Pattern matching algorithms

- Brute force algorithm
- Boyer-Moore algorithm
- Knuth-Morris-Pratt algorithm
Strings
• A string is a sequence of characters
• An alphabet \( \Sigma \) is the set of possible characters for a family of strings
• Example of alphabets:
  – ASCII or Unicode
  – \{0, 1\}
  – \{A, C, G, T\}

Strings
• Let \( P \) be a string of size \( m \)
  – A substring \( P[i..j] \) is a string containing characters of string \( P \) with ranks between \( i \) and \( j \)
  – A prefix of \( P \) is a substring of the type \( P[0..i] \)
  – A suffix of \( P \) is a substring of the type \( P[i..m-1] \)

Pattern Matching Problem
• Given strings \( T \) (text) and \( P \) (pattern), the pattern matching problem consists of finding a substring of \( T \) equal to \( P \)
• Applications:
  – Text editors
  – Search engines
  – Biological research
The Brute force Algorithm

\[ T = xabxyabxyabxz \]
\[ P = abxyabxz \]

Worst-case: \( \Theta(nm) \)

Example of worst case:

- \( T = \text{aaa ... ah} \)
- \( P = \text{aaah} \)
- may occur in images and DNA sequences
- unlikely in English text

Brute-Force Algorithm

Algorithm BruteForceMatch\( (T, P) \)

Input text \( T \) of size \( n \) and pattern \( P \) of size \( m \)

Output starting index of a substring of \( T \) equal to \( P \) or \(-1\)

if no such substring exists

for \( i \leftarrow 0 \) to \( n - m \)

\( j \leftarrow 0 \)

while \( j < m \wedge T[i + j] = P[j] \)

\( j \leftarrow j + 1 \)

if \( j = m \)

return \( i \) \{match at \( i \)\}

return \(-1\) \{no match anywhere\}

How to speed up the Brute Force method?

- When mismatch occurs
  - Shift \( P \) by more than one character
  - Never miss a occurrence of \( P \) in \( T \)

- Preprocessing approach
  - Preprocessing \( P \)
    - Boyer-Moore Algorithm
    - Knuth-Morris-Pratt Algorithm
  - Preprocessing \( T \)
    - Suffix trees
Boyer-Moore’s Algorithm

- The Boyer-Moore’s pattern matching algorithm is based on two principles
  - Right to Left Scan
  - Bad character shift

Right to left Scan Rule

```
1. a b a c a a b a c a a b a c a a b a c a a b a
   1
2. a b a c a a b a c a a b a c a a b a c a a b a
   4 3 2
3. a b a c a a b a c a a b a c a a b a c a a b a
   6
4. a b a c a a b a c a a b a c a a b a c a a b a
   6
```

Bad Character Shift Rule

- Shift More than one position at a time.
- When a mismatch occurs at \( T[j] = c \)
  - If \( P \) contains \( c \), shift \( P \) to align the last occurrence of \( c \) in \( P \) with \( T[j] \)
Bad Character Shift Rule

When a mismatch occurs at $T[i] = c$
- If $P$ contains $c$, shift $P$ to align the last occurrence of $c$ in $P$ with $T[i]$
- Else, shift $P$ to align $P[0]$ with $T[i + 1]$

What if we have already crossed the last occurrence?

Boyer-Moore’s Algorithm

- Example
Example

Last-Occurrence Function

- preprocesses the pattern \( P \) and the alphabet \( \Sigma \) to build the last-occurrence function \( L \) mapping \( \Sigma \) to integers, where \( L(c) \) is defined as
  - the largest index \( i \) such that \( P[i] = c \) or
  - \(-1\) if no such index exists

- Example:
  - \( \Sigma = \{a, b, c, d\} \)
  - \( P = abacab \)

- The last-occurrence function can be computed in time \( O(m + s) \), where \( m \) is the size of \( P \) and \( s \) is the size of \( \Sigma \)

The Boyer-Moore Algorithm

```plaintext
Algorithm BoyerMooreMatch(T, P, \Sigma)
1. \( L \leftarrow \text{lastOccurrenceFunction}(P, \Sigma) \)
2. \( i \leftarrow m - 1 \)
3. \( j \leftarrow m - 1 \)
4. repeat
5.   if \( T[i] = P[j] \)
6.     if \( j = 0 \)
7.       return \( i \) \{ match at \( i \) \}
8.     else
9.       \( i \leftarrow i - 1 \)
10.      \( j \leftarrow j - 1 \)
11.   else
12.     \{ character-jump \}
13.     \( i \leftarrow L[T[i]] \)
14.     \( i \leftarrow \min(j, m - i) + 1 \)
15. until \( i > n \)
16. return \(-1\) \{ no match \}
```
Analysis

- Boyer-Moore’s algorithm is significantly faster than the brute-force algorithm on English text.
- Boyer-Moore’s algorithm runs in time $O(nm + s)$.
- Example of worst case:
  - $T = aaaa...a$
  - $P = baaaS$
- The worst case may occur in images and DNA sequences but is unlikely in English text.

The Boyer-Moore algorithm

- Extended Bad character shift rule
  - then shift $P$ to the right so that the closest $x$ to the left of position $i$ in $P$ is below the mismatched $x$ in $T$.
  - The original Boyer-Moore algorithm uses the simpler bad character rule.

The Boyer-Moore algorithm

- Extended Bad character shift rule

  $T$: ------xhapp-----
  $P$: xcdqbyhaxp
  $0:12345678901$
  $R(x) = 2$
  $P$: xcdqbyhaxp

  The position of $x$ is 9, 2, 0.
  Find the top number $< j$.  

  $R(x)$ is the distance between $x$ and the closest character in $P$ that is not a match for $x$. 

  $R(x)$ is calculated as follows:

  $R(x) = m - l_1 - l_2 - l_3... - l_n$

  where $m$ is the length of $T$, $l_1, l_2, l_3... l_n$ are the lengths of the longest matching substrings ending at $x$ in $T$. 

The Knuth-Morris-Pratt algorithm

- Best known
  - For linear time for exact matching
- Preprocessing pattern \( P \)
- Compares from Left to Right

The KMP Ideas

- Shift more than one space
- Reduce comparison

The KMP Algorithm - Motivation

- When a mismatch occurs, what is the most we can shift the pattern?
- Answer: the largest prefix of \( P[0..j-1] \) that is a suffix of \( P[1..j-1] \)
KMP Failure Function

• Knuth-Morris-Pratt’s algorithm preprocesses the pattern to find matches of prefixes of the pattern with the pattern itself

• The failure function \( F(j) \) is defined as the size of the largest prefix of \( P[0..j] \) that is also a suffix of \( P[1..j] \)

\[
\begin{array}{c|cccccc}
 j & 0 & 1 & 2 & 3 & 4 & 5 \\
 P[j] & a & b & c & a & b & a \\
 F(j) & 0 & 0 & 1 & 1 & 2 & 3 \\
\end{array}
\]
The KMP Algorithm

Algorithm KMPMatch(T, P)
F ← failureFunction(P)
i ← 0
j ← 0
while i < n
if T[i] = P[j]
if j = m − 1
return i − j [match]
else
i ← i + 1
j ← j + 1
else
if j > 0
j ← F[j − 1]
else
i ← i + 1
return −1 [no match]

Example

<table>
<thead>
<tr>
<th>j</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>P[j]</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>F[j]</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>i</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>T[i]</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>c</td>
<td>a</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>j</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>P[j]</td>
<td>a</td>
<td>b</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>b</td>
<td></td>
</tr>
<tr>
<td>F[j]</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

The KMP Algorithm

- The failure function can be represented by an array and can be computed in \(O(m)\) time
- At each iteration of the while-loop, either
  - \(i\) increases by one, or
  - the shift amount \(i − j\) increases by at least one (observe that \(F[j − 1] < j\))
- Hence, there are no more than \(2m\) iterations of the while-loop
- Thus, KMP’s algorithm runs in optimal time \(O(m + n)\)
Computing the Failure Function

- The failure function can be represented by an array and can be computed in $O(n)$ time.
- The construction is similar to the KMP algorithm itself.
- At each iteration of the while-loop, either
  - $i$ increases by one, or
  - the shift amount $i-j$ increases by at least one (observe that $F(j-1) < j$)
- Hence, there are no more than $2n$ iterations of the while-loop.

**Algorithm: failureFunction**

```plaintext
Algorithm failureFunction(P)

F[0] ← 0
i ← 1
j ← 0
while i ≤ m
  if P[i] = P[j]
    {we have matched $j+1$ chars}
    F[i] ← j + 1
    i ← i + 1
    j ← j + 1
  else if j > 0 then
    {use failure function to shift $P$}
    j ← F[j-1]
  else
    F[i] ← 0 { no match }
    i ← i + 1
```
