Fast String Searching

J Strother Moore
Department of Computer Sciences
University of Texas at Austin
The Problem

One of the classic problems in computing is *string searching*: find the first occurrence of one character string ("the pattern") in another ("the text").

Generally, the text is very large (e.g., gigabytes) but the patterns are relatively small.
Examples

Find the word “comedy” in this *NY Times* article:

Fred Armisen’s office at “Saturday Night Live” is deceptively small, barely big enough to fit a desk, a couch, and an iPod. The glorified closet, the subject of a running joke on the comedy show, now in its 31st season, can simultaneously house a wisecracking . . .
AAAAAAAAAAAAAAAAAAAAAAGACAGGGCAACAAAGTGAGACCCTAAAAAAAAAAAAACCCCA
AAACGGAAGAACTTGGAATCCTGTGTCACAAAACAGGACAGGACTGCAAGACACGCTATTTATAT
TTGCTTTCTGCAAAAAAAAGACCTAACCTCCGCTAGAGAGGTGTTTGGTTGAAAAATCCCAAA
AAAAAAATAGAGAGTCCCCAATGTTCGGAATACGTCAAAAAAATCTTAGTGCGCTTAAATTAA
CAGACTCTCTCCCGGGAAGGTGGACATGCAGAACCTACCAAAAAAAAAGAGAAGAAAGAAT
TGCCCCACAAAAAAAGACCTTCTACCCTTTCGCAAAAAAGAAATGAGATCGTTTGCTGCAATCCAA
AAAAAGACTCTCTCCCGGAAGGTGGACATGCAGAACCTACCAAAAAAAAAGAGAAGAAAGAAT
TGCCCGGCAAAAAAGACCTTCTACCCTTTCGCAAAAAAGAAATGAGATCGTTTGCTGCAATCCAA
AAAAAAATAGAGAGTCCCCAATGTTCGGAATACGTCAAAAAAATCTTAGTGCGCTTAAATTAA
CAGACTCTCTCCCGGGAAGGTGGACATGCAGAACCTACCAAAAAAAAAGAGAAGAAAGAAT
TGCCCGGCAAAAAAAAGACCTTCTACCCTTTCGCAAAAAAGAAATGAGATCGTTTGCTGCAATCCAA
AAAAAAATAGAGAGTCCCCAATGTTCGGAATACGTCAAAAAAATCTTAGTGCGCTTAAATTAA
CAGACTCTCTCCCGGGAAGGTGGACATGCAGAACCTACCAAAAAAAAAGAGAAGAAAGAAT
TGCCCGGCAAAAAAAAGACCTTCTACCCTTTCGCAAAAAAGAAATGAGATCGTTTGCTGCAATCCAA
AAAAAAAAGACCATGAAATAATTTTCTGGATCATCCATACAGAACCAAAAAAAAGAGGTG
4
Fast Exact String Pattern-matching Algorithms Adapted to the Characteristics of the Medical Language

Christian Lewis, MD and Robert H. Baud, PhD

Affiliations of the authors: Puget Sound Health Care System, Seattle, Washington (CL); University Hospital of Geneva, Geneva, Switzerland (RHB).

Correspondence and reprints: Christian Lewis, MD, University Hospital of Geneva, Division of Medical Informatics, Rue Micheli-du-Crest, CH-1211 Geneva 4, Switzerland. E-mail: christian.lewis@huggeo.ch.

Received October 26, 1999; Accepted February 16, 2000.

Abstract

Objective: The authors consider the problem of exact string pattern matching using algorithms that do not require any preprocessing. To choose the most appropriate algorithm, distinctive features of the medical language must be taken into account. The characteristics of medical language are emphasized in this regard; the best algorithm of those reviewed is proposed, and detailed evaluations of time complexity for processing medical texts are provided.

Design: The authors first illustrate and discuss the techniques of various string pattern-matching algorithms. Next, the source code and the behavior of representative exact string pattern-matching algorithms are presented in a comprehensive manner to promote their implementation. Detailed explanations of the use of various techniques to improve performance are given.

Measurements: Real-time measures of time complexity with English medical texts are presented. They lead to results distinct from those found in the computer science literature, which are typically computed with normally distributed texts.

Results: The Horning algorithm achieves the best overall results when used with medical texts. This algorithm usually performs at least twice as fast as the other algorithms tested.
Variants of the problem allow wildcards in the pattern and/or the text. Exact matching is when no wildcards are allowed. We describe the fastest sequential algorithm for solving the exact string searching problem. The algorithm is called the Boyer-Moore fast string searching algorithm.
Example

Find the word “comedy” in this *NY Times* article:

Fred Armisen’s office at “Saturday Night Live” is deceptively small, barely big enough to fit a desk, a couch, and an iPod. The glorified closet, the subject of a running joke on the comedy show, now in its 31st season, can simultaneously house a wisecracking . . .
COMEDY

JOKE ON THE COMEDY
COMEDY

JOKE ON THE COMEDY
COMEDY

JOKE ON THE COMEDY
J O K E  O N  T H E  C O M E D Y
COMEDY

JOKE ON THE COMEDY
COMEDY

JOKE ON THE COMEDY
COMEDY

JOKE ON THE COMEDY
COMEDY

JOKE ON THE COMEDY
COMEDY

JOKE ON THE COMEDY
COMEDY

JOKE ON THE COMEDY
COMEDY
COMEDY

JOKE ON THE COMEDY
COMEDY

JOKE ON THE COMEDY
COMEDY

JOKE ON THE COMEDY
COMEDY

JOKE ON THE COMEDY
COMEDY

JOKE ON THE COMEDY
COMEDY

JOKE ON THE COMEDY
COMEDY

JOKE ON THE COMEDY
COMEDY

JOKE ON THE COMEDY
COMEDY

Joke on the Comedy
COMEDY

JOKE ON THE COMEDY
Key Property: The longer the pattern, the faster the search!
Pre-Computing the Skip Distance

pat: 543210
    COMEDY
txt: xxxxxx0xxxxxxxxxxxx...
    ↑

A 6  F 6  K 6  P 6  U 6
B 6  G 6  L 6  Q 6  V 6
C 5  H 6  M 3  R 6  W 6
D 1  I 6  N 6  S 6  X 6
E 2  J 6  O 4  T 6  Y 0
    Z 6

This is a 1-dimensional array, \texttt{skip[c]}, as big as the alphabet.
COMEDY

JOKE ON THE COMEDY

skip[c]:
A 6  F 6  K 6  P 6  U 6
B 6  G 6  L 6  Q 6  V 6
C 5  H 6  M 3  R 6  W 6
D 1  I 6  N 6  S 6  X 6
E 2  J 6  O 4  T 6  Y 0
               Z 6
COMEDY

JOKE ON THE COMEDY

skip[c]:
A 6  F 6  K 6  P 6  U 6
B 6  G 6  L 6  Q 6  V 6
C 5  H 6  M 3  R 6  W 6
D 1  I 6  N 6  S 6  X 6
E 2  J 6  O 4  T 6  Y 0
    Z 6
COMEDY

JOKE ON THE COMEDY

skip[c]:
A 6 F 6 K 6 P 6 U 6
B 6 G 6 L 6 Q 6 V 6
C 5 H 6 M 3 R 6 W 6
D 1 I 6 N 6 S 6 X 6
E 2 J 6 O 4 T 6 Y 0
            Z 6
COMEDY

O

JOKE ON THE COMEDY

skip[c]:

A 6  F 6  K 6  P 6  U 6
B 6  G 6  L 6  Q 6  V 6
C 5  H 6  M 3  R 6  W 6
D 1  I 6  N 6  S 6  X 6
E 2  J 6  O 4  T 6  Y 0
          Z 6
JOKE ON THE COMEDY

skip[c]:
A 6  F 6  K 6  P 6  U 6
B 6  G 6  L 6  Q 6  V 6
C 5  H 6  M 3  R 6  W 6
D 1  I 6  N 6  S 6  X 6
E 2  J 6  O 4  T 6  Y 0
        Z 6
COMEDY

JOKE ON THE COMEDY

skip[c]:
A 6   F 6   K 6   P 6   U 6
B 6   G 6   L 6   Q 6   V 6
C 5   H 6   M 3   R 6   W 6
D 1   I 6   N 6   S 6   X 6
E 2   J 6   O 4   T 6   Y 0
     Z 6
skip[c]:
A 6  F 6  K 6  P 6  U 6
B 6  G 6  L 6  Q 6  V 6
C 5  H 6  M 3  R 6  W 6
D 1  I 6  N 6  S 6  X 6
E 2  J 6  O 4  T 6  Y 0
    Z 6
COMEDY

JOKE ON THE COMEDY

skip[c]:
A 6  F 6  K 6  P 6  U 6
B 6  G 6  L 6  Q 6  V 6
C 5  H 6  M 3  R 6  W 6
D 1  I 6  N 6  S 6  X 6
E 2  J 6  0 4  T 6  Y 0
                Z 6
JOKE ON THE COMEDY

skip[c]:
A 6  F 6  K 6  P 6  U 6
B 6  G 6  L 6  Q 6  V 6
C 5  H 6  M 3  R 6  W 6
D 1  I 6  N 6  S 6  X 6
E 2  J 6  O 4  T 6  Y 0  Z 6
J O K E  O N  T H E  C O M E D Y

skip[c]:
A 6  F 6  K 6  P 6  U 6
B 6  G 6  L 6  Q 6  V 6
C 5  H 6  M 3  R 6  W 6
D 1  I 6  N 6  S 6  X 6
E 2  J 6  O 4  T 6  Y 0
               Z 6
skip[c]:
A 6  F 6  K 6  P 6  U 6
B 6  G 6  L 6  Q 6  V 6
C 5  H 6  M 3  R 6  W 6
D 1  I 6  N 6  S 6  X 6
E 2  J 6  O 4  T 6  Y 0  Z 6
But Wait! There’s More!

pat: NONPARTIPULAR

txt: ------------------------

|
But Wait! There’s More!

\textit{pat}: \textsc{NONPARTIPULAR}

\textit{txt}: \underline{-------------R----------}
But Wait! There’s More!

$\text{pat: NONPARTIPULAR}$

$\text{txt: } ------A---------}$

|
But Wait! There’s More!

\textit{pat}: NONPARTIPULAR
\textit{txt}: ------------P-------------
  |
But Wait! There’s More!

\[pat:\]  \text{NONPARTIPULAR}
\[txt:\]  \underline{---------P---------}

\[\mid\]

Slide 2 to match the discovered character.
But Wait! There’s More!

\[ \text{pat: } \text{NONPARTIPULAR} \]
\[ \text{txt: } \text{----------P??----------} \]
\[ \text{ | } \]
But Wait! There’s More!

\textit{pat:} \text{NONPARTIPULAR}
\textit{txt:} \text{---------PAR---------}
\text{ | }
But Wait! There’s More!

\[ \text{pat: NONPARTIPULAR} \]
\[ \text{txt: } \quad \text{-----------------------} \]
\[ \text{ } \quad \text{|} \]
But Wait! There’s More!

\textit{pat}: NONPARTIPULAR \textcolor{red}{R}

\textit{txt}: --------------- \textcolor{red}{R}--------------

|
But Wait! There’s More!

\textit{pat}: NONPARTIPULAR\textcolor{green}{AR}

\textit{txt}: \underline{-----------AR----------

|}
But Wait! There’s More!

pat: NONPARTIPLEAR

txt: -----------PAR----------
    |
But Wait! There’s More!

pat: NONPARTIPULAR

txt: ---------------PAR-------------
    |
But Wait! There’s More!

\text{pat:} \quad \text{NONPARTIPULAR}
\text{txt:} \quad \cdots\text{PAR}\cdots

Slide 7 to match the \textit{discovered substring}!
\[
\begin{array}{c|c|c|c}
  j & |pat| & \text{pat: NONPARTIPULAR} \\
  \hline
  \hline
  i & \text{txt: \underline{---PAR---}} & \\
  \hline
  i & \text{dt: } txt[i] \; pat[j + 1] \; \ldots \; pat[|pat|] & P \quad A \quad R
\end{array}
\]
dt: $txt[i]$ $pat[j + 1]$ ... $pat[|pat|]$

dt can be computed given $txt[i]$ and index $j$ in $pat$!

There are only $|\alpha| \times |pat|$ combinations, where $|\alpha|$ is the alphabet size.
The Skip Distance – Delta

Given $pat$, the skip can be pre-computed for every combination of character read, $c$, and pattern index, $j$, by finding how far we must slide to find the last occurrence of $dt$ in $pat$. 
\textit{pat}: NON\textcolor{red}{\textsc{PARTIPULAR}}

\textit{txt}: \__________\textcolor{red}{\textsc{PAR}}\__________
   |
pat: NONPARTIPULAR

txt: ------------PAR------------
pat: BC-ABC-BBC-CBC

txt: ---------------BBC-------------
pat:   BC–ABC–BBC–CBC

txt:   ---------------BBC---------------
pat: BC-ABC-BBC-CBC

txt: ------------ABC-------------
     |
pat:        BC-ABC-BBC-CBC

txt:  ---------------ABC-------------
     |
$pat$: BC-ABC-BBC-CBC
$\text{txt}$: -----------DBC-----------
pat: BC-ABC-BBC-CBC

txt: ---------------DBC---------------

|
$pat: \text{EE-ABC-BBC-CBC}$

txt: ------------------DBC-------------------
       |

68
pat: EE-ABC-BBC-CBC

txt: --------------DBC--------------
    |

69
The Delta Array

delta[c, j] is an array of size $|\alpha| \times |\text{pat}|$ that gives the skip distance when a mismatch occurs after comparing $c$ from $\text{txt}$ to $\text{pat}[j]$. 
The Algorithm

\text{fast}(\text{pat}, \text{txt})

\textbf{If} \text{ pat} = ""
  \textbf{then}
    \textbf{If} \text{ txt} = ""
      \textbf{then return Not-Found;}
    \textbf{else return 0; end;}
  \textbf{end;}

preprocess $pat$ to produce $delta$;

\[
j := |pat| - 1;
\]
\[
i := j;
\]
while (0 ≤ j ∧ i < |txt|) do
  If pat[j] = txt[i] then
    i := i − 1;
    j := j − 1;
  else
    i := i + delta[txt[i], j];
    j := |pat| − 1;
  end;
If \((j < 0)\)
    then return \(i + 1\);
else return Not-Found; end;
end;
Performance

How does the algorithm perform?

This depends on the size of the alphabet. We only have data on English text right now.

In our test:

txt: English text of length 177,985.
pat: 100 randomly chosen patterns of length 5 – 30, chosen from another English text and filtered so they do not occur in the search text.

The naive string searching algorithm would look at all 177,985 characters of the search text. In fact, it would look at some characters more than once.