fixed length λ)
  - λ-conservative seeds.

Previous Works

Regularities of *conservative* indeterminate strings

- In [1], the authors investigated the regularities of *conservative* indeterminate strings.
- In a conservative indeterminate string the number indeterminate positions is bounded by a constant.
- The authors presented algorithms for finding
  - The smallest *conservative cover* (number of indeterminate position in the cover is bounded by a given constant)
  - λ-conservative covers (conservative covers having a fixed length λ)
  - λ-conservative seeds.

Previous Works

Regularities of *conservative* indeterminate strings

- In [1], the authors investigated the regularities of *conservative* indeterminate strings.
- In a conservative indeterminate string the number indeterminate positions is bounded by a constant.
- The authors presented algorithms for finding
  - The smallest *conservative cover* (number of indeterminate position in the cover is bounded by a given constant)
  - λ-conservative covers (conservative covers having a fixed length λ)
  - λ-conservative seeds.

Previous Works

Regularities on indeterminate strings (without any restriction)

- Antoniou et al. presented an $O(n \log n)$ algorithm to find the smallest cover of an indeterminate 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.

Previous Works

Regularities on indeterminate strings (without any restriction)

- Antoniou et al. presented an $O(n \log n)$ algorithm to find the smallest cover of an indeterminate 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.

Outline

Main Results

- We devise an algorithm for computing all the covers of an indeterminate string x of length n in $O(n^2)$ time in the worst case.
- We also show that our algorithm works in $O(n)$ time on the average.
- We extend our algorithm to compute the cover array of x in $O(n^2)$ time and $O(n)$ space complexity in the worst case.
- Notably, our algorithm, unlike the algorithm of [1], does not enforce the restriction that the cover or the input string x must be conservative.

Main Results

- We devise an algorithm for computing all the covers of an indeterminate string x of length n in $O(n^2)$ time in the worst case.
- We also show that our algorithm works in $O(n)$ time on the average.
- We extend our algorithm to compute the cover array of x in $O(n^2)$ time and $O(n)$ space complexity in the worst case.
- Notably, our algorithm, unlike the algorithm of [1], does not enforce the restriction that the cover or the input string x must be conservative.

Main Results

- We devise an algorithm for computing all the covers of an indeterminate string x of length n in $O(n^2)$ time in the worst case.
- We also show that our algorithm works in $O(n)$ time on the average.
- We extend our algorithm to compute the cover array of x in $O(n^2)$ time and $O(n)$ space complexity in the worst case.
- Notably, our algorithm, unlike the algorithm of [1], does not enforce the restriction that the cover or the input string x must be conservative.

Main Results

- We devise an algorithm for computing all the covers of an indeterminate string x of length n in $O(n^2)$ time in the worst case.
- We also show that our algorithm works in $O(n)$ time on the average.
- We extend our algorithm to compute the cover array of x in $O(n^2)$ time and $O(n)$ space complexity in the worst case.
- Notably, our algorithm, unlike the algorithm of [1], does not enforce the restriction that the cover or the input string x must be conservative.

Outline

1. Introduction
   - Definitions
   - The Problems That We Studied
   - Previous Works

2. Our Contributions
   - Main Results
   - Our Algorithm

Definition of the The First Problem

We start with a formal definition of the first problem we handle in this paper.

Problem

Computing All Covers of an Indeterminate String over a fixed alphabet.
Input: *We are given an indeterminate string x, of length n on a fixed alphabet Σ.*
Output: *We need to compute all the covers of x.*

Our Algorithm Depends on the Following Facts

Fact

Every cover of string x is also a border of x.

Fact

If u and c are covers of x and $|u| < |c|$ then u must be a cover of c.

Our Algorithm Depends on the Following Lemma

Lemma

The expected number of borders of an indeterminate string is bounded by a constant.

The Algorithm

- In the first step, the deterministic border array of x is computed.
- In the second step, we check each border whether it is a cover of x or not.

The Algorithm

- In the first step, the deterministic border array of x is computed.
- In the second step, we check each border whether it is a cover of x or not.

The First Step

- Here, we utilize the algorithm provided by Holub and Smyth [3] for computing the deterministic border of an indeterminate string.
- The output of the algorithm is a two dimensional list β.
- Each entry β_i of β contains a list of pair (b, ν_a), where b is the length of the border and ν_a represents the required assignment.
- This list is kept sorted in decreasing order of border lengths of $x[1 \ldots i]$.

The First Step

- Here, we utilize the algorithm provided by Holub and Smyth [3] for computing the deterministic border of an indeterminate string.
- The output of the algorithm is a two dimensional list β.
- Each entry β_i of β contains a list of pair (b, ν_a), where b is the length of the border and ν_a represents the required assignment.
- This list is kept sorted in decreasing order of border lengths of $x[1 \ldots i]$.

The First Step

- Here, we utilize the algorithm provided by Holub and Smyth [3] for computing the deterministic border of an indeterminate string.
- The output of the algorithm is a two dimensional list β.
- Each entry β_i of β contains a list of pair (b, ν_a), where b is the length of the border and ν_a represents the required assignment.
- This list is kept sorted in decreasing order of border lengths of $x[1 \dots i]$.

The First Step

- Here, we utilize the algorithm provided by Holub and Smyth [3] for computing the deterministic border of an indeterminate string.
- The output of the algorithm is a two dimensional list β.
- Each entry β_i of β contains a list of pair (b, ν_a), where b is the length of the border and ν_a represents the required assignment.
- This list is kept sorted in decreasing order of border lengths of $x[1 \ldots i]$.

The First Step: Running Time Analysis

- If we assume that the maximum number of borders of any prefix of x is m, then the worst case running time of the algorithm is $O(nm)$.
- But from Lemma 3 we know that the expected number of borders of an indeterminate string is bounded by a constant.
- As a result the expected running time of the above algorithm is $O(n)$.

The First Step: Running Time Analysis

- If we assume that the maximum number of borders of any prefix of x is m, then the worst case running time of the algorithm is $O(nm)$.
- But from Lemma 3 we know that the expected number of borders of an indeterminate string is bounded by a constant.
- As a result the expected running time of the above algorithm is $O(n)$.

The First Step: Running Time Analysis

- If we assume that the maximum number of borders of any prefix of x is m, then the worst case running time of the algorithm is $O(nm)$.
- But from Lemma 3 we know that the expected number of borders of an indeterminate string is bounded by a constant.
- As a result the expected running time of the above algorithm is $O(n)$.

The Second Step

- Now, we find out the covers of string x. Here we need only the last entry of the border array, β_n, where $n = |x|$.
- To identify a border as a cover of x we use the pattern matching technique of an Aho-Corasick automaton.
- We build an Aho-Corasick automaton with the dictionary containing the border of x and parse x through the automaton to find out whether x can be covered by the it or not.

The Second Step

- Now, we find out the covers of string x. Here we need only the last entry of the border array, β_n, where $n = |x|$.
- To identify a border as a cover of x we use the pattern matching technique of an Aho-Corasick automaton.
- We build an Aho-Corasick automaton with the dictionary containing the border of x and parse x through the automaton to find out whether x can be covered by the it or not.

The Second Step

- Now, we find out the covers of string x. Here we need only the last entry of the border array, β_n, where $n = |x|$.
- To identify a border as a cover of x we use the pattern matching technique of an Aho-Corasick automaton.
- We build an Aho-Corasick automaton with the dictionary containing the border of x and parse x through the automaton to find out whether x can be covered by the it or not.

The Second Step: Building the Aho-Corasick automaton

- Suppose in iteration i, we have the length of the ith border of β_n equal to b.
- In this iteration, we build an Aho-Corasick automaton for the following dictionary:
-

$$D = \{x[1]x[2]\dots x[b]\}, \qquad \text{where } \forall j \in \text{ 1 to b }, \; x[j] \in \Sigma$$

The Second Step: Building the Aho-Corasick automaton

- Suppose in iteration i, we have the length of the ith border of β_n equal to b.
- In this iteration, we build an Aho-Corasick automaton for the following dictionary:
-

$$D = \{x[1]x[2]\ldots x[b]\}, \qquad \text{where } \forall j \in \text{ 1 to b }, \; x[j] \in \Sigma$$

The Second Step: Building the Aho-Corasick automaton

- Suppose in iteration i, we have the length of the ith border of β_n equal to b.
- In this iteration, we build an Aho-Corasick automaton for the following dictionary:
- $$D = \{x[1]x[2]\ldots x[b]\}, \qquad \text{where } \forall j \in \text{ 1 to b }, \; x[j] \in \Sigma$$

The Second Step: Function isCover

Algorithm 1 formally presents the steps of a function *isCover*(), which is the heart of the second step.

```
Construct the Aho-Corasick automaton for c
parse x and compute the positions where c occurs in x and
put the positions in the array Pos
for i = 2 to |Pos| do
    if Pos[i] − Pos[i − 1] > |c| then
        Return FALSE
    end if
end for
Return TRUE
```

Algorithm 1: Function isCover(x, c)

The Second Step: Running Time Analysis of Algorithm 1

- Clearly, Steps 3 and 2 run in $O(n)$.
- Now, the complexity of Step 1 is linear in the size of the dictionary on which the automaton is build.
- Here the length of the string in the dictionary can be $n-1$ in the worst case. So, the time and space complexity of this algorithm is $O(n)$.

The Second Step: Running Time Analysis of Algorithm 1

- Clearly, Steps 3 and 2 run in $O(n)$.
- Now, the complexity of Step 1 is linear in the size of the dictionary on which the automaton is build.
- Here the length of the string in the dictionary can be $n-1$ in the worst case. So, the time and space complexity of this algorithm is $O(n)$.

The Second Step: Running Time Analysis of Algorithm 1

- Clearly, Steps 3 and 2 run in $O(n)$.
- Now, the complexity of Step 1 is linear in the size of the dictionary on which the automaton is build.
- Here the length of the string in the dictionary can be $n - 1$ in the worst case. So, the time and space complexity of this algorithm is $O(n)$.

The Second Step: A Further Improvement

- According to Fact 2, if u and c are covers of x and $|u| < |c|$ then u must be a cover of c.
- Now if $\beta_n = \{b_1, b_2, \ldots, b_m\}$ then from the definition of border array $b_1 > b_2 > \ldots > b_n$.
- Now if in any iteration we find a b_i that is a cover of x then from Fact 2, we can say that for all $j \in i + 1 \ldots m$, if b_j is a cover of x if and only if b_j is a cover of b_i.
- So instead of parsing x we can parse b_i for the subsequent automatons and as $|b_i| \leq |x|$.

The Second Step: A Further Improvement

- According to Fact 2, if u and c are covers of x and $|u| < |c|$ then u must be a cover of c.
- Now if $\beta_n = \{b_1, b_2, \ldots, b_m\}$ then from the definition of border array $b_1 > b_2 > \ldots > b_n$.
- Now if in any iteration we find a b_i that is a cover of x then from Fact 2, we can say that for all $j \in i + 1 \ldots m$, if b_j is a cover of x if and only if b_j is a cover of b_i.
- So instead of parsing x we can parse b_i for the subsequent automatons and as $|b_i| \leq |x|$.

The Second Step: A Further Improvement

- According to Fact 2, if u and c are covers of x and $|u| < |c|$ then u must be a cover of c.
- Now if $\beta_n = \{b_1, b_2, \ldots, b_m\}$ then from the definition of border array $b_1 > b_2 > \ldots > b_n$.
- Now if in any iteration we find a b_i that is a cover of x then from Fact 2, we can say that for all $j \in i + 1 \ldots m$, if b_j is a cover of x if and only if b_j is a cover of b_i.
- So instead of parsing x we can parse b_i for the subsequent automatons and as $|b_i| \leq |x|$.

The Second Step: A Further Improvement

- According to Fact 2, if u and c are covers of x and $|u| < |c|$ then u must be a cover of c.
- Now if $\beta_n = \{b_1, b_2, \ldots, b_m\}$ then from the definition of border array $b_1 > b_2 > \ldots > b_n$.
- Now if in any iteration we find a b_i that is a cover of x then from Fact 2, we can say that for all $j \in i+1 \ldots m$, if b_j is a cover of x if and only if b_j is a cover of b_i.
- So instead of parsing x we can parse b_i for the subsequent automatons and as $|b_i| \leq |x|$.

The Second Step: Overall Algorithms

```
k ← n
AC ← ϕ {AC is a list used to store the covers of x}
for all b ϵ β_n do
    if isCover(x[1..k], x[1..b]) = true then
        m ← b
        AC.Add(k)
    end if
end for
```

Algorithm 2: Computing all covers of x

The Second Step: Running Time Analysis

- The running time of Algorithm 2 is $O(nm)$, where m is number of borders of x or alternatively number of entries in β_n.
- Again, from Lemma 3 we can say that the number of borders of an indeterminate string is bounded by a constant on average.
- Hence, the expected running time of Algorithm 2 is $O(n)$.

The Second Step: Running Time Analysis

- The running time of Algorithm 2 is $O(nm)$, where m is number of borders of x or alternatively number of entries in β_n.
- Again, from Lemma 3 we can say that the number of borders of an indeterminate string is bounded by a constant on average.
- Hence, the expected running time of Algorithm 2 is $O(n)$.

The Second Step: Running Time Analysis

- The running time of Algorithm 2 is $O(nm)$, where m is number of borders of x or alternatively number of entries in β_n.
- Again, from Lemma 3 we can say that the number of borders of an indeterminate string is bounded by a constant on average.
- Hence, the expected running time of Algorithm 2 is $O(n)$.

The Second Step: Running Time Analysis

- It follows from above that our algorithm for finding all the covers of an indeterminate string of length n runs in $O(n)$ time on the average.
- The worst case complexity of our algorithm is $O(nm)$, i.e., $O(n^2)$.
- Which is also an improvement since the current best known algorithm [2] for finding all covers requires $O(n^2 \log n)$ in the worst case.

The Second Step: Running Time Analysis

- It follows from above that our algorithm for finding all the covers of an indeterminate string of length n runs in $O(n)$ time on the average.
- The worst case complexity of our algorithm is $O(nm)$, i.e., $O(n^2)$.
- Which is also an improvement since the current best known algorithm [2] for finding all covers requires $O(n^2 \log n)$ in the worst case.

The Second Step: Running Time Analysis

- It follows from above that our algorithm for finding all the covers of an indeterminate string of length n runs in $O(n)$ time on the average.
- The worst case complexity of our algorithm is $O(nm)$, i.e., $O(n^2)$.
- Which is also an improvement since the current best known algorithm [2] for finding all covers requires $O(n^2 \log n)$ in the worst case.

Definition of the The Second Problem

We start with a formal definition of the second problem we handle in this paper.

Problem

Computing the Cover array of an Indeterminate String over a fixed alphabet.
Input: *We are given an indeterminate string x, of length n on a fixed alphabet Σ.*
Output: *We need to compute the cover array of x.*

Algorithm for Computing the Cover Array

- Here we only need the length of the largest border of each prefix of x. This information is stored in the first entry of each β_i of the border array.
- Let us assume that $\beta_i[1]$ denotes the first entry of the list β_i that is $\beta_i[1]$ is the length of the largest border of $x[1 \ldots i]$.

Algorithm for Computing the Cover Array

- Here we only need the length of the largest border of each prefix of x. This information is stored in the first entry of each β_i of the border array.
- Let us assume that $\beta_i[1]$ denotes the first entry of the list β_i that is $\beta_i[1]$ is the length of the largest border of $x[1 \ldots i]$.

Algorithm for Computing the Cover Array

1: $\gamma[i] \leftarrow 0 \qquad \forall\ i \in \{1 \ldots n\}$
2: **for** $i \leftarrow 1$ to n **do**
3:     **if** $isCover(x, x[1 \ldots \beta_{i[1]}]) = true$ **then**
4:         $\gamma[i] \leftarrow \beta_{i[1]}$
5:     **end if**
6: **end for**

Algorithm 3: Computing cover array γ of x

Running Time Analysis

- As the worst case running time of the $isCover(x, c)$ function is $O(n)$ and the algorithm iterates over the n lists of the border array β, the running time of Algorithm 3 is $O(n^2)$.

Summary

- In this paper we have presented an average case $O(n)$ time and space complex algorithm for computing all the covers of a given indeterminate string x of length n.
- We have also presented an algorithm for computing the cover array γ of an indeterminate string. This algorithm requires $O(n^2)$ time and $O(n)$ space in the worst case.
- Both of these algorithms are improvement over existing algorithms.

Thank You

References I

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.

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.

References II

J. Holub and W. F. Smyth:
Algorithms on indeterminate strings.
Miller, M., Park, K. (eds.): Proceedings of the 14th Australasian Workshop on Combinatorial Algorithms AWOCA'03, 2003, pp. 36–45.