CS481: Bioinformatics Algorithms

Can Alkan
EA224
calkan@cs.bilkent.edu.tr

http://www.cs.bilkent.edu.tr/~calkan/teaching/cs481/
The *Shift-And* Method

- Define $M$ to be a binary $n \times m$ matrix such that:

  
  $$M(i,j) = 1 \text{ iff the first } i \text{ characters of } P \text{ exactly match the } i \text{ characters of } T \text{ ending at character } j.$$  

  
  $$M(i,j) = 1 \text{ iff } P[1..i] \equiv T[j-i+1..j]$$
The *Shift-And* Method

- Let $T = \text{california}$
- Let $P = \text{for}$

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>m = 10</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>M</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

- $M(i,j) = 1$ iff the first $i$ characters of $P$ exactly match the $i$ characters of $T$ ending at character $j$. 
How to construct M

- We will construct M column by column.
- Two definitions:
  - Bit-Shift\((j-1)\) is the vector derived by shifting the vector for column \(j-1\) down by one and setting the first bit to 1.
- Example:

\[
\text{BitShift}\left(\begin{array}{c}
0 \\
1 \\
1 \\
1 \\
\end{array}\right) = \begin{array}{c}
1 \\
0 \\
1 \\
0 \\
\end{array}
\]
How to construct $M$

- We define the $n$-length binary vector $U(x)$ for each character $x$ in the alphabet. $U(x)$ is set to 1 for the positions in $P$ where character $x$ appears.

- Example:

$P = abaac$

$U(a) = \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \end{pmatrix}$

$U(b) = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}$

$U(c) = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}$
How to construct $M$

- Initialize column 0 of $M$ to all zeros
- For $j > 1$ column $j$ is obtained by

$$M(j) = \text{BitShift}(j - 1) \land T(j)$$
An example $j = 1$

$T = x \ a \ b \ x \ a \ b \ a \ a \ c \ a$

$P = a \ b \ a \ a \ c$

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$U(x) = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$

$\text{BitShift}(0) \& U(T(1)) = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} \& \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$
An example $j = 2$

$T = x a b x a b a a c a$

$P = a b a a c$

$$
\begin{pmatrix}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\
1 & 0 & 1 \\
2 & 0 & 0 \\
3 & 0 & 0 \\
4 & 0 & 0 \\
5 & 0 & 0
\end{pmatrix}
$$

$$
U(a) = \begin{pmatrix} 1 \\ 0 \end{pmatrix}
\quad BitShift(1) & U(T(2)) = \begin{pmatrix} 1 \\ 0 \end{pmatrix} & \begin{pmatrix} 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}
$$
An example $j = 3$

$T = x\ a\ b\ x\ a\ b\ a\ a\ c\ a$

$1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ 10$

$P = a\ b\ a\ a\ c$

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

\[
U(b) =
\begin{bmatrix}
0 \\
1 \\
0 \\
0 \\
0 \\
\end{bmatrix}
\]

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

\[
BitShift(2) \& U(T(3)) = \begin{bmatrix}
0 \\
0 \\
0 \\
0 \\
0 \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
1 \\
1 \\
1 \\
1 \\
1 \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
0 \\
0 \\
0 \\
0 \\
0 \\
\end{bmatrix}
\]

\[
= 0
\]

\[
\begin{bmatrix}
0 \\
0 \\
0 \\
0 \\
0 \\
\end{bmatrix}
\]

\[
= 0
\]
An example $j = 8$

\[ T = \langle x, a, b, x, a, b, a, a, c, a \rangle \]

\[ P = \langle a, b, a, a, c \rangle \]

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

\[ U(a) = \begin{pmatrix} 1 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix} \]

\[ BitShift(7) \& U(T(8)) = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 0 \\ 0 \end{pmatrix} \& \begin{pmatrix} 1 \\ 1 \\ 1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 0 \\ 0 \end{pmatrix} \]
Correctness

- For \( i > 1 \), Entry \( M(i,j) = 1 \) iff
  1) The first \( i-1 \) characters of \( P \) match the \( i-1 \) characters of \( T \) ending at character \( j-1 \).
  2) Character \( P(i) \equiv T(j) \).

- 1) is true when \( M(i-1,j-1) = 1 \).
- 2) is true when the \( i \)'th bit of \( U(T(j)) = 1 \).

- The algorithm computes the and of these two bits.
Correctness

\[ T = x \ a \ b \ x \ a \ b \ a \ a \ c \ a \]

\[ a \ b \ a \ a \ c \]

- \( M(4,8) = 1 \), this is because \( a \ b \ a \ a \) is a prefix of \( P \) of length 4 that ends at position 8 in \( T \).
- Condition 1) – We had \( a \ b \ a \) as a prefix of length 3 that ended at position 7 in \( T \) \( \leftrightarrow \) \( M(3,7) = 1 \).
- Condition 2) – The fourth bit of \( P \) is the eighth bit of \( T \) \( \leftrightarrow \) The fourth bit of \( U(T(8)) \) = 1.
How much did we pay?

- Formally the running time is \( \Theta(mn) \).
- However, the method is very efficient if \( n \) is the size of a single or a few computer words.
- Furthermore only two columns of \( M \) are needed at any given time. Hence, the space used by the algorithm is \( O(n) \).
Search in keyword trees

- Naïve threading in keyword trees do not *remember* the partial matches.
- \( P = \{ \text{apple, appropos} \} \)
- \( T = \text{appappappropos} \)
- When threading:
  - *app* is a partial match
  - But naïve threading will go back to the root and re-thread *app*

- Define *failure links*
Failure Link

v: a node in keyword tree K
L(v): the label on v, that is, the concatenation of characters on the path from the root to v.
lp(v): the length of the longest proper suffix of string L(v) that is a prefix of some pattern in P. Let this substring be $\alpha$.

Lemma. There is a unique node in the keyword tree that is labeled by string $\alpha$. Let this node be $n_v$. Note that $n_v$ can be the root.

The ordered pair $(v, n_v)$ is called a failure link.
Failure Link

\[ P = \{ \text{potato, tattoo, theater, other} \} \]
Failure Link

Failure link computation is $O(n)$
Failure Link

\[ \text{xxp} \quad \text{potato} \quad \text{ottoxxx} \]
Failure Link

\[ l = c - \text{lp}(w) = 8 - 3 = 5 \]

\[ c = 8 \]
How to construct failure links for a keyword tree in a linear time?

Let \( d \) be the distance of a node \((v)\) from the root \(r\).

When \( d \leq 1\), i.e., \(v\) is the root or \(v\) is one character away from \(r\),
then \(n_v = r\).

Suppose \(n_v\) has been computed for every node \((v)\) with \(d \leq k\),
we are going to compute \(n_v\) for every node with \(d = k + 1\).

\(v'\): the parent of \(v\), then \(v'\) is \(k\) characters from \(r\), that is \(d = k\)
thus the failure link for \(v'\) has been computed. \(n_{v'}\)

\(x\): the character on edge \((v', v)\)
(1) If there is an edge \((n_v', w)\) out of \(n_v\) labeled with \(x\), then \(n_v = w\).
Failure Link
(2) If such an edge does not exist, examine $n_{v'}$ to see if there is an edge out of it labeled with $x$. Continue until the root.
(2) If such an edge does not exist, examine $n_{_{nV}}$ to see if there is an edge out of it labeled with $x$. Continue until the root.
Failure Link
Failure Link
Output: calculate $n_v$ for $v$

Algorithm $n_v$

$v'$ is the parent of $v$ in $K$

$x$ is the character on edge $(v', v)$

$w = n_v$.

while there is no edge out of $w$ labeled with $x$ and $w \neq r$

\[ w = n_w \]

If there is an edge $(w, w')$ out of $w$ labeled $x$ then

\[ n_v = w' \]

else $n_v = r$
Aho-Corasick Algorithm

Input: Pattern set $P$ and text $T$
Output: all occurrences in $T$ any pattern from $P$

Algorithm AC

$l=1$;
c=1;
w=root of $K$
Repeat
    while there is an edge $(w, w')$ labeled with $T(c)$
        if $w'$ is numbered by pattern $i$ then
            report that $p_i$ occurs in $T$ starting at $l$;
        w=w'; c++;
    w=n_w and $l=c-\text{lp}(w)$;
Until c>m
Suffix arrays

- Suffix arrays were introduced by Manber and Myers in 1993
- More space efficient than suffix trees
- A suffix array for a string $x$ of length $m$ is an array of size $m$ that specifies the lexicographic ordering of the suffixes of $x$. 
## Suffix arrays

Example of a suffix array for acaaacatat$

<table>
<thead>
<tr>
<th></th>
<th>Suffix</th>
<th>Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>aaacatat$</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>aacatat$</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>acaaacatat$</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>acatat$</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>atat$</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>at$</td>
<td>9</td>
</tr>
<tr>
<td>6</td>
<td>caaacatat$</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>catat$</td>
<td>6</td>
</tr>
<tr>
<td>8</td>
<td>tat$</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>t$</td>
<td>10</td>
</tr>
<tr>
<td>10</td>
<td>$</td>
<td>11</td>
</tr>
</tbody>
</table>
**Suffix array construction**

- Naive in place construction
  - Similar to insertion sort
  - Insert all the suffixes into the array one by one making sure that the new inserted suffix is in its correct place
  - Running time complexity:
    - $O(m^2)$ where $m$ is the length of the string
- Manber and Myers give a $O(m \log m)$ construction.
Suffix arrays

- $O(n)$ space where $n$ is the size of the database string
- Space efficient. However, there’s an increase in query time
- Lookup query
  - Based on binary search
  - $O(m \log n)$ time; $m$ is the size of the query
  - Can reduce time to $O(m + \log n)$ using a more efficient implementation
Searching for a pattern in Suffix Arrays

\[
\text{find(Pattern P in SuffixArray A):}
\]

\[
i = 0
\]

\[
lo = 0, \ hi = \text{length}(A)
\]

\[
\text{for } 0 \leq i < \text{length}(P):
\]

\[
\text{Binary search for } x,y
\]

\[
\text{where } P[i] = S[A[j] + i] \text{ for } lo \leq x \leq j < y \leq hi
\]

\[
lo = x, \ hi = y
\]

\[
\text{return } \{A[lo], A[lo+1], \ldots, A[hi-1]\}\]
Search example

Search *is* in *mississippi*$

Examine the pattern letter by letter, reducing the range of occurrence each time.

First letter *i*:
- occurs in indices from 0 to 3

So, pattern should be between these indices.

Second letter *s*:
- occurs in indices from 2 to 3

Done.

Output: *issippi*$ and *ississippi*$

<table>
<thead>
<tr>
<th>0</th>
<th>11</th>
<th>i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8</td>
<td>ippi$</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>issippi$</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>ississippi$</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>mississippi$</td>
</tr>
<tr>
<td>5</td>
<td>10</td>
<td>pi$</td>
</tr>
<tr>
<td>6</td>
<td>9</td>
<td>ppi$</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>sippi$</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>sissippi$</td>
</tr>
<tr>
<td>9</td>
<td>6</td>
<td>sissippi$</td>
</tr>
<tr>
<td>10</td>
<td>3</td>
<td>ssissippi$</td>
</tr>
<tr>
<td>11</td>
<td>12</td>
<td>$</td>
</tr>
</tbody>
</table>
Suffix Arrays

- It can be built very fast.
- It can answer queries very fast:
  - How many times ATG appears?
- Disadvantages:
  - Can’t do approximate matching
  - Hard to insert new stuff (need to rebuild the array) dynamically.