CMPSC250
Lecture 23: Boyer-Moore and Rabin-Karp
Substring Search

Prof. John Wenskovitch
03/16/2016
Last Time

• Strings
  – They’re important
  – We need to sort them (and search them (and search inside them))

• Brute Force substring search
• KMP (Knuth-Morris-Pratt) substring search
Substring Search

• Problem overview:
  – Given a pattern and a text sequence, find the pattern in the text.
  – Return the location where the pattern starts, or length of the text if not found
KMP Substring Search

internal representation

\[
\begin{array}{cccccc}
 j & 0 & 1 & 2 & 3 & 4 & 5 \\
\text{pat.charAt}(j) & A & B & A & B & A & C \\
\text{dfa}[i][j] & A & 1 & 1 & 3 & 1 & 5 & 1 \\
& B & 0 & 2 & 0 & 4 & 0 & 4 \\
& C & 0 & 0 & 0 & 0 & 0 & 6 \\
\end{array}
\]

- mismatch transition (back up)
- match transition (increment)
- halt state
Boyer-Moore Substring Search

• Idea: If we assume that backing up in the text string is not an issue (we’re not dealing with streaming data), then we can scan the pattern from right-to-left when trying to match it against the text.
Boyer-Moore – Finding Skip Distance

```java
int R = 256;
for (int c = 0; c < R; c++) {
    right[c] = -1;
}

for (int j = 0; j < M; j++) {
    right[pat.charAt(j)] = j;
}
```
Boyer-Moore – Finding Skip Distance

<table>
<thead>
<tr>
<th></th>
<th>N</th>
<th>E</th>
<th>E</th>
<th>D</th>
<th>L</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td></td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>A</td>
<td></td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>B</td>
<td></td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>C</td>
<td></td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>D</td>
<td></td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>-1</td>
<td>3</td>
</tr>
<tr>
<td>E</td>
<td></td>
<td>-1</td>
<td>-1</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td>2</td>
<td>5</td>
</tr>
</tbody>
</table>

right[c]

-1
-1
-1
3
5
4
4
-1
0
-1
Boyer-Moore – Search

```java
int search(String txt) {
    int N = txt.length();
    int M = pat.length();
    int skip;
    for (int i = 0; i < N-M; i += skip) {
        skip = 0;
        for (int j = M-1; j >= 0; j--) {
            if (pat.charAt(j) != txt.charAt(i+j)) {
                skip = j - right[txt.charAt(i+j)];
                if (skip < 1) {
                    skip = 1;
                } //if
                break;
            } //if
        } //for
        if (skip == 0) {
            return i; // found
        } //if
    } //for
    return N; // not found
} //search
```
Boyer-Moore – Search

- Index $i$ moves left to right across the text
- Index $j$ moves right to left across the pattern
- Inner loop tests whether the pattern aligns with the text at position $i$.
- If the pattern matches from $M - 1$ to 0, search hit.
- If not, three cases:
  - If char causing mismatch is not found in the pattern, move the pattern $j+1$ to the right.
  - If char $c$ causing mismatch is found in the pattern, use the $\text{right}[\cdot]$ array to line up the pattern with the text, so that we increment $i$ by $j - \text{right}[c]$.
  - If this computation will not increase $i$, just increment $i$ instead to make sure the pattern always slides at least one position to the right (that’s why we put -1s in $\text{right}[\cdot]$).
Boyer-Moore – Visual

increment i by j - right['N']

to line up text with N in pattern

lining up text with rightmost E
would shift pattern left

so increment i by 1

reset j to M-1

could do better with KMP-like table
Boyer-Moore – Analysis

• **Theorem:** On typical inputs, substring search with Boyer-Moore uses $\sim N/M$ character compares to search for a pattern of length $M$ in a text of length $N$.

• **Proof:** (Hand-wavy) In many practical situations, all but a few of the alphabet characters appear nowhere in the pattern, so nearly all compares lead to $M$ characters being skipped on every iteration.
Rabin-Karp Fingerprint Search

• Idea: Compute a hash function for the pattern, and then look for a match by using the same hash function for each possible $M$-character substring of the text.
A string of length $M$ corresponds to an $M$-digit base-$R$ number.

- To use a hash table of size $Q$ for keys of this type, we need a hash function that converts this number to an int value between 0 and $Q - 1$.
- Modular hashing will work, with a random prime $Q$, picking a large $Q$ while avoiding overflow.

```java
long hash(String key, int M) {
    long h = 0;
    for (int j = 0; j < M; j++) {
        h = (R * h + key.charAt(j)) % Q;
    } //for
    return h;
} //hash
```
Rabin-Karp – Hashing

\[ x_i = t_i R^{M-1} + t_{i+1} R^{M-2} + t_{i+2} R^{M-3} + \cdots + t_{i+(M-1)} R^0 \]

\[ x_{i+1} = t_{i+1} R^{M-1} + t_{i+2} R^{M-2} + \cdots + t_{i+(M-1)} R^1 + t_{i+M} R^0 \]

\[ x_{i+1} = (x_i - t_i R^{M-1}) \times R + t_{i+M} \]

---

**pat.charAt(j)**

<table>
<thead>
<tr>
<th>i</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>6</td>
<td>5</td>
<td>3</td>
<td>5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0</th>
<th>2</th>
<th>% 997 = 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>6</td>
</tr>
</tbody>
</table>
int search (String txt) {
    int N = txt.length;
    long txtHash = hash(txt, M);
    if (patHash == txtHash && check(0)) {
        return 0; // match
    } //if
    for (int i = M; i < N; i++) {
        txtHash = (txtHash + Q - RM*txt.charAt(i-M) % Q) % Q;
        txtHash = (txtHash*R + txt.charAt(i)) % Q;
        if (patHash == txtHash) {
            if (check(i - M + 1)) {
                return i - M + 1; // match
            } //if
        } //if
    } //for
    return N; // no match
} //search
How Do We Known the Hash Value is Giving us a Correct Match?

• We don’t.
• We’re not actually building a hash table, just looking for a single match to the pattern.
• Pick a really big Q, like $10^{20}$ or $10^{40}$.
• The probability of a false positive match is then $\frac{1}{10^{20}}$ or $\frac{1}{10^{40}}$. 
Monte Carlo Algorithms

• This is called a **Monte Carlo algorithm**.
  – Has guaranteed completion time, but fails to output a correct answer with small probability.

• The alternative is a **Las Vegas algorithm**.
  – May never finish, but if it does, the answer will be correct.
  – *(have we already seen a LV algorithm?)*
Theorem: The Monte Carlo version of Rabin-Karp is linear time on the length of text input $N$, and is extremely likely to be correct.

Proof: In the worst case, we scan across the full input string $N$ to find a match. In the average case, we scan across $N/2$ characters in the input string.

So this is worse than KMP and BM, so why use it?

- Only other algorithm (besides brute force) that uses no extra space (KMP requires $MR$, BM requires $R$).
## Substring Search Summary

<table>
<thead>
<tr>
<th>algorithm</th>
<th>version</th>
<th>operation count</th>
<th>backup in input?</th>
<th>correct?</th>
<th>extra space</th>
</tr>
</thead>
<tbody>
<tr>
<td>brute force</td>
<td>—</td>
<td>$MN$</td>
<td>yes</td>
<td>yes</td>
<td>1</td>
</tr>
<tr>
<td>Knuth-Morris-Pratt</td>
<td>full DFA (Algorithm 5.6)</td>
<td>$2N$</td>
<td>no</td>
<td>yes</td>
<td>MR</td>
</tr>
<tr>
<td></td>
<td>mismatch transitions only</td>
<td>$3N$</td>
<td>no</td>
<td>yes</td>
<td>M</td>
</tr>
<tr>
<td></td>
<td>full algorithm</td>
<td>$3N$</td>
<td>yes</td>
<td>yes</td>
<td>R</td>
</tr>
<tr>
<td>Boyer-Moore</td>
<td>mismatched char heuristic only (Algorithm 5.7)</td>
<td>$MN$</td>
<td>yes</td>
<td>yes</td>
<td>R</td>
</tr>
<tr>
<td></td>
<td>Monte Carlo (Algorithm 5.8)</td>
<td>$7N$</td>
<td>no</td>
<td>yes†</td>
<td>1</td>
</tr>
<tr>
<td>Rabin-Karp‡</td>
<td>Las Vegas</td>
<td>$7N^†$</td>
<td>yes</td>
<td>yes</td>
<td>1</td>
</tr>
</tbody>
</table>
Any Questions?