String Matching (14)

John Reif

November 12th, 1998

1 STRING MATCHING

1.1 Matching Strings

see the seven seas seashells by the seashore.
Find all occurrences of *seas*. Common problem to lots of applications:

- UNIX grep
- Word processors
- Web searching
- DNA

1.2 Alphabets

The alphabet is the set of symbols Σ.

- English $\Sigma = \{a, b, c, d, ..., x, y, z\}$
- ASCII 256 symbols, UNICODE 65536 symbols
- DNA $\Sigma = \{A, C, G, T\}$

Set of all strings: Σ^*
Size of alphabet: $|\Sigma|$

1.3 Notation

Let P be a string in Σ^*.
$P[i..j]$: Array range notation character i to j inclusive indexed from 1 to length of string.
$|P|$ denotes the length of string P.
AB is the concatenation of strings A and B.
$Q \sqsubset R$: the string Q is a prefix of R. $R = QS$ for some S.

$$\begin{aligned} Q &= \texttt{abc} \\ R &= \texttt{abcbbca} \\ S &= \texttt{bbca} \end{aligned}$$

$Q \sqsupset R$: the string Q is a suffix of R. $R = SQ$ for some S.

$$\begin{aligned} Q &= \texttt{bca} \\ R &= \texttt{abcbbca} \\ S &= \texttt{abcb} \end{aligned}$$

1.4 The String Matching Problem

Given a pattern string $P \in \Sigma^*$, with length $|P| = m$ and a text string $T \in \Sigma^*$, with length $|T| = n$, where $m, n > 0$ and $m \leq n$:
If P occurs as a substring of T, find the first occurrence—that is, find s such that $T[s + 1..s + m] = P[1, m]$ $(0 \leq s \leq n - m)$.

1.5 Extensions

Other related problems:

- Find all matches
- Find multiple patterns
- Wildcard matching
- Formatting

2 NAIVE PATTERN MATCHER

2.1 Examples

First thing you'd try:

$$\begin{aligned} P &= \texttt{abc} \\ T &= \texttt{abaccbabacbaabcabacabacccaabbca}\ldots \\ . &= \texttt{abc} \\ . &= \ \texttt{abc} \\ . &= \ \ \texttt{abc} \\ . &= \ \ \ \texttt{abc}\ldots \end{aligned}$$

Test the pattern at every position of the text.

2.2 Algorithm

```
NAIVE-PATTERN-MATCHER(T, P)
n ← length[T]
m ← length[P]
for s ← 0 to n − m
    do j ← 1
       while j ≤ m and P[j] = T[s + j]
           do j ← j + 1
       if j > m
          then print "Pattern occurs at shift" s
```

2.3 Analysis

Remember that $|P| = m$, $|T| = n$.
Inner loop will take m steps to confirm the pattern matches
Outer loop will take $n - m + 1$ steps
Therefore, worst case is $\Theta((n - m + 1)m)$

2.4 More Analysis

When does the worst case occur?

$$T = \texttt{aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa} \ldots. \quad P = \texttt{aaaaaa}.$$

When does the best case occur?

$$T = \texttt{aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa} \ldots. \quad P = \texttt{bbbbbb}.$$

2.5 Intuition for Improvement

When could the Naive Algorithm do better?

$$\begin{aligned} T &= \texttt{aaaaaaaabaaaaaaaaaaaaaaaaaaaaaa} \ldots \\ &= \texttt{aaaaaaaaa} \\ &= \ \texttt{aaaaaaaaa} \\ &= \ \ \texttt{aaaaaaaaa} \\ &= \ \ \ \texttt{aaaaaaaaa} \ldots \end{aligned}$$

Use information that is gathered in comparisons to motivate larger shifts than 1 (e.g., here we'd like to shift $|P|$)

3 BOYER-MOORE

3.1 Naive Algorithm Again

```
NAIVE-PATTERN-MATCHER-2(T, P)
n ← length[T]
m ← length[P]


s ← 0
while s ≤ n − m
    do j ← m
        while j > 0 and P[j] = T[s + j]
            do j ← j − 1
        if j = 0
            then print "Pattern occurs at shift" s
                 s ← s + 1
            else s ← s + 1
```

Same algorithm as before, but check P right-to-left instead.
When would this alone help?

When the last character in the pattern is rare. Fast jumps!

3.2 Boyer-Moore Algorithm

```
BOYER-MOORE-MATCHER(T, P, Σ)
n ← length[T]
m ← length[P]
λ ← COMPUTE-LAST-OCCURRENCE-FUNCTION(P, m, Σ)
γ ← COMPUTE-GOOD-SUFFIX-FUNCTION(P, m)
s ← 0
while s ≤ n − m
    do j ← m
        while j > 0 and P[j] = T[s + j]
            do j ← j − 1
        if j = 0
            then print "Pattern occurs at shift" s
                 s ← s + γ[0]
            else s ← s + max(γ[j], j − λ[T[s + j]])
```

Heuristics must have the property that they don't miss any matches.

3.3 Bad-Character Heuristic

In the case of a mismatch, $P[j] \neq T[s+j]$:
Look for the occurrence of mismatch character in the pattern, such that $P[k] = T[s+j]$.
Three cases:
$k = 0$: The character occurs nowhere in the pattern

```
aaac........
aaaaaa
    aaaaaa
```

$k < j$: The rightmost occurrence of the bad character is to the left of j

```
aaacaa........
acaaaa
  acaaaa
```

$k > j$: The rightmost occurrence of the bad character is to the right of j

```
..aaacac........
  aaaaac
aaaaac
```

3.4 Bad-Character Heuristic Analysis

In every case, we propose a shift of $j - k$. Negative values occur for case 3, but the other heuristic is always greater than 0, so the max guarantees progress.
Define $\lambda[a]$ to be the index of the right-most index of a in P. If $a \notin P$ then $\lambda[a] = 0$.
Then $j - \lambda[T[s+j]]$ is the bad-character shift we want. (j is current offset from s in T, $T[s+j]$ is the current character.)

3.5 Bad-Character Heuristic Algorithm

Compute the λ mapping.
Compute-Last-Occurrence-Function(P, m, Σ)

for each $a \in \Sigma$
  do $\lambda[a] = 0$
for $j \leftarrow 1$ **to** m
  do $\lambda[P[j]] \leftarrow j$

Running time is $O(|\Sigma| + m)$.

3.6 When does it help?

How much can the Bad-Character Heuristic help?
How likely is this in English?

3.7 Good-Suffix Heuristic

Use the fact that we know that a suffix of the pattern P matched:

```
T = abbbaabab
            ||
P =    cabaab
P =       cabaab
```

Propose to shift the pattern to the right by the least amount that guarantees not to skip any occurrence of the good-suffix already matched.
Define γ to be the shift proposed by the good-suffix heuristic.

3.8 Prefix Function

P_k refers to the first k characters of P (the k length prefix of P).
$P_m = P$

$$\begin{aligned} P &= \texttt{aababddc} \\ P_1 &= \texttt{a} \\ P_2 &= \texttt{aa} \\ P_3 &= \texttt{aab} \end{aligned}$$

etc.
Given a pattern $P[1, m]$, define the prefix-function π:
$\pi[j] = \max\{k : k < j \text{ and } P_k \sqsupset P_j\}$
Example.

3.9 Computing the Prefix Function

With amortized analysis, can show that computing the prefix-function is $O(m)$.
For more informaiton, see the book, pages 871–874.

3.10 Similarity Relation

Define $Q \sim R$ ("Q is similar to R") to mean $Q \sqsubset R$ *or* $R \sqsubset Q$.

```
R =      bcba
Q = bbbbbbcba
```

If $\sim$, can align strings at right and some suffix will match.

3.11 Good-Suffix Heuristic

Propose to shift the pattern to the right by the least amount that guarantees not to skip any occurrence of the good-suffix already matched.

```
T = abbbaabab
          ||
P =     cabaab
P =         cabaab
```

$\gamma[j] = m - \max\{k : 0 \leq k < m \text{ and } P[j+1..m] \sim P_k\}$.
In example above, $j = 4$, want maximum k such that $P[5..6] = \texttt{ab} \sim P_k$.

$$\begin{aligned} P_1 &= \texttt{c} \\ P_2 &= \texttt{ca} \\ P_3 &= \texttt{cab} \\ P_4 &= \texttt{caba} \\ P_5 &= \texttt{cabaa} \end{aligned}$$

So, $\gamma[4] = 3$.

3.12 Good-Suffix Heuristic Algorithm

The resulting definition of the Good-Suffix Heuristic:
$\gamma[j] = \min(\{m - \pi[m]\} \cup \{l - \pi' : 1 \leq l \leq m \text{ and } j = m - \pi'[l]\})$
Implentation of this defintion:
Compute-Good-Suffix-Function(P, m)
$\pi \leftarrow$ Compute-Prefix-Function(P)
$P' \leftarrow$ reverse(P)
$\pi' \leftarrow$ Compute-Prefix-Function(P')
for $j \leftarrow 0$ **to** m
  do $\gamma[j] \leftarrow m - \pi[m]$
for $l \leftarrow 1$ **to** m
  do $j \leftarrow m - \pi'[l]$
    if $\gamma[j] > l - \pi'[l]$
      then $\gamma[j] \leftarrow l - \pi'[l]$
return

3.13 Analysis of Compute-Good-Suffix-Function

Compute-Prefix-Function takes $O(m)$ and two loops both of which are $O(m)$. Therefore, Compute-Good-Suffix-Function is $O(m)$

3.14 Analysis of Boyer-Moore

Compute-Last-Occurrence-Function is $O(m + |\Sigma|)$
Compute-Good-Suffix-Function is $O(m)$
so Boyer-Moore-Matcher is $O((n - m + 1)m + |\Sigma|)$
So, theoretically no better than Naive (actually a little worse), but in practice it seems to do better.

4 STRING MATCHING WITH FINITE AUTOMATA

4.1 Finite Automata

A finite automaton M has:

- Q is a finite set of states
- $q_0 \in Q$ is the start state
- $A \subseteq Q$ is the set of accepting states
- Σ is the finite input alphabet
- δ is the transition function. Maps a state and an input symbol to another state.

4.2 Example

A finite automaton begins in the start state, reads input symbols, making transitions based on these symbols. When the input sequence terminates, if it is in an accepting state, the automaton is said to *accept* that input string, otherwise the input is *rejected*.
Define a function ϕ called the final-state function, which $\phi(w)$ is the state that M is in after processing w.

4.3 The Suffix-Function

Given a pattern $P[1..m]$, define the suffix-function σ such that $\sigma(x)$ is the length of the longest prefix of P that is also a suffix of x:
$\sigma(x) = \max \{k : P_k \sqsupset x\}$

$$\begin{aligned} P &= \texttt{abaabc} \\ P_1 &= \texttt{a} \\ P_2 &= \texttt{ab} \\ P_3 &= \texttt{aba} \\ P_4 &= \texttt{abaa} \\ \sigma(\texttt{abbaba}) &= \texttt{aba} \end{aligned}$$

4.4 String-Matching Automata

For a given pattern P, we can define its string-matching automata:

- $Q = \{0, \ldots, m\}$ (states)
- $q_0 = 0$ (start state)
- $A = \{m\}$ (accepting state)
- $\delta(q, a) = \sigma(P_q a)$

The transition function chooses the next state to maintain the invariant:
$\phi(T_i) = \sigma(T_i)$
after scanning in the first i characters, the state number is the longest prefix of P that is also a suffix of T_i.

4.5 Finite-Automaton-Matcher

```
FINITE-AUTOMATON-MATCHER(T, δ, m)
n ← length[T]
q ← 0
for i ← 1 to n
    do q ← δ(q, T[i])
       if q = m
          then s ← i − m
               print "Pattern occurs at shift" s
```

This is clearly $O(n)$ since we take one step for each symbol of T.
So, this seems better than the others? Why not use this?

4.6 Computing the transition function

This can be done according to its definition, and takes $O(m^3|\Sigma|)$.
More clever versions can achieve $O(m|\Sigma|)$.
This brings the total running time for Finite-Automaton-Matcher to $O(n + m|\Sigma|)$.
Actually, not too bad if pattern is small relative to the size of the text.