Seminumerical String Matching

Most matching methods based on char comparisons

“Seminumerical” methods that apply bit-level and/or arithmetic operations:

* FFT-based $O(n \log m)$ solution to the match count problem:
  * For each target position $j$, find the number of matching characters in corresponding positions of $P[1 \ldots n]$ and $T[j-n+1 \ldots j]$
* Karp-Rabin pattern matching, based on comparing hash values of $P$ and of substrings of $T$

We discuss Shift-And, which is a practical and efficient matching method for short patterns

Shift-And: Basic idea

Consider locating exact occurrences of pattern $P[1 \ldots n]$ in target $T[1 \ldots m]$, by going through positions $j = 1, \ldots, m$

Basic idea: The state of the search is represented as a state vector, which is a bit vector $S[1 \ldots n]$

The vector records, simultaneously, occurrences of any prefix of $P$ that end at the current target position $j$:

$$S[i] = 1 \text{ iff } P[1 \ldots i] = T[j-i+1 \ldots j]$$

(for $i = 1, \ldots, n$)

Computing the State Vectors

Consider values of $S$ for positions $j = 1, \ldots, m$ as columns $M(1), \ldots, M(m)$ of an $n \times m$ matrix $M$:

<table>
<thead>
<tr>
<th>T</th>
<th>m e n n e l u t t u l e n</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>1 2 3 4 5 6 7 8 9 10 11 12</td>
</tr>
<tr>
<td>e 1</td>
<td>1 0 1 0 0 1 0 0 0 0 0 1</td>
</tr>
<tr>
<td>n 2</td>
<td>0 0 1 0 0 1 0 0 0 0 0 1</td>
</tr>
<tr>
<td>n 3</td>
<td>0 0 0 1 0 1 0 0 0 0 0 0</td>
</tr>
<tr>
<td>n 4</td>
<td>0 0 0 0 0 1 0 0 0 0 0 0</td>
</tr>
<tr>
<td>n 5</td>
<td>0 0 0 0 0 1 0 0 0 0 0 0</td>
</tr>
<tr>
<td>n 6</td>
<td>0 0 0 0 0 0 1 0 0 0 0 0</td>
</tr>
<tr>
<td>n 7</td>
<td>0 0 0 0 0 0 0 1 0 0 0 0</td>
</tr>
<tr>
<td>n 8</td>
<td>0 0 0 0 0 0 0 0 1 0 0 0</td>
</tr>
<tr>
<td>n 9</td>
<td>0 0 0 0 0 0 0 0 0 1 0 0</td>
</tr>
<tr>
<td>n 10</td>
<td>0 0 0 0 0 0 0 0 0 0 1 0</td>
</tr>
<tr>
<td>n 11</td>
<td>0 0 0 0 0 0 0 0 0 0 0 1</td>
</tr>
<tr>
<td>n 12</td>
<td>0 0 0 0 0 0 0 0 0 0 0 0</td>
</tr>
</tbody>
</table>

Now $M(j)[i] = 1$ iff $P[1 \ldots i]$ matches a suffix of $T[1 \ldots j]$

$$M(j)[i] = 1 \text{ iff } T[j] = P[i], \text{ and } M(1)[i] = 0 \text{ for } i > 0$$

Computing the Columns of $M$

How about $M(j)[i]$ for $i, j > 1$?

Now $M(j)[i] = 1 \Leftrightarrow P[1 \ldots i] = T[j-i+1 \ldots j]$

$$P[1 \ldots i-1] = T[j-i+1 \ldots j-1] \text{ and } P[i] = T[j]$$

$M(j-1)[i-1] = 1 \text{ and } P[i] = T[j]$

So, we can compute the columns in order $j = 1, \ldots, m$

But this requires time $\Theta(mn)$ (and character comparisons!)

If $|P|$ is bounded by a constant, we can compute each column in constant time $\Theta(m)$ time matching algorithm

Is this a practical observation?

Fast Computation of State Vectors

(Remember: State vector is the current column of $M$)

Assume that $|P| \leq w$, where $w$ is the processor word length (normally 32 or 64). Then we can represent the state vector as a single number, and update it really fast $\rightarrow$ a linear-time matching method fast in practice, too

For updating the state vector in a single step, compute for each $a \in \Sigma$ an $n$-bit occurrence mask $U(a)$, which indicates the occurrences of char $a$ in $P$:

$$U(a)[i] = \begin{cases} 1 & \text{if } P[i] = a \\ 0 & \text{otherwise} \end{cases}$$

Example of Occurrence Masks

Occurrence masks for pattern “ennen”:

<table>
<thead>
<tr>
<th>U a b c d e f . . . m n o . . . z</th>
</tr>
</thead>
<tbody>
<tr>
<td>e 1 0 0 0 0 1 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>n 2 0 0 0 0 0 0 0 1 0 0 0 0 0</td>
</tr>
<tr>
<td>n 3 0 0 0 0 0 0 0 0 1 0 0 0 0</td>
</tr>
<tr>
<td>e 4 0 0 0 0 0 0 0 0 0 1 0 0 0</td>
</tr>
<tr>
<td>n 5 0 0 0 0 0 0 0 0 0 0 1 0 0</td>
</tr>
</tbody>
</table>

These can be computed in time $\Theta(|\Sigma|n)$

($= O(n)$ if $\Sigma$ is fixed)

($= O(1)$ if $|P| = n$ is bounded by a constant)
Define an operation which shifts elements of bit-vector $S$ forward by one, and inserts 1 as the first element:

$$\text{BitShift}(S)[i] = \begin{cases} 1 & \text{for } i = 1 \\ S[i-1] & \text{for } i > 1 \end{cases} \text{ for } i = 1, \ldots, n$$

This can be implemented in C-like languages as follows:

$$S = (S << 1) \mid 1$$

Now ANDing BitShift($S$) with the occurrence mask of the current text character $T[j]$ updates the state vector correctly, from column $M(j-1)$ to $M(j)$:

$$M(j) = \text{BitShift}(S) \& U[j]$$

### Correctness of State Vector Updates

**Theorem** If $S = M(j-1)$, then

$$\text{BitShift}(S) \& U(T[j]) = M(j).$$

**Proof.** Assume that $S[i] = M(j-1)[i]$ for $i = 1, \ldots, n$, and let $S' = \text{BitShift}(S) \& U(T[j])$.

- $S'[i] = 1$ if $U(T[j])[i] = 1$ or $T[j] = 1$, and $M(j)[i] = 1$

For $i > 1$,

- $S'[i] = 1$ if $S[i-1] = 1$ and $U(T[j])[i] = 1$
- $M(j-1)[i-1] = 1$ and $T[j] = 1$
- $M(j)[i] = 1$

So, $S'[i] = M(j)[i]$ for each $i = 1, \ldots, n$. \hfill $\blacksquare$

### Shift-And Implementation

**Sketch of a full algorithm using C-like bit operations:**

```plaintext
computeOccurrenceMasks(P);
S := 0;
one_at_row_n := 1 << (n-1);
for j := 1 to n do
  S := BitShift(S) & U(T[j]);
  if (S & one_at_row_n) then
    Report an occurrence at j - n + 1;
endfor;
```

**Time:** $\Theta(m)$ (assuming fixed alphabet and $|P| \leq w$)

### Extension to Inexact Matching

Wu and Manber (CACM 10/1992) extended Shift-And to inexact matching or matching with errors:

*Find all target positions where $P$ occurs with at most $k$ errors (single-char mismatches, insertions or deletions)\(^1\)*

\(^1\)The algorithm runs in $O(nk + m)$ time.

Now we restrict to mismatches (and discuss inexact matching in general later)

**Ex:** ATCGAA occurs with 2 mismatches at posn 4 of

```
1 2 3 4 5 6 7 8 9 10 11
A A T A T C A C A A
```

and with 4 mismatches at positions 2 and 6

### Shift-And with Mismatches: Main ideas

**Wu-Manber method** is efficient for small $|P| \leq w$ and $k \leq 4$. It is included in their approximate search tool agrep

Instead of a single table $M$ (as before), compute tables $M^0, M^1, \ldots, M^k$, each of size $n \times m$:

$$M^i[j] = 1$$ if $P[1 \ldots i]$ matches $T[j-i+1 \ldots j]$ with at most $i$ mismatches

**Obs 1:** $M^0 = M$

**Obs 2:** $M^k[j][n] = 1$ if $T[j-n+1 \ldots j]$ is an occurrence of $P$ with at most $k$ mismatches

**Obs 3:** For $i \geq 1$, $M^i[j][1] = 1$, and $M^1[i][i] = 0$ when $i > 1$

### How to Compute $M^k$

Compute column $j-1$ of each table before column $j$, and for each $j$, tables in order $M^0, \ldots, M^k$:

- Column $M^0(j)$ from $M^0(j-1)$ as in Shift-And
- When computing column of $M^i(j)$ for $i > 0$ and $j > 1$,
  - Column $M^{i-1}(j-1)$,
  - Column $M^i(j-1)$, and
  - Column $M^{i-1}(j)$

have been computed

*(Example of dynamic programming)*
Computing Table Entries

When is $M_l(j)[i] = 1$ (for $l > 0$)? ⇔ When does $P[1 \ldots i]$ match $T[j - i + 1 \ldots]$ with at most $l$ mismatches? (*)

1. If $P[1 \ldots i]$ matches $T[j - i + 1 \ldots j]$ with at most $l - 1$ mismatches ($\Leftrightarrow M_{l-1}(j)[i] = 1$)
2. If $P[1 \ldots i - 1]$ matches $T[j - i + 1 \ldots j - 1]$ with at most $l$ mismatches and $P[i] = T[j]$ ($\Leftrightarrow M_l(j - 1)[i - 1] = 1$ and $P[i] = T[j]$)
3. If $P[1 \ldots i - 1]$ matches $T[j - i + 1 \ldots j - 1]$ with at most $l - 1$ mismatches ($\Leftrightarrow M_{l-1}(j - 1)[i - 1] = 1$)

Conversely, (*) holds only if at least one of (1)–(3) holds.

Computing Columns of $M_l$

An entire column $M_l(j)$ can now be computed, using bit operations, as follows:

$$M_l(j) := M_{l-1}(j) \quad (1)$$
OR $$\text{BitShift}(M_l(j - 1)) \text{ AND } U(T[j]) \quad (2)$$
OR $$\text{BitShift}(M_{l-1}(j - 1)) \quad (3)$$

(1)–(3) correspond to the cases of the previous slide.

Complexity

Only two columns ($j$ and $j - 1$) of each table need to be maintained → Space complexity is $\Theta(kn) = \Theta(k \times |P|)$

$k + 1$ tables each of size $n \times m$ are computed, each table entry in constant time → total time is $\Theta(knm)$

If $n \leq w$, each column is computed with a few machine instructions, and time is $\Theta(km)$ (and fast in practice).