Class 18

BOYER-MOORE STRING MATCHING

HASHING

Boyer-Moore Algorithm

Shifts come in two varieties:

- Bad-symbol shift
  - similar to shift in Horpool's, but the bad character determining the shift is where we found the mismatch, not the last character in pattern
- Good-suffix shift
  - takes advantage of when we've matched some suffix of the pattern
- Take maximum of possible shifts!

Boyer-Moore Algorithm

Horspool is a simplification of Boyer-Moore.

However, Case 1 of Horspool same as Boyer-Moore:

Look at character *c*:

$s_0 \quad \ldots \quad c \quad \ldots \quad s_{n-1}$

B A R B E R

Case 1: No *c*'s in the pattern. Suppose *c* is the letter S:

$s_0 \quad \ldots \quad$ S $\quad \ldots \quad s_{n-1}$

$\not\parallel$

B A R B E R

B A R B E R

Shift pattern by its entire length.

Boyer-Moore Algorithm

Boyer-Moore is different in all other cases, i.e. if at least one character is a match.

Consider:

s_0	$\ldots$	c	s_{i-k+1}	$\ldots$	s_i	$\ldots$	s_{n-1}	text
		$\not\parallel$	$\parallel$		$\parallel$			
p_0	$\ldots$	p_{m-k-1}	p_{m-k}	$\ldots$	p_{m-1}			pattern

Boyer-Moore determines shift size by considering two quantities:

1. Character *c* from text that caused mismatch.
2. $t_1(c) - k$, where $t_1(c)$ is the entry in the shift-table.

s_0	$\ldots$	c	s_{i-k+1}	$\ldots$	s_i	$\ldots$	s_{n-1}	text
		$\not\parallel$	$\parallel$		$\parallel$			
p_0	$\ldots$	p_{m-k-1}	p_{m-k}	$\ldots$	p_{m-1}			pattern
			p_0	$\ldots$		p_{m-1}		

$s_0 \; \cdots \; c \; s_{i-k+1} \; \cdots \; s_i \; \cdots \; s_{n-1}$ text

$p_0 \; \cdots \; p_{m-k-1} \; p_{m-k} \; \cdots \; p_{m-1}$ pattern

$p_0 \; \cdots \; p_{m-1}$

Boyer-Moore Algorithm

Consider the following example in which *c* does not appear in the pattern:

```
s0   ...        S E R            ...  sn-1
                ≠ | |
          B A R B E R
                    B A R B E R
```

We shifted by $t_1(S) - k$.

Using the Horspool shift-table from last time, this is: 6-2 = 4.

Incidentally, this is called a *bad-symbol shift*.

$s_0 \; \cdots \; c \; s_{i-k+1} \; \cdots \; s_i \; \cdots \; s_{n-1}$ text

$p_0 \; \cdots \; p_{m-k-1} \; p_{m-k} \; \cdots \; p_{m-1}$ pattern

$p_0 \; \cdots \; p_{m-1}$

Boyer-Moore Algorithm

The same formula from the prior slide can also be used when the mismatched character *c* occurs in the pattern, provided $t_1(S) - k > 0$

```
s0   ...      A E R          ...  sn-1
              ≠ | |
        B A R B E R
            B A R B E R
```

We can shift by $t_1(A) - k$.

Using the Horspool example from last time, this is: 4-2 = 2.

If $t_1(S) - k <= 0$, then shift by one character to the right.

Boyer-Moore Algorithm

Let *suff(k)* be the ending portion of the pattern.

Let *k* be the length.

Consider the case when there is another occurrence of *suff(k)* in the pattern, not preceded by the same character as its rightmost occurrence.

If the pattern were to be preceded by the same character, we would have the same fail.

k	**pattern**	d_2
1	$ABC\overline{B}A\underline{B}$	2
2	$\overline{AB}CB\underline{AB}$	4

Boyer-Moore Algorithm

k	**pattern**	d_2
1	$ABC\overline{B}A\underline{B}$	2
2	$\overline{AB}CB\underline{AB}$	4

In this case, we can shift the pattern by the distance d_2 between the second rightmost occurrence of *suff(k)* and its rightmost occurrence.

In the above example, those distances k=1 is 2 and for k=2 is 4.

This type of shift is called a *good-suffix shift*.

Boyer-Moore Algorithm

Suppose there is **no** other occurrence of *suff(k)* in the pattern not preceded by the same character as its rightmost occurrence.

In <u>most</u> cases, shift pattern by its entire length:

```
s0   ...      c  B  A  B                ...  s_{n-1}
              ∦  ||  ||  ||
        D  B  C  B  A  B
                             D  B  C  B  A  B
```

Counterexample:

```
s0   ...      c  B  A  B  C  B  A  B    ...  s_{n-1}
              ∦  ||  ||  ||
        A  B  C  B  A  B
                             A  B  C  B  A  B
```

Boyer-Moore Algorithm

Problem: ABCBAB has the same pattern as prefix as its suffix.

To avoid such an erroneous shift based on suffix of size *k*, for which there is no other occurrence in the pattern not preceded by the same character as in its rightmost occurrence, we need to find the longest prefix size $l < k$ that matches the suffix of the same size *l*.

Is such a prefix exists the shift size d_2 is computed as the distance between this prefix and the corresponding suffix.

Otherwise, d_2 is set to the pattern's length *m*.

Boyer-Moore Algorithm

Here is a good-suffix table for the pattern ABCBAB:

k	**pattern**	d_2
1	ABCBAB	2
2	ABCBAB	4
3	ABCBAB	4
4	ABCBAB	4
5	ABCBAB	4

Boyer-Moore Algorithm

The Boyer-Moore algorithm

Step 1 For a given pattern and the alphabet used in both the pattern and the text, construct the bad-symbol shift table as described earlier.

Step 2 Using the pattern, construct the good-suffix shift table as described earlier.

Step 3 Align the pattern against the beginning of the text.

So far so good.

Boyer-Moore Algorithm

Step 4 Repeat the following step until either a matching substring is found or the pattern reaches beyond the last character of the text. Starting with the last character in the pattern, compare the corresponding characters in the pattern and the text until either all m character pairs are matched (then stop) or a mismatching pair is encountered after $k \geq 0$ character pairs are matched successfully. In the latter case, retrieve the entry $t_1(c)$ from the c's column of the bad-symbol table where c is the text's mismatched character. If $k > 0$, also retrieve the corresponding d_2 entry from the good-suffix table. Shift the pattern to the right by the number of positions computed by the formula

$$d = \begin{cases} d_1 & \text{if } k = 0, \\ \max\{d_1, d_2\} & \text{if } k > 0, \end{cases}$$

where $d_1 = \max\{t_1(c) - k, 1\}$.

Boyer-Moore Example

```
B E S S _ K N E W _ A B O U T _ B A O B A B S
B A O B A B
```

$d_1 = t_1(K) - 0 = 6$ B A O B A B

$d_1 = t_1(_) - 2 = 4$ B A O B A B

$d_2 = 5$ $d_1 = t_1(_) - 1 = 5$

$d = \max\{4, 5\} = 5$ $d_2 = 2$

$d = \max\{5, 2\} = 5$

B A O B A B

Good-suffix shift table

k	pattern	d_2
1	BAOBAB	2
2	BAOBAB	5
3	BAOBAB	5
4	BAOBAB	5
5	BAOBAB	5

Bad-symbol shift table

c	A	B	C	D	...	O	...	Z	_
$t_1(c)$	1	2	6	6	6	3	6	6	6

Analysis

m = length of pattern, n = length of text, Σ = size of alphabet

Horspool's
- Preprocessing in O(m+Σ) time, O(Σ) space
- Search in O(mn) worst-case, Θ(n) for random texts

Boyer-Moore "with Galil rule"
- Preprocessing in O(m+Σ) time, O(Σ) space
- Search in O(n+m) worst-case

Both algorithms excel if m, Σ are large

Hashing

A very efficient method for implementing a *dictionary,* i.e., a set with the operations:
- find
- insert
- delete

Idea: distribute keys among a one-dimensional array called a *hash table*, as specified by a *hash function*

Representation-change, space-for-time tradeoff

All above operations are (amortized) O(1).

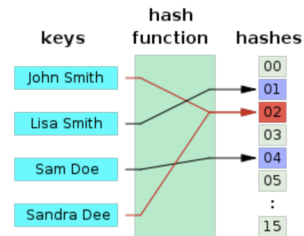
Important applications:
- symbol tables (in code compilation)
- databases (often using *extendible hashing*)
- Bloom filters (efficient support for "set membership" queries)

Hash Functions

Hashing Idea: map key K into hash table of size m, using a *hash function, h*

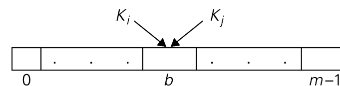
$h: K \rightarrow$ location (cell) in the hash table

Generally, a hash function should:
- be easy to compute
- distribute keys evenly throughout the hash table



Collisions

If $h(K_1) = h(K_2)$, there is a *collision*.



Good hash functions result in few collisions.

Different strategies to handle collisions result in the two principal versions of hashing:
- *Open hashing:* each cell is a header of linked list of all keys hashed to it
- *Closed hashing:* one key per cell
  - In case of collision, find another cell by:
    - *linear probing:* use next free bucket
    - *Quadratic probing:* use free bucket calculated by index^2
    - *double hashing:* use second hash function to compute increment

Open Hashing (Separate Chaining)

- Example: A, FOOL, AND, HIS, MONEY, ARE, SOON, PARTED
- $h(K)$ = sum of K's letters' positions in the alphabet MOD 13

Key	A	FOOL	AND	HIS	MONEY	ARE	SOON	PARTED
***h*(*K*)**	1	9	6	10	7	11	11	12

Search for KID. *h*(KID) = 11

Open Hashing Analysis

Let m be the size of the hash-table.

Let n be the number of elements in the hash-table.

The load factor $\alpha = n/m$.

Load α is typically kept small; around 0.7

Expected length of linked list: <= 1

Open hashing still works if $n > m$

Closed Hashing (Open Addressing)

Key	A	FOOL	AND	HIS	MONEY	ARE	SOON	PARTED
***h*(*K*)**	1	9	6	10	7	11	11	12

0	1	2	3	4	5	6	7	8	9	10	11	12
	A											
	A								FOOL			
	A					AND			FOOL			
	A					AND			FOOL	HIS		
	A					AND	MONEY		FOOL	HIS		
	A					AND	MONEY		FOOL	HIS	ARE	
	A					AND	MONEY		FOOL	HIS	ARE	SOON
PARTED	A					AND	MONEY		FOOL	HIS	ARE	SOON

Closed Hashing Analysis

Does not work if $n > m$

Number of probes to find/insert/delete a key depends on load factor $\alpha = n/m$ (hash table density) and collision resolution strategy.

For linear probing:

$$S = (½)\,(1+ 1/(1- \alpha)) \quad \text{and} \quad U = (½)\,(1+ 1/(1- \alpha)^2)$$

As the table gets filled (α approaches 1), number of probes in linear probing increases dramatically:

α	$\frac{1}{2}(1+\frac{1}{1-\alpha})$	$\frac{1}{2}(1+\frac{1}{(1-\alpha)^2})$
50%	1.5	2.5
75%	2.5	8.5
90%	5.5	50.5