Exact Matching, Part 1

photophosphorescent

Steven Bedrick
CS/EE 5/655, 11/12/14
Plan for today:

String-matching problems

Exact matching: naïve algorithm

Exact matching: linear-time algorithm
The exact string-matching problem:

Given a short pattern $P$...

... and a longer text $T$...

... identify the locations in $T$ where instances of $P$ occur.
Often, we have a large number of patterns...

... and a large amount of text...

... and we want to find instances efficiently.

Worst-case: $O(nm)$

Ideally: sub-linear
Approximate matching:

Like exact matching, but allowing “small differences”:

Functionally similar amino acid substitutions/indels?

Vowel shifts, systematic consonant dropping, morphological variants, etc.?
Review: distance metrics:

1. Non-negative property:
   \[ d(a, b) \geq 0 \]  for all \( a \) and \( b \)

2. Zero property:
   \[ d(a, b) = 0 \]  iff \( a = b \)

3. Symmetry:
   \[ d(a, b) = d(b, a) \]  for all \( a \) and \( b \)

4. Triangle inequality:
   \[ d(a, b) + d(b, c) \geq d(a, c) \]  for all \( a, b \) and \( c \)

http://en.wikipedia.org/wiki/Triangle_inequality
Approximate matching:

Precise formulation of “distance” will is heavily application-dependent...

... but the algorithms don’t care!
Today, though, we’re talking about exact.

\[ P = \text{"xtpxtd"} \]

\[ T = \text{"xluxtpxtdqwtdxtpxtsyxtpxtdy"} \]

Occurrences of \( P \) in \( T \):

\[ \text{xluxxtpxtdqwtdxtpxtsyxtpxtdy} \]
Demo
Performance:

If $P$ has length $m$, and $T$ length $n$, worst-case is $O(mn)$.

- suppose $P$ is $aaaaaab$, and $T$ is $aaaaaaaaaaaaaaaaaaa$

Don’t use the naïve algorithm!
Performance:

How might we improve?

\[ P = "xtpxtd" \]

\[ T = "xluxtpxtdqwtedxtpxtsyxtpxtdy" \]

Does our pattern have any internal structure?
Demo
Improvements to exact match almost always involve pre-processing either $P$ or $T$.

The goal: save time (comparisons) by identifying repetitive elements.
The choice of which to focus on is application dependent:

Mapping new DNA sample? Probably transform $P$, since you’ll be working with sequence-tagged sites.

Of course, there are probably errors in your STS library, so...
The choice of which to focus on is application dependent:

Searching for arbitrary words in a large text? Focus on transforming the text.

Today’s algorithm will focus on patterns.
Definitions: for a string $S$:

$S[i,j] = \text{contiguous substring starting at } i \text{ and ending at } j$. $S(i) = S[i,i]$

$S = \text{aardvark}$

$S[2,4] = \text{ard}$  \hspace{1cm} $S(4) = d$

For $i > 1$, $Z_i(S)$ is the length of the longest prefix of $S[i,|S|]$ that is also a prefix of $S$. 
For $i > 1$, $Z_i(S)$ is the length of the longest prefix of $S[i, |S|]$ that is also a prefix of $S$.

$S = xtpxtd$

$S[4, |S|] = xtd$

$Z_4(S) = 2$
For $i > 1$, $Z_i(S)$ is the length of the longest prefix of $S[i, |S|]$ that is also a prefix of $S$.

$$S = \text{xtpxtd}$$

$$S[4, |S|] = \text{xtd}$$

$$Z_4(S) = 2$$

In this string, for all other positions $k$,

$$Z_k(P) = 0.$$
\[ P = \text{aardvark} \]

\[ Z_2(P) = 1 \]
P = aardvark

\[ Z_2(P) = 1 \]
\[ P = \text{aardvark} \]
\[ Z_2(P) = 1 \]
\[ Z_6(P) = 1 \]

\[ P = \text{alfalfa} \]
\[ Z_4(P) = 4 \]
$P = \text{aardvark}$

$Z_2(P) = 1$

$Z_6(P) = 1$

$P = \text{alfalfa}$

$Z_4(P) = 4$

$Z_6(P) = Z_{10}(P) = 3$

$P = \text{photophosphorescent}$

$Z_6(P) = Z_{10}(P) = 3$
\[ P = \textcolor{red}{\text{aardvark}} \]

\[ Z_2(P) = 1 \]

\[ Z_6(P) = 1 \]

\[ P = \textcolor{red}{\text{alfaalfa}} \]

\[ Z_4(P) = 4 \]

\[ P = \textcolor{red}{\text{photophosphorescent}} \]

\[ Z_6(P) = Z_{10}(P) = 3 \]
These regions of prefix-overlap are called z-boxes.

\[ P = \text{photo} \text{phosphorescent} \]
Why do we care?

Let’s stick $P$ and $T$ together with a sentinal character:

$$P$T = pho$photophosphorescent$$
Why do we care?

Let’s stick $P$ and $T$ together with a sentinel character:

$$P$T = pho$photophosphorescent$$

Any point whose $Z_i = |P|$ matches!
How to calculate $Z_i$?

Naïve way: $O(nm)$

There exists a linear-time solution!
Next steps:

Knuth-Morris-Pratt:

A linear-time extension to what we’ve seen, with some clever preprocessing to allow larger shifts.

Boyer-Moore:

A totally different approach, moving right-to-left and often achieving sub-linear performance.