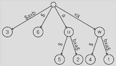
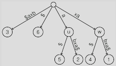


Datenstrukturen und Effiziente Algorithmen

Vorlesung *Datenstrukturen und Effiziente Algorithmen* im WS 18/19

Marc Hellmuth
Institut für Mathematik und Informatik
Universität Greifswald



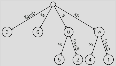
Exact String Matching

Exact String Matching Problem

Given pattern P of length n and text T of length $m \geq n$ find all (k) occurences of P in T as substring.

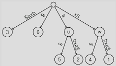
Solutions

- build suffix tree $\mathcal{T}$ for T, search path labeled P in $\mathcal{T}$, then all leaves below end of path
  preprocessing of T: $O(m)$, searching $O(n+k)$
- Boyer-Moore preprocessing of P: $O(n)$, searching $O(m)$
- will later see: suffix trees allow preprocessing of P, too, then same time bounds as Boyer-Moore or the Z-Algorithm



Exact Set Matching Problem

Find all k occurrences of any pattern in a set $\mathcal{P} = \{P_1, P_2, \ldots, P_z\}$ of patterns with total length n as substring in text T of length m.



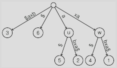
Exact Set Matching Problem

Find all k occurrences of any pattern in a set $\mathcal{P} = \{P_1, P_2, \ldots, P_z\}$ of patterns with total length n as substring in text T of length m.

Example 1

BLAST

- approximate similarity search program for protein sequences
- most highly cited paper published in the 1990s
- as a subtask and heuristik speedup BLAST needs to find exact matches of a set of patterns in a very large text (database)



Exact Set Matching Problem

Find all k occurrences of any pattern in a set $\mathcal{P} = \{P_1, P_2, \ldots, P_z\}$ of patterns with total length n as substring in text T of length m.

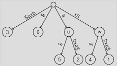
Example 1

BLAST

- approximate similarity search program for protein sequences
- most highly cited paper published in the 1990s
- as a subtask and heuristik speedup BLAST needs to find exact matches of a set of patterns in a very large text (database)

Solutions

- naive solution with repeated Z-Algorithm or Boyer-Moore takes time



Exact Set Matching Problem

Find all k occurrences of any pattern in a set $\mathcal{P} = \{P_1, P_2, \ldots, P_z\}$ of patterns with total length n as substring in text T of length m.

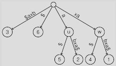
Example 1

BLAST

- approximate similarity search program for protein sequences
- most highly cited paper published in the 1990s
- as a subtask and heuristik speedup BLAST needs to find exact matches of a set of patterns in a very large text (database)

Solutions

- naive solution with repeated Z-Algorithm or Boyer-Moore takes time



Exact Set Matching Problem

Find all k occurrences of any pattern in a set $\mathcal{P} = \{P_1, P_2, \ldots, P_z\}$ of patterns with total length n as substring in text T of length m.

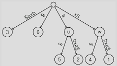
Example 1

BLAST

- approximate similarity search program for protein sequences
- most highly cited paper published in the 1990s
- as a subtask and heuristik speedup BLAST needs to find exact matches of a set of patterns in a very large text (database)

Solutions

- naive solution with repeated Z-Algorithm or Boyer-Moore takes time $O(n + zm)$
- Aho-Corasick algorithm (1975) builds a keyword tree for $\mathcal{P}$ preprocessing: $O(n)$, search: $O(m + k)$



Exact Set Matching Problem

Find all k occurrences of any pattern in a set $\mathcal{P} = \{P_1, P_2, \ldots, P_z\}$ of patterns with total length n as substring in text T of length m.

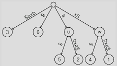
Example 1

BLAST

- approximate similarity search program for protein sequences
- most highly cited paper published in the 1990s
- as a subtask and heuristik speedup BLAST needs to find exact matches of a set of patterns in a very large text (database)

Solutions

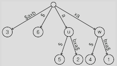
- naive solution with repeated Z-Algorithm or Boyer-Moore takes time $O(n + zm)$
- Aho-Corasick algorithm (1975) builds a keyword tree for $\mathcal{P}$
preprocessing: $O(n)$, search: $O(m + k)$
- sufix trees: build suffix tree for T then search each P_i individually
preprocessing: $O(m)$, search: $O(n + k)$



Substring Problem for a Database of Strings

Substring Problem for a Database of Strings

One searches occurrences of a pattern P as substring in a set $\mathcal{S} = \{S_1, S_2, \ldots, S_z\}$ of strings (database). This task repeatedly occurs for the same database and different patterns P.



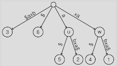
Substring Problem for a Database of Strings

Substring Problem for a Database of Strings

One searches occurrences of a pattern P as substring in a set $\mathcal{S} = \{S_1, S_2, \ldots, S_z\}$ of strings (database). This task repeatedly occurs for the same database and different patterns P.

Example 2 (MIA: identifying dead soldiers)

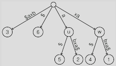
- mitochondrial DNA of the remains of a dead soldier (P) is searched against a database of mitochondrial DNA regions $\mathcal{S}$
- in constructing $\mathcal{S}$ a certain highly variable region is chosen, so individuals can be distinguished
- P is allowed to be a substring (condition of DNA sample)



Substring Problem for a Database of Strings

Solution

- with suffix trees
  1. build generalized suffix tree $\mathcal{T}$ for $\mathcal{S}$
  2. starting from the root, search path labeled P in $\mathcal{T}$
  3. if no such path is found, then P does not occur in any string in $\mathcal{S}$
  4. if such a path is found then, then visit all leaves below the end of the path. For each leaf with label (i, j) report an occurence of P in S_i starting at position j
- preprocessing: $O(m)$, search $O(n + k)$, where k is the number of occurrences
- cannot achieve the same efficiency with Z-Algorithm, Boyer-Moore or Aho-Corasick



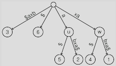
Longest Common Substring

Longest Common Substring Problem for Two Strings

Find a longest common substring of two given strings S_1 and S_2.

Example 3

$S_1 = \texttt{hopfenstange}$
$S_2 = \texttt{kippfenster}$
$P = \texttt{pfenst}$ is a longest common substring of S_1 and S_2.



Longest Common Substring

Longest Common Substring Problem for Two Strings

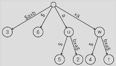
Find a longest common substring of two given strings S_1 and S_2.

Example 3

$S_1 = \texttt{hopfenstange}$
$S_2 = \texttt{kippfenster}$
$P = \texttt{pfenst}$ is a longest common substring of S_1 and S_2.

Remark

Don Knuth conjectured 1970 that a linear-time algorithm for this problem is impossible.



Longest Common Substring

Solution

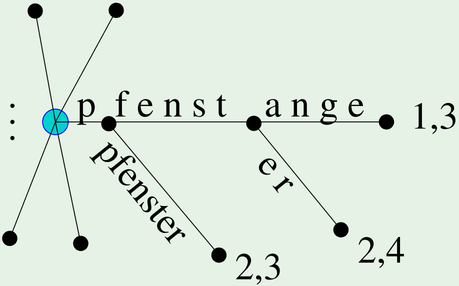
1. build a generalized suffix tree for $\{S_1, S_2\}$
2. mark each internal node v with a 1 (2) if there is a leaf in the subtree rooted at v that is representing a suffix of S_1 (S_2)
3. find a node v marked 1 and 2 with highest string-depth
4. the label of v is a longest substring of S_1 and S_2

Running time

- step 1 takes linear time as shown before
- steps 2 and 3 are easily done in linear time with standard tree traversal methods
- overall running time $O(|S_1| + |S_2|)$

Longest Common Substring

Example 4 (S_1 = `hopfenstange`, S_2 = `kippfenster`)



Ziv-Lempel Data Compression

Ziv-Lempel

- family of algorithms for data compression (gzip, winzip, winrar) based on an idea of Ziv and Lempel in 1977
- we here consider one of the variants on an abstract level
- idea: compress a string S (a file) by saving space by storing repeated copies of substrings implicitly rather than explicitly

Ziv-Lempel Data Compression

Ziv-Lempel

- family of algorithms for data compression (gzip, winzip, winrar) based on an idea of Ziv and Lempel in 1977
- we here consider one of the variants on an abstract level
- idea: compress a string S (a file) by saving space by storing repeated copies of substrings implicitly rather than explicitly

Definition 5

For any position i in a string S of length m, define the substring p_i to be the longest prefix of $S[i..m]$ that also occurs as a substring in $S[1..i-1]$.
Let $\ell_i := |p_i|$ and, if $\ell_i > 0$ define s_i to be the starting point of the left-most copy of p_i in S.

Ziv-Lempel Data Compression

Ziv-Lempel

- family of algorithms for data compression (gzip, winzip, winrar) based on an idea of Ziv and Lempel in 1977
- we here consider one of the variants on an abstract level
- idea: compress a string S (a file) by saving space by storing repeated copies of substrings implicitly rather than explicitly

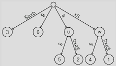
Definition 5

For any position i in a string S of length m, define the substring p_i to be the longest prefix of $S[i..m]$ that also occurs as a substring in $S[1..i-1]$.
Let $\ell_i := |p_i|$ and, if $\ell_i > 0$ define s_i to be the starting point of the left-most copy of p_i in S.

Example 6

$S =$ "`blaukraut bleibt blaukraut und brautkleid bleibt brautkleid`"

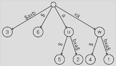
$p_{18} =$ "`blaukraut `", $\ell_{18} = 10$, $s_i = 1$



Ziv-Lempel Data Compression

Compression Algorithm

- when ℓ_i is large, then storing (s_i, ℓ_i) instead of $S[i..i+\ell_i-1]$ takes less space
- when $S[1..i-1]$ is known then $S[1..i+\ell_i-1]$ can be reconstructed with (s_i, ℓ_i) ('decompressed')
- compression and decompression goes left-to-right



Ziv-Lempel Data Compression

Compression Algorithm

- when ℓ_i is large, then storing (s_i, ℓ_i) instead of $S[i..i+\ell_i-1]$ takes less space
- when $S[1..i-1]$ is known then $S[1..i+\ell_i-1]$ can be reconstructed with (s_i, ℓ_i) ('decompressed')
- compression and decompression goes left-to-right

```
i ← 1
repeat
    compute ℓ_i and s_i
    if ℓ_i > 0 then
        output (s_i, ℓ_i)
        i ← i + ℓ_i
    else
        output S[i]
        i ← i + 1
until i > n
```

Ziv-Lempel Data Compression

Example 7

$S =$ “`blaukraut bleibt blaukraut und brautkleid bleibt brautkleid`”

```
          |      |      |      |      |
```

becomes

`blaukr(3,2)t (1,2)ei(1,1)(8,4)(3,8)(4,1)nd`

`(10,2)(6,4)(5,1)(12,3)(30,3)(12,7)(33,9)`

Ziv-Lempel Data Compression

Example 7

$S =$ "`blaukraut bleibt blaukraut und brautkleid bleibt brautkleid`"

```
          |       |       |       |       |
```

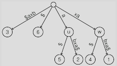
becomes

`blaukr(3,2)t (1,2)ei(1,1)(8,4)(3,8)(4,1)nd`

`(10,2)(6,4)(5,1)(12,3)(30,3)(12,7)(33,9)`

Remark

For real compression algorithms there should be a threshold for ℓ_i and implicit storage should only be chosen when it uses less space. This depends on the actual encoding to bits and bytes.



Ziv-Lempel Data Compression

Implementation of compression with suffix trees

1. compute a suffix tree $\mathcal{T}$ for S
2. mark each node v with the smallest suffix number of a leaf at or below v: c_v
   c_v is the position of the first occurence of the path label of v in S
3. to compute s_i and ℓ_i start matching $S[i..m]$ from the root of $\mathcal{T}$. Let p be a traversed point in $\mathcal{T}$. If p is not itself a node, then let c_p be c_v where v is the nearest node below p. The traversal ends at the deepest point p, such that
   - the path label of p is a prefix of $S[i..m]$ and
   - $|\text{path-label of } p| + c_p \leq i$
     (the first occurence is contained in $S[1..i-1]$)
4. $\ell_i \leftarrow |\text{path-label of } p|$, $s_i \leftarrow c_p$

The running time to compress S with above algorithm is $O(m)$.