Intrusion Detection and Malware Analysis

String matching algorithms

Pavel Laskov
Wilhelm Schickard Institute for Computer Science
Problem (Exact string matching)

Given a string \( P \) called a **pattern** and a longer string \( T \) called **text**, find all occurrences, if any, of pattern \( P \) in text \( T \).

Example

Let \( P = aba \) and \( T = bbabaxababay \), then \( P \) occurs in \( T \) starting at positions 3, 7 and 9 (notice the possible overlap of \( P \) in \( T \)).

Remark (Substrings vs. subsequences)

It is a general convention to denote contiguous patterns as **strings** whereas non-contiguous patterns (in the left-to-right order) are referred to as **sequences**.
Inexact matching and alignment

**Problem (Inexact string matching)**

*Given a string* $P$ *and a text* $T$ *find all strings* $S$ *in* $T$ *that contain at most* $k$ *“errors” with respect to* $P$.

**Example**

Let $P = aba$ and $T = bbabaxababay$, then inexact match of $P$ in $T$ with respect to 1 character substitution occurs at positions 1, 3, 5, 7 and 9.

**Problem (Sequence alignment)**

*Alignment* of two strings $S_1$ and $S_2$ *is obtained by inserting spaces either into or at the ends of* $S_1$ and $S_2$ *so that every character of one string corresponds to exactly one character of the other one.*

**Example**

```
q  a  c  _  d  b  d
q  a  w  x  _  b  _
```
A whirlwind tour of string matching

Today:
- Naive string matching
- Fundamental preprocessing
- Knuth-Morris-Pratt
- Set matching

Other problems/algorithms:
- Regular expression matching
- Rabin-Karp fingerprints
- Suffix trees/arrays
- Inexact matching
- Sequence alignment
Naive string matching

- Align left end of $P$ with the left end of $T$ and compare characters of $P$ and $T$ until a mismatch is found or $P$ is exhausted.
- Shift $P$ by one character and restart from the left of $P$.
- Continue until the right end of $P$ shifts past the right end of $T$.
- Running time: $O(mn)$
Naive algorithm:

T: xabxyabxyabxz
P: abxyabxz

*  
  abxyabxz
  ^^^^^^*
  abxyabxz
  ^^^^^^^^*
  abxyabxz
  ^^^^^^*
  abxyabxz
  ^^^^^^*
  abxyabxz
Speeding up the naive method

Naive algorithm:

T: xabxyabxyabxz
P: abxyabxz

Larger shifts:

T: xabxyabxyabxz
P: abxyabxz

Saving partial matches:

T: xabxyabxyabxz
P: abxyabxz

abxyabxz

abxyabxz

abxyabxz

abxyabxz
Speeding up the naive method

Naive algorithm:

T: xabxyabxyabxz
P: abxyabxz

* 
  abxyabxz
  ^^^^^^^
  abxyabxz
  * 
    abxyabxz
    ^^^^^
    abxyabxz
    * 
      abxyabxz
      ^~~~~~~
      abxyabxz
      * 
        abxyabxz

Larger shifts:

T: xabxyabxyabxz
P: abxyabxz

* 
  abxyabxz
  ^^^^^^^
  abxyabxz
  * 
    abxyabxz
    ^~~~~~~
    abxyabxz
    * 
      abxyabxz
      ^~~~~~~

Saving partial matches:

T: xabxyabxyabxz
P: abxyabxz

* 
  abxyabxz
  ^^^^^^^
  abxyabxz
  * 
    abxyabxz
    ^~~~~~~
    abxyabxz
    * 
      abxyabxz
      ^~~~~~~
General idea: spend some modest amount of time on learning about the internal structure of $P$ or $T$ in order to save some time during search.

Different preprocessing techniques were employed in various original string matching algorithms.

Similarity of preprocessing techniques can be expressed in terms of a fundamental preprocessing which is independent of a search algorithm.
**Definition**

Given a string $S$ and a position $i > 1$, let $Z_i(S)$ be the length of the longest substring of $S$ that starts at $i$ and matches a prefix of $S$. This substring is called a $Z$-box.
**Definition**

Given a string $S$ and a position $i > 1$, let $Z_i(S)$ be the length of the longest substring of $S$ that starts at $i$ and matches a prefix of $S$. This substring is called a $Z$-box.

**Definition**

For any index $i$, $r_i$ is the right-most end of $Z$-boxes beginning at or before $i$; $l_i$ is the left end of the corresponding $Z$-box.
Goal: Compute $Z_i$ for each successive position $i$ starting from $i = 2$.

All $Z$-values are kept, as well as the recent pair $r, l$.

Initialization: explicitly compute $Z_2$ by scanning $S[2 \ldots |S|]$ and comparing it with $S[1 \ldots |S|]$.

Recursion: given $Z_2, \ldots, Z_{k-1}, r_{k-1}, l_{k-1}$, compute $Z_k, r_k, l_k$. 
**Recursive step of the Z-algorithm**

- **Key insight:** character in the position \( k \) also appears in the position \( k' = k - l_{k-1} + 1 \); this also holds for entire substrings \( S[k \ldots r_{k-1}] \) and \( S[k' \ldots Z_{l_{k-1}}] \).

\[
S: \begin{array}{cccc}
\alpha & \beta \\
k' & Z_{l_{k-1}} & l_{k-1} & k & r_{k-1}
\end{array}
\]
Recursive step of the Z-algorithm

- **Key insight:** character in the position $k$ also appears in the position $k' = k - l_{k-1} + 1$; this also holds for entire substrings $S[k \ldots r_{k-1}]$ and $S[k' \ldots Z_{l_{k-1}}]$.

- If $Z_{k'} \leq |\beta|$, then $Z_k = Z_{k'}$, and $r, l$ remain unchanged.
**Recursive step of the Z-algorithm**

- **Key insight:** character in the position $k$ also appears in the position $k' = k - l_{k-1} + 1$; this also holds for entire substrings $S[k \ldots r_{k-1}]$ and $S[k' \ldots Z_{l_{k-1}}]$.

- If $Z_{k'} \leq |\beta|$, then $Z_k = Z_{k'}$, and $r, l$ remain unchanged.

- If $Z_{k'} \geq |\beta|$ then the entire $\beta$ is a prefix of $S$. Keep scanning until a mismatch occurs, set $l$ to $k$ and $r$ to the character before a mismatch.
Theorem

All values $Z_i(S)$ can be computed in $O(|S|)$ time.

Proof.

- For each “≤”-iteration $k$, a constant-time work is needed for each iteration.
- In each “≥”-iteration, the value $r_k$ strictly increases, however not beyond the end of $S$. Hence the overall amount of work is bound by $|S|$. 

\[\square\]
Knuth-Morris-Pratt algorithm

General idea:

- Keep scanning the pattern $P$ and text $T$ left-to-right until mismatch is found.
- Shift $P$ such that its prefix overlaps with its suffix before the mismatch position.
- Continue scanning from the mismatching position.
Let the **shift pointer** $s_p_i$ be the length of the longest prefix of $P$ which is the suffix of $P[1 \ldots i]$ (0 if no such prefix exists).

- In KMP, if a mismatch is found in position $i + 1$ in $P$, $P$ can be shifted by $i - s_p_i$ positions to the right.
- For each $i$ in $P$, $s_p_i$ is the length of a $Z$-box ending at $i$.
- **Computing $s_p_i$:** after initializing all $s_p$’s to 0, loop backwards over $j$ and set $s_p_{j+Z_j-1} = Z_j$. 
Correctness of the shift rule

**Theorem**

For any alignment of $P$ and $T$, if characters 1 through $i$ of $P$ match their counterparts in $T$ but character $i + 1$ mismatches $T(k)$, then $P$ can be shifted by $i - sp_i$ positions to the right without passing any occurrence of $P$ in $T$.

**Proof.**

Let $\beta$ denote the prefix of $P$ after shift of length $sp$. By definition of $sp$, $\beta$ matches its counterpart in $T$. Suppose there exist a missed occurrence of $P$ in $T$ which begins earlier than a shifted $P$, i.e. it begins with a prefix $\alpha \beta$. Then $\alpha \beta$ is a suffix of $P$ before shift which is a prefix of (missed) $P$. However, the longest such suffix must have been of $|\beta|$, which is a contradiction. $\square$
procedure Knuth-Morris-Pratt(T, P) ▷ |P| = n; |T| = m
  Preprocess P to find $F(k) = sp_{k-1} + 1$ for all $k$ from 1 to $n + 1$.
  $c \leftarrow 1$
  $p \leftarrow 1$
  while $c + (n - p) \leq m$ do
    while $P(p) = T(c)$ and $p \leq n$ do
      $p \leftarrow p + 1$
      $c \leftarrow c + 1$
    end while
    if $p = n + 1$ then  ▷ End of p
      Report an occurrence of P at the position $c - n$ in T
    end if
    if $p = 1$ then  ▷ Mismatch at position 1 of p
      $c \leftarrow c + 1$
    end if
    $p \leftarrow F(p)$  ▷ Shift P by $p - sp$
  end while
end procedure
Exact pattern set matching

Problem
Given a set of patterns \( \mathcal{P} = \{P_1, \ldots, P_z\} \), find all occurrences of some pattern from \( \mathcal{P} \) in text \( T \).

Possible solutions:
- Run a standard single pattern matching algorithm (e.g. KMP) \( z \) times: \( O(z(m + n)) \).
- Build a suffix tree for \( T \) and scan each pattern in \( \mathcal{P} \) against it: \( O(m + zn) \).
- Build a keyword tree for \( \mathcal{P} \) and run the Aho-Corasick algorithm; \( O(m + n + k) \), where \( k \) is the number of matches.
A **keyword tree** $\mathcal{K}$ corresponding to a set of patterns $\mathcal{P}$ is a tree satisfying the following conditions:

- Each edge is labeled with exactly one character.
- Any two edges out of the same node have different labels.
- Every pattern in $\mathcal{P}$ maps to some node $v$ in $\mathcal{K}$ such this pattern is spelled out by edge labels on the path from the root to $v$.
- Every leaf in $\mathcal{K}$ corresponds to some pattern in $\mathcal{P}$. 

**Keyword tree (trie)**
Keyword tree example

Keyword tree for \( \mathcal{P} = \{ \text{“error”,”potato”,”pottery”,”other”,”otter”} \} : \)
Naive set matching using keyword tree

- Follow a unique path in $\mathcal{K}$ that matches a substring of $T$ starting from a fixed position $l$ until either a marked node or a mismatch is encountered.
- Move to the next position $l$ and repeat until $T$ is exhausted.
- **Running time:** $O(mn)$. 
Definition
For any node $v$ in a keyword tree, let
- $L(v)$ denote the label sequence on the path from root to $v$,
- $lp(v)$ be the longest suffix of $L(v)$ which is a prefix of some pattern in $\mathcal{P}$, and
- $n_v$ be the unique node in $\mathcal{K}$ labeled with the suffix of $L(v)$ of length $lp(v)$.

Definition
A failure link is a pair of nodes $(v, n_v)$. 
Keyword tree for $\mathcal{P} = \{ "error", "potato", "pottery", "other", "otter" \}$ augmented with failure links:
**Aho-Corasick algorithm**

**procedure** AHO-CORASICK Search\((T, P, K)\)

\(c \leftarrow 1\) \hspace{1cm} \triangleright Text running index

\(l \leftarrow 1\) \hspace{1cm} \triangleright Starting position of current match

\(w \leftarrow \text{root of } K\) \hspace{1cm} \triangleright Running keyword tree node pointer

repeat

\textbf{while} there is an edge \((w, w')\) labeled with \(T(c)\) do

\hspace{1cm} \textbf{if} \(w'\) is marked with \(i\) \textbf{then}

\hspace{1cm} \hspace{1cm} Report an occurrence of \(P_i\) at the position \(l\) in \(T\)

\hspace{1cm} \textbf{end if}

\hspace{1cm} \(w \leftarrow w'\)

\hspace{1cm} \(c \leftarrow c + 1\)

\textbf{end while}

\(w \leftarrow n_w\)

\(l \leftarrow c - \text{lp}(w)\)

\textbf{until} \(c > m\)

end procedure
function FAILURE_LINK(v, K)
    v' ← parent of v
    x ← character on the edge (v', v)
    w ← n_v'   ▷ Begin with the failure link of v’s parent
    while there is no edge of w labeled with x, and w ≠ root do
        w ← n_w   ▷ Follow failure links until an edge labeled x found
    end while
    if there is an edge (w, w') out of w labeled with x then
        n_v ← w'
    else
        n_v ← root
    end if
end function
D. Gusfield.

*Algorithms on strings, trees, and sequences.*