Lecture 12
Approximate pattern matching (part one)

Benjamin Sach

Exact pattern matching

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

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

$P$ matches at location $i$ if

\[ P[j] = T[i + j] \]  

for all $0 \leq j < m$

(our strings are zero-indexed)

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

In the remainder, we assume that the symbols are numbers…

(if they aren’t just use their bit-representation)

Consider the following expression,

\[ d(i) = m - 1 \sum_{j=0}^{m-1} (T[i + j] - P[j])^2 \]

When do exact matches occur? When $d(i) = 0$

\[ d(i) = 0 \iff T[i \ldots i + m - 1] = P \]

By multiplying out the brackets we have that,

\[ d(i) = \sum_{j=0}^{m-1} P[j]^2 + \sum_{j=0}^{m-1} T[i + j]^2 - 2 \sum_{j=0}^{m-1} P[j]T[i + j] \]

The first term depends only on the pattern
It is the same for all $i$ so can be computed in $O(m)$ time

How hard is it to compute each of these terms?

(for all $0 \leq i < n$)
Exact pattern matching: computing terms

\[ d(i) = \sum_{j=0}^{m-1} P[j]^2 + \sum_{j=0}^{m-1} T[i+j]^2 - \sum_{j=0}^{m-1} P[j]T[i+j] \]

The second term depends only on the text. It can be computed by a sliding window in \( O(n) \) time.

- compute the old value
- add this
- subtract this

\[ \sum_{j=0}^{m-1} T[i+j]^2 = \sum_{j=0}^{m-1} \left( T[(i-1)+j]^2 + T[i+m-1]^2 - T[i-1]^2 \right) \]

\[ m \]

\[ \begin{array}{cccccccc}
1 & 2 & 3 & 2 & 1 & 2 & 1 & 2
\end{array} \]

Exact pattern matching: computing terms

\[ d(i) = \sum_{j=0}^{m-1} P[j]^2 + \sum_{j=0}^{m-1} T[i+j]^2 - \sum_{j=0}^{m-1} P[j]T[i+j] \]

The third term looks more tricky… Luckily, we have a tool for this: cross-correlation

\[ (T \otimes P)[i] = \sum_{j=0}^{m-1} P[j]T[i+j] \]

\[ \cdots 2 \mid 1 \mid 2 \mid 1 \mid 3 \cdots \]

\[ (1 \times 1) + (2 \times 2) + (1 \times 1) = 6 \]

We can compute \((T \otimes P)\) via the FFT.

We obtain \((T \otimes P)[i]\) for all \(i\) in \(O(n \log m)\) time. (we won’t cover this result in this course)

Pattern matching with wildcards

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

\[ T = 0 \mid 1 \mid 2 \mid 3 \mid 0 \mid 1 \mid 2 \mid 1 \mid 0 \mid 0 \mid 3 \mid 1 \mid 1 \]

\[ P = 0 \mid 1 \mid 2 \mid 0 \mid 3 \]

\(m\)

Symbol matches any single character

**Goal:** Find all the locations where \(P\) matches in \(T\).

- \(P\) matches at location \(i\) if
  - for all \(0 \leq j < m\), we have that \(P[j] = T[i+j]\)
  - or \(P[j] = ?\) or \(T[i+j] = ?\)

  - A naive algorithm takes \(O(nm)\) time

  - We can do better using cross-correlations.

Pattern matching with wildcards: a simple solution

\[ T' = 1 \mid 2 \mid 3 \mid 0 \mid 1 \mid 2 \mid 1 \mid 0 \mid 0 \mid 3 \mid 1 \mid 1 \]

\[ P' = 1 \mid 2 \mid 0 \mid 1 \]

Again we rewrite the symbols as integers… call these \(P'\) and \(T'\)

- the wildcard symbol (?) must be encoded as 0
- all other symbols are encoded as positive integers

Consider the following expression.

\[ d_w(i) = \sum_{j=0}^{m-1} \left( P'[j]T'[i+j] - P'[j] \right) \]

\[ 0 \not\equiv T[i+j] \equiv 7 \]

\[ 0 \not\equiv P[j] \equiv T[i+j] \equiv 7 \]

When do exact matches occur?

\[ d_w(i) = 0 \] if \(P\) matches \(T[i \ldots i+m-1]\)
### Pattern matching with wildcards: computing terms

By multiplying out the brackets we have that,

\[ d_w(i) = \sum_{j=0}^{m-1} P'[j]T'[i + j](T'[i + j] - P'[j])^2 = \sum_{j=0}^{m-1} P'[j]^3 T'[i + j] + \sum_{j=0}^{m-1} P'[j]T'[i + j]^3 - 2\sum_{j=0}^{m-1} P'[j]^2 T'[i + j]^2 \]

How hard is it to compute each of these terms? (for all 0 ≤ i < n)

### Pattern matching with wildcards: putting it all together

\[ d_w(i) = \sum_{j=0}^{m-1} P'[j]^3 T'[i + j] + \sum_{j=0}^{m-1} P'[j]T'[i + j]^3 - 2\sum_{j=0}^{m-1} P'[j]^2 T'[i + j]^2 \]

Algorithm summary

- Compute the pattern and text transformations (O(n) time)
- Compute each term using cross-correlations (O(n log m) time)
- Find all i such that \( d_w(i) = 0 \) using the values found (O(n) time)

**Overall, we obtain a time complexity of O(n log m).**

- This is the best known algorithm for this problem.

### Pattern matching with mismatches

**Input** A text string \( T \) (length n) and a pattern string \( P \) (length m)

**Goal:** For all i, output, Ham(i), the Hamming distance between \( P \) and \( T[i \ldots i + m - 1] \)

- 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 takes \( O(nm) \) time
- Can we do better using cross-correlations...

### It's a small alphabet after all

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

- Replace all symbols with 1 and everything else with 0

**Input** A text string \( T_\alpha \) (length n) and a pattern string \( P_\alpha \) (length m)

**Goal:** For all i, output, Ham(i), the Hamming distance between \( P_\alpha \) and \( T_\alpha[i \ldots i + m - 1] \)

- The Hamming distance is the number of mismatches
- i.e. the number of distinct j such that \( P_\alpha[j] \neq T_\alpha[i + j] \)

- A naive algorithm takes \( O(nm) \) time
- Can we do better using cross-correlations...

- We denote these new strings \( T_\alpha \) and \( P_\alpha \) and consider...

\[ (T_\alpha \otimes P_\alpha)[i] = \sum_{j=0}^{m-1} P_\alpha[j]T_\alpha[i + j] \]

This is the number of matching \( \alpha \)s at the i-th alignment,

- which we can compute (for all i) in \( O(n \log m) \) time
It's a small alphabet after all

We saw how to find all matches with a single symbol in $O(n \log m)$ time.

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

Algorithm Summary

Construct $T_\sigma$ and $P_\sigma$ for every symbol $\sigma$ in $\Sigma$ ($O(|\Sigma| \log m)$ time).
Compute $T_\sigma \otimes P_\sigma$ (for every symbol $\sigma$ in $\Sigma$) ($O(|\Sigma| \log m)$ time).
For every $i$, compute:

$$\text{Ham}(i) = m - \sum_{\sigma \in \Sigma} (T_\sigma \otimes P_\sigma)[i] \ . \quad (O(|\Sigma| \log m) \text{ time})$$

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

However, $|\Sigma|$ could be as big as $m$...

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

Conclusions

- Pattern matching with wildcards can be computed in $O(n \log m)$ time.
- Pattern matching with mismatches can be computed in $O(n|\Sigma| \log m)$ time.

“Maybe if I had paid more attention in Data Structures and Algorithms, I would find this comic funny”

- Anonymous Bristol CS student (posted on Facebook)