MA/CSSE 473 Day 26

Student questions

Boyer-Moore

B Trees

Recap: Boyer Moore Intro

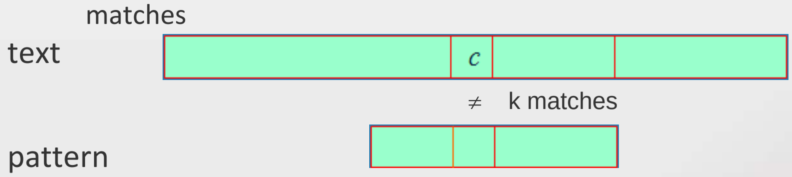
- When determining how far to shift after a mismatch
  - Horspool only uses the text character corresponding to the rightmost pattern character
  - Can we do better?
- Often there is a partial match (on the right end of the pattern) before a mismatch occurs
- Boyer-Moore takes into account k, the number of matched characters before a mismatch occurs.
- If k=0, same shift as Horspool. So we consider $0 < k < m$ (if $k = m$, it is a match).

Boyer-Moore Algorithm

- Based on two main ideas:
- compare pattern characters to text characters from right to left
- precompute the shift amounts in two tables
  - **bad-symbol table** indicates how much to shift based on the text's character that causes a mismatch
  - **good-suffix table** indicates how much to shift based on matched part (suffix) of the pattern

Bad-symbol shift in Boyer-Moore

- If the rightmost character of the pattern does not match, Boyer-Moore algorithm acts much like Horspool's
- If the rightmost character of the pattern does match, BM compares preceding characters right to left until either
  - all pattern's characters match, or
  - a mismatch on text's character c is encountered after $k > 0$ matches



bad-symbol shift: How much should we shift by?

$d_1 = \max\{t_1(c) - k, 1\}$,
where $t_1(c)$ is the value from the Horspool shift table.

Boyer-Moore Algorithm

After successfully matching $0 < k < m$ characters, with a mismatch at character k from the end (the character in the text is c), the algorithm shifts the pattern right by

$$d = \max\{d_1, d_2\}$$

where $d_1 = \max\{t_1(c) - k, 1\}$ is the bad-symbol shift

$d_2(k)$ is the good-suffix shift

Remaining question:
How to compute good-suffix shift table?

$d_2[k]$ = ???

Boyer-Moore Recap 2

n	length of text
m	length of pattern
i	position in text that we are trying to match with rightmost pattern character
k	number of characters (from the right) successfully matched before a mismatch

After successfully matching $0 \le k < m$ characters, the algorithm shifts the pattern right by

$$\boldsymbol{d = \max\{d_1, d_2\}}$$

where $\boldsymbol{d_1 = \max\{t_1[c] - k, 1\}}$ is the *bad-symbol* shift

($t_1[c]$ is from Horspool table)

$\boldsymbol{d_2[k]}$ is the *good-suffix* shift

(next we explore how to compute it)

Good-suffix Shift in Boyer-Moore

- Good-suffix shift d_2 is applied after the k last characters of the pattern are successfully matched
  - 0 < k < m
- How can we take advantage of this?
- As in the bad suffix table, we want to pre-compute some information based on the characters in the suffix.
- We create a **good suffix table** whose indices are k = 1...m-1, and whose values are how far we can shift after matching a k-character suffix (from the right).
- Spend some time talking with one or two other students. Try to come up with criteria for how far we can shift.
- Example patterns: CABABA AWOWWOW
  WOWWOW ABRACADABRA

Solution (hide this until after class)

1. banana

k	shift
1	4
2	6
3	2
4	6
5	6

2. wowwow

k	shift
1	2
2	5
3	3
4	3
5	3

3. abcdcbcabcabc

k	shift
1	8
2	6
3	10
4	10
5	3
6	10
7	10
8	10
9	10
10	10
11	10
12	10

Boyer-Moore example (Levitin)

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

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

$d_1 = t_1(\text{K}) = 6$ B A O B A B

$d_1 = t_1(\_)-2 = 4$

$\underline{d_2(2) = 5}$

B A O B A B

$\underline{d_1 = t_1(\_)-1 = 5}$

$d_2(1) = 2$

B A O B A B (success)

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

Boyer-Moore Example (mine)

```
pattern = abracadabra
text =
abracadabtabradabracadabcadaxbrabbracadabraxxxxxxabracadabracadabra
m = 11,  n = 67
badCharacterTable: a3 b2 r1 a3 c6 x11
GoodSuffixTable: (1,3) (2,10) (3,10) (4,7) (5,7) (6,7) (7,7) (8,7)
(9,7) (10, 7)

abracadabtabradabracadabcadaxbrabbracadabraxxxxxxabracadabracadabra
abracadabra
i =  10     k =  1     t1 =  11     d1 =  10     d2 =  3
abracadabtabradabracadabcadaxbrabbracadabraxxxxxxabracadabracadabra
          abracadabra
i =  20     k =  1     t1 =  6     d1 =  5     d2 =  3
abracadabtabradabracadabcadaxbrabbracadabraxxxxxxabracadabracadabra
               abracadabra
i =  25     k =  1     t1 =  6     d1 =  5     d2 =  3
abracadabtabradabracadabcadaxbrabbracadabraxxxxxxabracadabracadabra
                    abracadabra
i =  30     k =  0     t1 =  1     d1 =  1
```

Boyer-Moore Example (mine)

First step is a repeat from the previous slide

```
abracadabtabradabracadabcadaxbrabbracadabraxxxxxxabracadabracadabra
                    abracadabra
i =  30     k =  0     t1 =  1     d1 =  1
abracadabtabradabracadabcadaxbrabbracadabraxxxxxxabracadabracadabra
                     abracadabra
i =  31     k =  3     t1 =  11     d1 =  8     d2 =  10
abracadabtabradabracadabcadaxbrabbracadabraxxxxxxabracadabracadabra
                               abracadabra
i =  41     k =  0     t1 =  1     d1 =  1
abracadabtabradabracadabcadaxbrabbracadabraxxxxxxabracadabracadabra
                                abracadabra
i =  42     k =  10     t1 =  2     d1 =  1     d2 =  7
abracadabtabradabracadabcadaxbrabbracadabraxxxxxxabracadabracadabra
                                       abracadabra
i =  49     k =  1     t1 =  11     d1 =  10     d2 =  3
abracadabtabradabracadabcadaxbrabbracadabraxxxxxxabracadabracadabra
                                                 abracadabra
49
```

Brute force took 50 times through the outer loop; Horspool took 13; Boyer-Moore 9 times.

Boyer-Moore Example

- On Moore's home page
- http://www.cs.utexas.edu/users/moore/best-ideas/string-searching/fstrpos-example.html

B-trees

- We will do a quick overview.
- For the whole scoop on B-trees (Actually B+ trees), take CSSE 333, Databases.
- Nodes can contain multiple keys and pointers to other to subtrees

B-tree nodes

- Each node can represent a block of disk storage; pointers are disk addresses
- This way, when we look up a node (requiring a disk access), we can get a lot more information than if we used a binary tree
- In an n-node of a B-tree, there are n pointers to subtrees, and thus n-1 keys
- For all keys in T_i , $K_i \leq T_i < K_{i+1}$
  K_i is the smallest key that appears in T_i

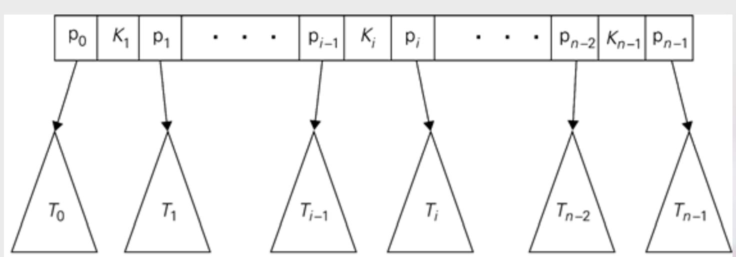


FIGURE 7.7 Parental node of a B-tree

B-tree nodes (tree of order m)

- All nodes have at most m-1 keys
- All keys and associated data are stored in special ***leaf*** nodes (that thus need no child pointers)
- The other (parent) nodes are ***index*** nodes
- All index nodes except the root have between $\lceil m/2 \rceil$ and m children
- root has between 2 and m children
- All leaves are at the same level
- The space-time tradeoff is because of duplicating some keys at multiple levels of the tree
- Especially useful for **data that is too big to fit in memory.** Why?
- Example on next slide

Example B-tree(order 4)

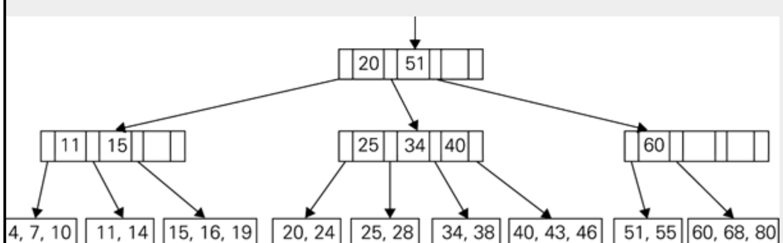


FIGURE 7.8 Example of a B-tree of order 4

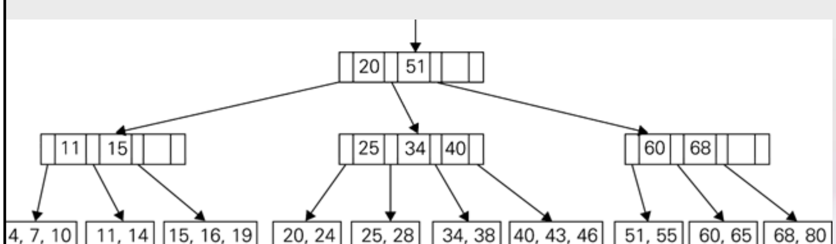


FIGURE 7.9 B-tree obtained after inserting 65 into the B-tree in Figure 7.8

Search for an item

- Within each parent or leaf node, the keys are sorted, so we can use binary search (log m), which is a constant with respect to n, the number of items in the table
- Thus the search time is proportional to the height of the tree
- Max height is approximately $\log_{\lceil m/2 \rceil} n$
- **Exercise for you:** Read and understand the straightforward analysis on pages 273-274
- Insert and delete are also proportional to height of the tree