Pattern matching part three
Hamming distance

Benjamin Sach

Advanced Algorithms – COMS31900
2014/2015

Pattern matching with mismatches

Input: A text string $T$ (length $n$) and a pattern string $P$ (length $m$)

$T$ 0 0 0 0 0 0 n n n n n n
$P$ 0 1 2 3 4 5 6 7 8 9 10 11

This is alignment $i$

Ham($i$) = 3

Goal: For every alignment $i$, output Ham($i$), the Hamming distance between $P$ and $T[i..i+m-1]$

The Hamming distance is the number of mismatches

i.e. the number of distinct $j$ such that $P[j] \neq T[i+j]$

A naive algorithm for this problem takes $O(nm)$ time

... but we can do better

Exact pattern matching

Input: A text string $T$ (length $n$) and a pattern string $P$ (length $m$)

$T$ 0 1 2 3 4 5 6 7 8 9 10 11
$P$ 0 2 4 6 8 10 12

Goal: Find all the locations where $P$ matches in $T$

$P$ matches at location $i$ if $P[j] = T[i+j]$ for all $0 \leq j < m$

- A naive algorithm takes $O(nm)$ time
- Many $O(n)$ time algorithms are known (for example the KMP algorithm)

It’s a small alphabet after all

Imagine that the alphabet contains only a small number of different symbols, which we will consider individually...

$T$ 0 1 2 3 4 5 6 7 8 9 10 11
$P$ 0 1 1 1 1 1 1

Replace all $d$ symbols with 1 and everything else with 0

We denote these new strings $T_d$ and $P_d$ and consider...

$T_d \otimes P_d[i] = \sum_{j=0}^{n-k} P_d[j] \times T_d[i+j]$ 1 n

This is the exactly number of matching $i$ at the $i$-th alignment.

How can we work out $(T_d \otimes P_d)$ quickly?

Last year on COMS21103...

Let $A$ and $B$ be $(n-1)$ degree polynomials which can be expressed as...

$A(x) = \sum_{i=0}^{n-1} a_i x^i$ and $B(x) = \sum_{i=0}^{n-1} b_i x^i$

$A[i] = a_i \leftrightarrow P_d[i]$ (or be seen as arrays of length $n$) $B[i] = b_i \leftrightarrow T_d[i+m-i]$ 0

The polynomial $C = A \times B$ can be expressed as...

$C(x) = \sum_{i=0}^{2n-1} c_i x^i$ where $c_i = \sum_{j=0}^{n-1} P_d[j] T_d[i+j]$ 1 n

By the magic of the FFT we can compute $C$ (i.e. every $c_i$ in $O(n \log n)$ time.

Hint 3 Let $A = P_d$ (padded with zeros) and $B = T_d$ (reversed)... now $C$ contains $(T_d \otimes P_d)$
Count all matches involving infrequent symbols.

Let \( \Sigma \) denote the set of alphabet symbols and \( |\Sigma| \) be its size.

Algorithm Summary

Construct \( T_x \) and \( P_x \) for each symbol \( x \) in \( \Sigma \) \((O(n|\Sigma|) \text{ time})\).

Compute \( (T_x \odot P_x) \) for each symbol \( x \) in \( \Sigma \) \((O(n|\Sigma| \text{ log } m) \text{ time})\).

For every \( i \), compute,

\[
\text{Ham}(i) = n - \sum_{\sigma \in \Sigma} (T_x \odot P_x)[i] \quad (O(n|\Sigma|) \text{ time})
\]

This takes \( O(n|\Sigma| \text{ log } m) \) total time and \( O(n) \) space.

However, \( |\Sigma| \) could be as big as \( m \)...

...in which case, this is worse than the naive method!

Coping with a large alphabet

We will now see an algorithm which runs in \( O(\sqrt{m} \text{ log } m) \) time regardless of the alphabet size.

**Definition:** An alphabet symbol is frequent if it occurs at least \( \sqrt{m} \) times in \( P \).

**Definition:** An alphabet symbol is infrequent if it occurs fewer than \( \sqrt{m} \) times in \( P \).

**Key Idea:** Our algorithm will have two main stages:

- **Stage 1** will count all the matches involving frequent symbols (at each alignment of \( P \) and \( T \)).
- **Stage 2** will count all the matches involving infrequent symbols (at each alignment of \( P \) and \( T \)).

The total number of matches is the sum of the matches from Stage 1 and Stage 2.

The infrequent/frequent symbols trick

**Definition:** A symbol is infrequent if it occurs fewer than \( \sqrt{m} \) times in \( P \).

**Definition:** A symbol is frequent if it occurs at least \( \sqrt{m} \) times in \( P \).

Every symbol is either frequent or infrequent.

Stage 2: Count all matches involving infrequent symbols.

Construct an array \( A \) of length \( (n - m + 1) \) which initially contains all zeros.

Make a single pass through \( T \)...

For each character \( T[k] \), where \( 0 \leq k < n \)

- If \( T[k] \) is infrequent:

  - For all \( j \) such that \( T[k] = P[j] \),
  - Increase \( A[k-j] \) by one (except when \( k-j < 0 \)).

Stage 2: Count all matches involving infrequent symbols.

Construct an array \( A \) of length \( (n - m + 1) \) which initially contains all zeros.

Make a single pass through \( T \)...

For each character \( T[k] \), where \( 0 \leq k < n \)

- If \( T[k] \) is infrequent:

  - For all \( j \) such that \( T[k] = P[j] \),
  - Increase \( A[k-j] \) by one (except when \( k-j < 0 \)).
The infrequent/frequent symbols trick

Definition: A symbol is **infrequent** if it occurs fewer than $\sqrt{m}$ times in $P$.

<table>
<thead>
<tr>
<th>$T$</th>
<th>$X$</th>
<th>$X$</th>
<th>$X$</th>
<th>$X$</th>
<th>$X$</th>
<th>$X$</th>
<th>$X$</th>
<th>$X$</th>
<th>$X$</th>
<th>$X$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P$</td>
<td>$a$</td>
<td>$b$</td>
<td>$c$</td>
<td>$d$</td>
<td>$c$</td>
<td>$d$</td>
<td>$c$</td>
<td>$d$</td>
<td>$c$</td>
<td>$d$</td>
</tr>
<tr>
<td>$A$</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Stage 2: Count all matches involving **infrequent** symbols.

Construct an array $A$ of length $(n - m + 1)$, which initially contains all zeros.

- $O(\sqrt{m})$ time
- Make a single pass through $T$...
- For each character $T[k]$, where $0 \leq k < n$
  - If $T[k]$ is infrequent...
  - For all $j$ such that $T[k] = P[j]$
    - Increase $A[k - j]$ by one (except when $(k - j) < 0$)

Pattern matching with mismatches: putting it all together

Algorithm summary

- Stage 0: Classify each symbol as frequent or infrequent: $O(m \log m)$ time
- Stage 1: Count all matches involving frequent symbols: $O(n \sqrt{m \log m})$ time
- Stage 2: Count all matches involving infrequent symbols: $O(n \sqrt{m})$ time

- at any alignment $i$ the number of mismatches is just $m$ minus the total number of matches

Overall, we obtain a time complexity of $O(n \sqrt{m \log m})$.

Conclusion

Input: A text string $T$ (length $n$) and a pattern string $P$ (length $m$)

```
T  X X X X X X X X X
P  a b c d c d c d c
```

Ham$(8) = 3$

Goal: For every alignment $i$, output

Ham$(i)$, the Hamming distance between $P$ and $T[i..i + m - 1]$

The Hamming distance is the number of mismatches

A naive algorithm for this problem takes $O(nm)$ time

We have seen two alternative algorithms:

- One algorithm takes $O(nm \log m)$ time (where $|\Sigma|$ is the alphabet size)
- The other algorithm takes $O(n \sqrt{m \log m})$ time (regardless of the alphabet size)