CS/COE 1501

cs.pitt.edu/~bill/1501/

String Pattern Matching
General idea

• Have a pattern string \( p \) of length \( m \)

• Have a text string \( a \) of length \( n \)

• Can we find an index \( i \) of string \( a \) such that each of the \( m \) characters in the substring of \( a \) starting at \( i \) matches each character in \( p \)

  • Example: can we find the pattern “fox” in the text “the quick brown fox jumps over the lazy dog”?

    • Yes! At index 16 of the text string!
Simple approach

• **BRUTE FORCE**
  • start at the beginning of both pattern and text
  • compare characters left to right
  • mismatch?
  • start again at the 2nd character of the text and the beginning of the pattern...
public static int bf_search(String pat, String txt) {
    int m = pat.length();
    int n = txt.length();
    for (int i = 0; i <= n - m; i++) {
        int j;
        for (j = 0; j < m; j++) {
            if (txt.charAt(i + j) != pat.charAt(j))
                break;
        }
        if (j == m)
            return i; // found at offset i
    }
    return n; // not found
}
Brute force analysis

- Runtime?
  - What does the worst case look like?
    - $a = \ldots$
    - $p = \ldots$
    - $m (n - m + 1)$
      - $\Theta(nm)$ if $n \gg m$
  - Is the average case runtime any better?
    - Assume we mostly miss on the first pattern character
      - $\Theta(n + m)$
        - $\Theta(n)$ if $n \gg m$
Where do we improve?

• Improve worst case
  • Theoretically very interesting
  • Practically doesn’t come up that often for human language

• Improve average case
  • Much more practically helpful
    • Especially if we anticipate searching through large files
First: improving the worst case

Discovered the same algorithm independently

Knuth

Morris

Pratt

Worked together

Jointly published in 1976
• Knuth Morris Pratt algorithm (KMP)
• Goal: avoid backing up in the text string on a mismatch
• Main idea: In checking the pattern, we learned something about the characters in the text, take advantage of this knowledge to avoid backing up
How do we keep track of text processed?

- Build a deterministic finite-state automata (DFA) storing information about the pattern
  - From a given state in searching through the pattern, if you encounter a mismatch, how many characters currently match from the beginning of the pattern
DFA example

Pattern: ABABAC
Representing the DFA in code

- DFA can be represented as a 2D array:
  - `dfa[cur_text_char][pattern_counter] = new_pattern_counter`
  - Storage needed?
    - `mR`

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
public int kmp_search(String pat, String txt) {
    int M = pat.length();
    int N = txt.length();
    int i, j;
    for (i = 0, j = 0; i < N && j < M; i++)
        j = dfa[txt.charAt(i)][j];
    if (j == M) return i - M; // found
    return N; // not found
}

Runtime?
What if we compare starting at the end of the pattern?
- $a = \text{ABCDVABCDWABCDXABCDYABCDZ}$
- $p = \text{ABCD}$
- V does not match E
  - Further V is nowhere in the pattern...
  - So skip ahead $m$ positions with 1 comparison!
    - Runtime?
      - In the best case, $n/m$

When searching through text with a large alphabet, will often come across characters not in the pattern.
- One of Boyer Moore’s heuristics takes advantage of this fact
  - Mismatched character heuristic
Missed character heuristic

• How well it works depends on the pattern and text at hand
  • What do we do in the general case after a mismatch?
    • Consider:
      • $a = XYXYZYXXXXXXXXXXXXXXXXX$
      • $p = XYXYZ$
    • If mismatched character *does* appear in $p$, need to “slide” to the right to the next occurrence of that character in $p$
      • Requires us to pre-process the pattern
        • Create a right array

        ```java
        for all $i$: right[$i$] = -1;
        for all $j$ from 0 to $m$:
          right[p.charAt($j$)] = $j$;
        ```
Missed character heuristic example

Text:  A B C D X A B C D C A B C D Y A E C D E A B C D E

Pattern:  A B C D E

right = [0, 1, 2, 3, 4, -1, -1, ... ]
What does the worst case look like?

Runtime:

Θ(nm)

Same as brute force!

This is why missed character is only one of Boyer Moore’s heuristics

The Galil rule works similarly to KMP

See BoyerMoore.java
Another approach

- Hashing was cool, let's try using that

```java
public static int hash_search(String pat, String txt) {
    int m = pat.length();
    int n = txt.length();
    int pat_hash = h(pat);
    for (int i = 0; i <= n - m; i++) {
        if (h(txt.substring(i, i + m)) == pat_hash)
            return i; // found!
    }
    return n; // not found
}
```
Well that was simple

- Is it efficient?
  - Nope! Practically worse than brute force
    - Instead of $nm$ character comparisons, we perform $n$ hashes of $m$ character strings
  - Can we make an efficient pattern matching algorithm based on hashing?
Horner’s method

- Brought up during the hashing lecture

```java
public long horner_hash(String key, int m) {
    long h = 0;
    for (int j = 0; j < m; j++)
        h = (R * h + key.charAt(j)) % Q;
    return h;
}
```

- `horner_hash("abcd", 4) =`
  - 'a' * R^3 + 'b' * R^2 + 'c' * R + 'd' mod Q
- `horner_hash("bcde", 4) =`
  - 'b' * R^3 + 'c' * R^2 + 'd' * R + 'e' mod Q
- `horner_hash("cdef", 4) =`
  - 'c' * R^3 + 'd' * R^2 + 'e' * R + 'f' mod Q
Efficient hash-based pattern matching

text = "abcdefg"

pattern = "defg"

• This is Rabin-Karp
What about collisions?

• Note that we’re not storing any values in a hash table...
  • So increasing Q doesn’t affect memory utilization!
    • Make Q really big and the chance of a collision becomes really small!
      • But not 0...

• OK, so do a character by character comparison on a collision just to be sure
  • Worst case runtime?
    • Back to brute-force-esque runtime...
Assorted casinos

- Two options:
  - Do a character by character comparison after collision
    - Guaranteed correct
    - Probably fast
  - Assume a hash match means a substring match
    - Guaranteed fast
    - Probably correct