

Biosequence Algorithms, Spring 2005 Lecture 2

Pekka Kilpeläinen

University of Kuopio
Department of Computer Science

BSA Lecture 2: Exact string matching – p.1/18

I: Exact String Matching (tarkka (merkkijono)hahmon sovitus)

- the naive method
- a linear-time method based on “fundamental preprocessing”

BSA Lecture 2: Exact string matching – p.2/18

Exact String Matching Problem

Perhaps the most basic string problem of all:

Given **pattern** P (*hahmo*) and **target** T (*kohde*), find all **occurrences** of P in T (that is, substrings equal to P)

Example: Pattern $P = \text{“aba”}$ occurs in text

i : 123456789012

T : bbabaxababay

at locations $i = 3, i = 7$, and $i = 9$

Multiple applications: word processing, file searching (Unix `grep`), information searching on the Net, *sequence databases*

BSA Lecture 2: Exact string matching – p.3/18

Relevance of Exact Match Algorithms?

For practical word-processing the problem can be considered solved

Why then study exact matching?

- Efficient solutions relevant for sequence DBs. (For example, *over 4 hour* search in *Genbank* for a 30 char pattern using a popular interface (GCG) vs a *few minutes* using the Boyer-Moore algorithm.)
- Used as a *subtask* for more complex searches
- Basic ideas possibly applicable to new and less understood problems.

BSA Lecture 2: Exact string matching – p.4/18

Naive Pattern Matching

Compare $P[1 \dots n]$ char-by-char against each n -length substring of $T[1 \dots m]$:

```
for  $i := 1$  to  $m - n + 1$  do
  if  $T[i] = P[1]$  then
     $l := 1$ ; // chars matched
    while  $l < n$  and  $T[i + l] = P[l + 1]$  do  $l := l + 1$ ;
    if  $l = n$  then Report a match at  $i$ ;
  endif;
endfor;
```

Drawback: $n(m - n + 1) = \Theta(nm)$ comparisons in the worst case; Rare in word processing, but probable if small alphabet and lots of repetitions in strings (as in bio-sequences)

BSA Lecture 2: Exact string matching – p.5/18

Illustration

Naive method “shifts” P by one position along the target:

T : xabcdabcdabcx

P : a bcdabcx

abcdabc x

a bcdabcx

a bcdabcx

a bcdabcx

abcdabcx

(Legend: successful and unsuccessful comparison);

20 comparisons in total

BSA Lecture 2: Exact string matching – p.6/18

Ideas for Speed-up I

I: Use longer shifts that avoid comparisons known to fail:

T : xabcdabcdabcx

P : a bcdabcx

abcdabc x (AHA: $P[1]$ doesn't occur

abcdabcx until a shift by 4)

↪ total of 17 comparisons

BSA Lecture 2: Exact string matching – p.7/18

Ideas for Speed-up II

II: Avoid comparisons known to succeed:

T : xabcdabcdabcx

P : a bcdabcx

abcdabc x

abcdabcx

From earlier comparisons, we know the prefix “abc” to match; ↪ total of 14 comparisons

Next: Preprocessing the pattern to implement these ideas

↪ linear-time ($O(|P| + |T|)$) pattern matching algorithms

BSA Lecture 2: Exact string matching – p.8/18

Fundamental Preprocessing

Developed by Gusfield, to explain diverse classical algorithms; also leads to simple linear time matching

Given a string $S[1 \dots n]$ and $i \in \{2, \dots, n\}$, define Z_i to be the length of the longest common prefix of S and $S[i \dots n]$

Example: For $S[1 \dots 11] = aabcaabxaaz$

$$\begin{aligned} Z_2 &= 1, Z_3 = Z_4 = 0 \\ Z_5 &= 3 (\leftarrow S[5 \dots 11] = aabxaaz) \\ &\vdots \\ Z_9 &= 2, Z_{10} = 1, Z_{11} = 0 \end{aligned}$$

If S is not clear from context, we write $Z_i(S)$ instead of Z_i

BSA Lecture 2: Exact string matching - p.9/18

How to compute the Z_i values?

A direct approach \rightsquigarrow time $\Theta(\sum_{i=1}^{n-1} i) = \Theta(n^2)$

Definitions for a linear time solution:

For $Z_i > 0$, let the **Z-box** at i be $S[i \dots i + Z_i - 1]$ (occurrence of a maximal non-empty prefix starting at i).

For every $i \geq 2$, let r_i be the right-most of endpoints of any Z-box at $j \leq i$. (If there is no such, let $r_i = 0$)

If $r_i > 0$, let l_i be the left end of a Z-box $S[j \dots r_i]$ occurring at $j \leq i$. (Otherwise $l_i = 0$.)

BSA Lecture 2: Exact string matching - p.10/18

Example of Z-boxes

Example: (with Z-boxes surrounded by brackets, and indices below):

a b [a b [a b [a]]] x [a b [a]]
1 2 3 4 5 6 7 8 9 10 11

Then

$$\begin{aligned} Z_2 &= 0, r_2 = l_2 = 0 \\ Z_3 &= 5, r_3 = 7, l_3 = 3 \\ Z_4 &= 0, r_4 = 7, l_4 = 3 \\ Z_5 &= 3, r_5 = 7, l_5 = 5, \text{ (or 3)} \\ Z_8 &= 0, r_8 = 7, l_8 = 7, \text{ (or 3, or 5)} \end{aligned}$$

BSA Lecture 2: Exact string matching - p.11/18

Fundamental Preprocessing in Linear Time

Basic method: a single scan of positions $k = 2, \dots, n$ in S , utilizing Z_i values already computed ($2 \leq i < k$);

Variables l and r for the most recent l_i and r_i ; (That is, r is the right-most end of any Z-box seen so far)

To begin, Z_2 is computed by comparing $S[1 \dots n]$ and $S[2 \dots n]$ explicitly, until the first mismatch

BSA Lecture 2: Exact string matching - p.12/18

How to use computed Z_i values?

Example: Suppose that $k = 121$, $r_{120} = 131$ and $l_{120} = 101$; we're inside Z-box $S[101 \dots 131] = S[1 \dots 31]$. Thus $S[121 \dots 131] = S[21 \dots 31]$. (Draw a picture!)

Now if Z_{21} is, say, 9, we know that $Z_{121} = 9$ (without examining any characters).

General method for computing Z_2, \dots, Z_n ,

the Z algorithm:

Initialize: $l := 0; r := 0$;

Then compute Z_k for each $k = 2, \dots, n$ as follows:

BSA Lecture 2: Exact string matching - p.13/18

The Z Algorithm

for $k := 2, \dots, n$ either case 1 or case 2 applies:

1. if $k > r$ then
 $Z_k := \max\{j \leq n - k + 1 \mid S[1 \dots j] = S[k \dots k + j - 1]\}$;
 If $Z_k > 0$, set $l := k$ and $r := k + Z_k - 1$;
2. if $k \leq r$, we're inside Z-box $S[l \dots r] = S[1 \dots Z_l]$, and thus $S[k \dots r] = S[k' \dots Z_l]$ for $k' = k - l + 1$. (Draw a picture!)
 Let $t = |S[k \dots r]|$;
 (a) If $Z_{k'} < t$, we know to set $Z_k := Z_{k'}$.
 (b) Otherwise $S[k \dots r] = S[k' \dots Z_l] = S[1 \dots t]$. Find $j := \max\{j \leq n - r \mid S[r + 1 \dots r + j] = S[t + 1 \dots t + j]\}$; and set $Z_k := t + j$, $r := r + j$, and $l := k$;

BSA Lecture 2: Exact string matching - p.14/18

Correctness and Complexity

Theorem 1.4.1 Algorithm Z is correct.

Proof. Straight-forward inspection. \square

Theorem 1.4.2 Algorithm Z works in time $O(|S|)$.

Proof. Each of the $|S| - 1$ iterations takes, besides the character comparisons (resulting in a match or a mismatch), constant time. Out of the character comparisons ...

each *mismatch* ends an iteration \rightarrow number of them $< |S|$

each *match* increments the value of r at least by 1 \rightarrow number of successful comparisons $\leq |S|$ \square

BSA Lecture 2: Exact string matching - p.15/18

Simplest Linear-Time Matching

The Z algorithm provides a **linear-time matching algorithm**, which is perhaps the simplest of all:

Given $P[1 \dots n]$ and $T[1 \dots m]$,

let $S := P\$T$ (where $\$$ appears in neither P nor T);

Compute $Z_i(S)$ for $i = 2, \dots, m + n + 1$;

This takes time $O(n + m)$

Because of '\$' each $Z_i \leq n$.

Now each position $i > n + 1$ with $Z_i = n$ (and only such) indicates an occurrence of P in T at position $i - (n + 1)$.

BSA Lecture 2: Exact string matching - p.16/18

Space Complexity

How much space do we need for the Z values?

Computed $Z_{k'}$ values are used in Case 2 of Algorithm Z . There we have $k \leq r$ and $S[k \dots r] = S[k' \dots Z_i]$. Therefore $k' \leq Z_i \leq n$, and thus it suffices to store Z_i values for $i \leq n$, i.e., to use $O(|P|)$ space

NB After the preprocessing, algorithm Z performs exactly the comparisons shown on Slide "Ideas for Speed-up II" btw characters of P and T

Why Continue?

We've got a simple linear-time matching algorithm. Why to study others?

- *Boyer-Moore* algorithm is very efficient in practice ("sub-linear time")
- *Knuth-Morris-Pratt* generalizes to matching a *set of patterns* in linear time \rightsquigarrow *Aho-Corasick* algorithm
- *suffix trees* support, after $O(|T|)$ time preprocessing, matching in time $O(|P|)$ (and have many other applications)