String Matching: Boyer-Moore Algorithm

Greg Plaxton
Theory in Programming Practice, Fall 2005
Department of Computer Science
University of Texas at Austin

The (Exact) String Matching Problem

- Given a text string t and a pattern string p, find all occurrences of p in t

Three Efficient String Matching Algorithms

- Rabin-Karp
  - This is a simple randomized algorithm that tends to run in linear time in most scenarios of practical interest
  - The worst case running time is as bad as that of the naive algorithm, i.e., $\Theta(\overline{p} \cdot \overline{t})$
- Knuth-Morris-Pratt
  - The worst case running time of this algorithm is linear, i.e., $O(\overline{p} + \overline{t})$
- Boyer-Moore (this lecture and the next)
  - This algorithm tends to have the best performance in practice, as it often runs in sublinear time
  - The worst case running time is as bad as that of the naive algorithm

Boyer-Moore String Matching Algorithm

- At any moment, imagine that the pattern is *aligned* with a portion of the text of the same length, though only a part of the aligned text may have been matched with the pattern
- Henceforth, alignment refers to the substring of t that is aligned with p and l is the index of the left end of the alignment; i.e., $p[0]$ is aligned with $t[l]$ and, in general, $p[i]$, $0 \leq i < m$, with $t[l + i]$
- Whenever there is a mismatch, the pattern is *shifted* to the right, i.e., l is increased, as explained in the following sections

Algorithm Outline

- The overall structure of the program is a loop that has the invariant Q1: Every occurrence of p in t that starts before l has been recorded

- The following loop records every occurrence of p in t eventually

  $l := 0$;
  { Q1 }
  loop
     { Q1 }
     "increase l while preserving Q1"
  endloop

The Variable j

- Next, we show how to increase l while preserving Q1

- We introduce variable j, $0 \leq j < m$, with the meaning that the suffix of p starting at position j matches the corresponding portion of the alignment

  Q2: $0 \leq j \leq m$, $p[j..m] = t[l + j..l + m]$

- Thus, the whole pattern is matched when $j = 0$, and no part has been matched when $j = m$

A Refinement of the Previous Algorithm

- We establish Q2 by setting j to m
- We match the symbols from *right to left* of the pattern until we find a mismatch or the whole pattern is matched

$j := m;$
$\{$ Q2 $\}$
while $j > 0 \ \wedge \ p[j-1] = t[l+j-1]$ **do** $j := j-1$ **endwhile**
$\{$ Q1 $\wedge$ Q2 $\wedge \ (j = 0 \ \vee \ p[j-1] \neq t[l+j-1]) \ \}$
if $j = 0$
  then $\{$ Q1 $\wedge$ Q2 $\wedge \ j = 0 \ \}$ record a match at l; $l := l'$ $\{$ Q1 $\}$
  else $\{$ Q1 $\wedge$ Q2 $\wedge \ j > 0 \ \wedge \ p[j-1] \neq t[l+j-1] \ \}$ $l := l''$$\{$ Q1 $\}$
endif
$\{$ Q1 $\}$

- How do we compute l' and l''?

Computation of l'

- This turns out to be essentially a special case of the computation of l''
- So we focus primarily on the computation of l'' in the presentation that follows

Computation of l''

- The precondition for the computation of l'' is,

$$\text{Q1} \wedge \text{Q2} \wedge j > 0 \;\wedge\; p[j-1] \neq t[l+j-1].$$

- We consider two heuristics, each of which can be used to calculate a value of l''; the greater value is assigned to l
  - The first heuristic, called the *bad symbol heuristic*, exploits the fact that we have a mismatch at position $j-1$ of the pattern
  - The second heuristic, called the *good suffix heuristic*, uses the fact that we have matched a (possibly empty) suffix of p with the suffix of the alignment, i.e., $p[j..m] = t[l+j..l+m]$.

The Bad Symbol Heuristic: Easy Case

- Suppose we have the pattern “attendance” that we have aligned against a portion of the text whose suffix is “hce”, as shown below

text	-	-	-	-	-	-	-	h	c	e								
pattern	a	t	t	e	n	d	a	n	c	e								
align									a	t	t	e	n	d	a	n	c	e

- The suffix “ce” has been matched; the symbols ’h’ and ’n’ do not match

- There is no ’h’ in the pattern, so no match can include this ’h’ of the text

- Hence, the pattern may be shifted to the symbol following ’h’ in the text, as shown by *align* above

The Bad Symbol Heuristic: The More Interesting Case

- Next, suppose the mismatched symbol in the text is 't', as shown below

text	-	-	-	-	-	-	-	t	c	e
pattern	a	t	t	e	n	d	a	n	c	e

- There are two ways to align the 't' in the pattern with a 't' in the text

text	-	-	t	c	e	-	-	-	-	-	-
align1	a	t	t	e	n	d	a	n	c	e	
align2		a	t	t	e	n	d	a	n	c	e

- Which alignment should we choose?

Minimum Shift Rule

- Rule: Shift the pattern by the minimum allowable amount
- Justification: Preservation of Q1
  - We never skip over a possible match following this rule, because no smaller shift yields a match at the given position, and, hence no full match
- So, in the example of the previous slide, we should use $align1$

Motivation for the Minimum Shift Rule: Example

- In this example, the leftmost symbol 'y' of the pattern "xxy" fails to match the text symbol 'x'

text	-	-	x	-	-
pattern	x	x	y		
align1		x	x	y	
align2			x	x	y

- If we were to advance to alignment *align2*, we might skip a match in position in *align1*, violating invariant Q1

Bad Symbol Heuristic: Implementation

- For each symbol in the alphabet, we precalculate its rightmost position in the pattern
- if the mismatched symbol's rightmost occurrence in the pattern is at $p[k]$, then $p[0]$ is aligned with $t[l-k+j-1]$, or l is increased by $-k+j-1$
- For a nonexistent symbol in the pattern, like 'h', we set its rightmost occurrence to -1 so that l is increased to $l+j$, as required
- Note that the shift $-k+j-1$ is negative if $k > j-1$, which can easily occur
  - But the good suffix heuristic always yields a positive increment for l, so the maximum of these two increments is positive

The Good Suffix Heuristic

- Suppose we have a pattern "abxabyab" of which we have already matched the suffix "ab", but there is a mismatch with the preceding symbol 'y', as shown below

text	-	-	-	-	-	z	a	b	-	-
pattern	a	b	x	a	b	y	a	b		

- Then, we shift the pattern to the right so that the matched part is occupied by the same symbols, "ab"; this is possible only if there is another occurrence of "ab" in the pattern

Case 1: The Matched Suffix Occurs Elsewhere in the Pattern

- For the pattern of the previous slide, the matched portion "ab" occurs in two other places
- Thus there are two possible alignments to consider, as shown below

text	-	-	z	a	b	-	-	-	-	-	-
align1	a	b	x	a	b	y	a	b			
align2				a	b	x	a	b	y	a	b

- By the minimum shift rule, we select *align1*

Case 2: The Matched Suffix Does Not Occur Elsewhere

- No complete match of the suffix s is possible if s does not occur elsewhere in p

- This possibility is shown in the example below, where s is "xab"

text	-	y	x	a	b	-	-	-
pattern	a	b	x	a	b			
align				a	b	x	a	b

- In this case, the best that can be done is to match with a suffix of "xab" that is also a prefix of p

- In the example above, "ab" is a suffix of s (and hence also a suffix of p) that is also a prefix of p

Good Suffix Heuristic

- Let s denote the matched suffix and let

$$R = \{r \text{ is a proper prefix of } p \wedge \\ (r \text{ is a suffix of } s \vee s \text{ is a suffix of } r)\}$$

- The good suffix heuristic aligns an r in R with the end of the previous alignment

- According to the minimum shift rule, the amount $b(s)$ by which the pattern is shifted is

$$b(s) = \min\{\overline{p} - \overline{r} \mid r \in R\}$$

- Next time we will develop an efficient algorithm for computing $b(s)$

Updating l: Summary

- In the algorithm outlined earlier, we have two assignments to l
  - $l := l'$, when the whole pattern has matched
  - $l := l''$, when $p[j..\overline{p}] = t[l + j..l + \overline{p}]$ and $p[j - 1] \neq t[l + j - 1]$
- These assignments are implemented as follows
  - $l := l'$ is implemented by $l := l + b(p)$
  - $l := l''$ is implemented by $l := l + \max(b(s), j - 1 - rt(h))$, where $s = p[j..\overline{p}]$, $h = t[l + j - 1]$, and $rt(h)$ is the index of the rightmost occurrence of h in p (or -1 if h does not occur in p)