Biosequence Algorithms, Spring 2007
Lecture 5: The Shift-And Method

Pekka Kilpeläinen

University of Kuopio
Department of Computer Science
Seminumerical String Matching

Most matching methods based on **char comparisons**

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

- FFT-based $O(m \log m)$ 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.
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 $i = 1, \ldots, n$)
**Shift-And State Vector**

**Example:** Consider the state of the search at position \( j = 6 \) of text “mennentullen” for pattern \( P = \text{“ennen”} \)

Prefixes \( P[\ldots 2] = \text{“en”} \) and \( P[\ldots 5] = \text{“ennen”} \) both match a suffix of \( T[\ldots 6] \)

\[
S = \begin{bmatrix}
0 \\
1 \\
0 \\
0 \\
1
\end{bmatrix}
\]

**NB:** We’ve found an occurrence of \( P \) iff \( S[n] = 1 \)

How to compute \( S \) for each \( j = 1, \ldots, m \)?
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></th>
<th>$T$:</th>
<th>m e n n e n t u l l e n</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P$</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>e 1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>n 2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>n 3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>e 4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>n 5</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

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

$\rightarrow M(j)[1] = 1$ iff $T[j] = P[1]$, and $M(1)[i] = 0$ for $i > 0$
Computing the Columns of $M$

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

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

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

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

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

Is this a practical observation?
Fast Computation of State Vectors

(Remember: State vector = 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

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

$$\text{BitShift}(S)[i] = \begin{cases} 
1 & \text{for } i = 1 \\
S[i - 1] & \text{for } i = 2, \ldots, n 
\end{cases}$$

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

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

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 $M(j)$:
**Updating the State Vector: Example**

Consider the step from $j = 3$ to $j = 4$, while searching for $P =$“ennen” in $T =$“mennentullen” ($T[4] = n$):

\[
\begin{array}{cccccc}
S = M(3) & \rightarrow & \text{BitShift}(S) & \& & U(n) & = & M(4) \\
0 & 1 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 \\
\end{array}
\]


The method is correct in general, too:
Correctness of State Vector Updates

Theorem If \( S = M(j - 1) \), then
\[
\text{BitShift}(S) \ \text{AND} \ U(T[j]) = M(j).
\]

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

\[ S'[1] = 1 \iff U(T[j])[1] = 1 \iff T[j] = P[1] \iff M(j)[1] = 1 \]

For \( i > 1 \),

\[ S'[i] = 1 \iff S[i - 1] = 1 \ \text{and} \ U(T[j])[i] = 1 \]
\[ \iff M(j - 1)[i - 1] = 1 \ \text{and} \ T[j] = P[i] \]
\[ \iff M(j)[i] = 1 \]

So, \( S'[i] = M(j)[i] \) for each \( i = 1, \ldots, n \). \( \square \)
Sketch of a full algorithm using C-like bit operations:

```c
computeOccurrenceMasks(P);
S:= 0;
one_at_row_n:= 1 << (n - 1);
for j := 1 to m 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 $|P| \leq w$)
Remarks

This is sometimes called “bit(-level) parallelism”

Original authors (Baeza-Yates & Gonnet, CACM 10/1992) use complemented bits, and call the method **Shift-Or**. Their version is a bit more efficient, but slightly less intuitive.

On English text **Shift-Or** is reported to be about $2.5 \times$ faster than the naive method, and more efficient than Boyer-Moore with patterns of less than 4–10 characters (depending on the BM implementation)

**Shift-And** easily generalizes to matching with wild-cards either in $P$ or $T$ (or both) (Exercise)
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)

("mismatch" = “epävastaavuus” tai “yhteensopimattomuus”)

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

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

\[
\begin{array}{cccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\
\end{array}
\]

and with 4 mismatches at positions 2 and 6
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 $n \times m$:

$M^l(j)[i] = 1$ iff $P[1 \ldots i]$ matches $T[j - i + 1 \ldots j]$ with at most $l$ mismatches.

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

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

**Obs 3:** For $l \geq 1$, $M^l(j)[1] = 1$, and $M^l(1)[i] = 0$ 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 $M^0, M^1, \ldots, M^k$

- column $M^0(j)$ from $M^0(j - 1)$ as in Shift-And

When computing column of $M^l(j)$ for $l > 0$ and $j > 1$,

- column $M^{l-1}(j - 1)$,
- column $M^l(j - 1)$, and
- column $M^{l-1}(j)$

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 $T[j - i + 1 \ldots j]$ with at most $l$ mismatches? (*)

(1) If $P[1 \ldots i]$ matches $T[j - i + 1 \ldots j]$ with at most $l - 1$ mismatches (⇔ $M^{l-1}(j)[i] = 1$)

(2) If $P[1 \ldots i - 1]$ matches $T[j - i + 1 \ldots j - 1]$ with at most $l$ mismatches and $P[i] = T[j]$ (⇔ $M^l(j - 1)[i - 1] = 1$ 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 (⇔ $M^{l-1}(j - 1)[i - 1] = 1$)

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 $\leadsto$ **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 $\leadsto$ 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)