Γιάννης Τζίτζικας

Ημερομηνία:

Πανεπιστήμιο Κρήτης, Τμήμα Επιστήμης Υπολογιστών
Άνοιξη 2008

Δομές Ευρετηρίου: Διάρθρωση Διάλεξης

• Εισαγωγή - κίνητρο
• Ανεστραμμένα Αρχεία (Inverted files)
• Δένδρα Καταλήξεων (Suffix trees)
• Αρχεία Υπογραφών (Signature files)
• Σειριακή Αναζήτηση σε Κείμενο (Sequential Text Searching)
• Απάντηση Επερωτήσεων “Ταιριάσματος Προτύπου” (Answering Pattern-Matching Queries)
Signature files (αρχεία υπογραφών)

Αρχεία Υπογραφών (Signature Files)

Κύρια σημεία:
• Δομή ευρετηρίου που βασίζεται στο hashing
• Μικρή χωρική επιβάρυνση (10%-20% του μεγέθους των κειμένων)
• Αναζήτηση = σειριακή αναζήτηση στο αρχείο υπογραφών
• Κατάλληλη για όχι πολύ μεγάλα κείμενα

Συγκεκριμένα

• Χρήση hash function που αντιστοιχεί λέξεις κειμένου σε bit masks των B bits
• Διαμέριση του κειμένου σε blocks των b λέξεων το καθένα
• Bit mask of a block = Bitwise OR of the bits masks of all words in the block
• Bit masks are then concatenated
Αρχεία Υπογραφών: Παράδειγμα

This is a text. A text has many words. Words are made from letters.

<table>
<thead>
<tr>
<th>Block 1</th>
<th>Block 2</th>
<th>Block 3</th>
<th>Block 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Text Signature</td>
<td>000101</td>
<td>110101</td>
<td>100100</td>
</tr>
</tbody>
</table>

Signature Function:

- \( h(\text{text}) = 000101 \)
- \( h(\text{many}) = 110000 \)
- \( h(\text{words}) = 100100 \)
- \( h(\text{made}) = 001100 \)
- \( h(\text{letters}) = 100001 \)

Γιατί Bitwise-OR?

Αρχεία Υπογραφών: Αναζήτηση

Έστω ότι αναζητούμε μια λέξη \( w \):

1/ \( W := h(w) \) (we hash the word to a bit mask \( W \))

2/ Compare \( W \) with all bit masks \( B_i \) of all text blocks

   If \( (W \& B_i = W) \), the text block \( i \) is candidate (may contain the word \( w \))

3/ For all candidate text blocks, perform an online traversal to verify that the word \( w \) is actually there
False drops (false hits)

- **False drop**: All bits of the $W$ are set in $B_i$ but the word $w$ is not there

\[ w=\text{«words»}, \ h(\text{«words»})=100100 \]

<table>
<thead>
<tr>
<th>Text</th>
<th>Block 1</th>
<th>Block 2</th>
<th>Block 3</th>
<th>Block 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>This is a text.</td>
<td>A text has many</td>
<td>words. Words are</td>
<td>made from letters.</td>
<td></td>
</tr>
</tbody>
</table>

Text Signature

\[ \text{h(text)}=000101 \]
\[ \text{h(many)}=110000 \]
\[ \text{h(words)}=100100 \]
\[ \text{h(made)}=001100 \]
\[ \text{h(letters)}=100001 \]

Signature Function

\[ 000101 \quad 110101 \quad 100100 \quad 101101 \]

Διαμόρφωση (Configuration) υπογραφών

- **Σχεδιαστικοί στόχοι**:
  - Μείωσε την πιθανότητα εμφάνισης false drops
  - Κράτησε το μέγεθος του αρχείου υπογραφών μικρό
    - δεν έχουμε κανένα false drop αν $b=1$ και $B=\log_2(V)$

- **Παράμετροι**:
  - $B$ (το μέγεθος των bit mask)
  - $L$ ($L<B$) το πλήθος των bit που είναι 1 (σε κάθε $h(w)$)

- **The (space)-(false drop probability) tradeoff**:
  - 10% space overhead => 2% false drop probability
  - 20% space overhead => 0.046% false drop probability
Signature files: Phrase and Proximity Queries

- **Good for phrase searches and reasonable proximity queries**
  - this is because all the words must be present in a block in order for that block to hold the phrase or the proximity query. Hence the OR of all the query masks is searched

- **Remark:**
  - no other patterns (e.g. range queries) can be searched in this scheme

- **Μέγεθος αρχείου υπογραφών:**
  - bit masks of each block plus one pointer for each block
    - (pointing to the corresponding position at the original text)

- **Συντήρηση αρχείου υπογραφών:**
  - Η προσθήκη/διαγραφή αρχείων αντιμετωπίζεται εύκολα
    - προσθέτονται/διαγράφονται τα αντίστοιχα bit masks
Phrase/Proximity Queries and **Block Boundaries**

q=<information retrieval>

Text blocks

![Illustration of text blocks with "Information retrieval" highlighted and "j-1 common words" indicated.]

(πρόβλημα! Μπορούμε όμως να το λύσουμε με επικαλυπτόμενα blocks)

Overlapping blocks

![Illustration showing overlapping blocks with "Information retrieval" highlighted.]

For j-proximity queries

![Diagram illustrating a query and matching blocks with "j-1 common words" indicated.]

Δομές Ευρετηρίου: Διάρθρωση Διάλεξης

- Εισαγωγή - κίνητρο
- Inverted files (ανεστραμμένα αρχεία)
- Suffix trees (δένδρα καταλήξεων)
- Signature files (αρχεία υπογραφών)
- **Sequential Text Searching**
- Answering Pattern-Matching Queries
find the first occurrence (or all occurrences)
of a string (or pattern) \( p \) (of length \( m \)) in a string \( s \) (of length \( n \))

Commonly, \( n \) is much larger than \( m \).

**Χρήσεις:**
- Για εύρεση των εγγράφων που περιέχουν μια λέξη (αν δεν έχουμε ευρετήριο).
- Στην περίπτωση που έχουμε ανεστραμμένο ευρετήριο με block addressing.
- Στην περίπτωση που έχουμε αρχείο υπογραφών για να βεβαιωθούμε ότι ένα match δεν είναι false drop.

**Sequential Text Searching Algorithms**

- Brute-Force Algorithm
- Knuth-Morris-Pratt
- Boyer-Moore family
Brute-Force Algorithm

Brute-Force (BF), or sequential text searching:

- **Try all** possible positions in the text. For each position verify whether the pattern matches at that position.

- Since there are \(O(n)\) text positions and each one is examined at \(O(m)\) worst-case cost, the worst-case of brute-force searching is \(O(nm)\).

Naive-String-Matcher(S,P)

\[
\begin{align*}
  n & := \text{length}(S) \\
  m & := \text{length}(P) \\
  \text{for } i & = 0 \text{ to } n-m \text{ do} \\
  & \quad \text{if } P[1..m] = S[i+1 .. i+m] \text{ then} \\
  & \quad \quad \text{return "Pattern occurs at position } i\text{"} \\
  & \quad \text{fi} \\
  \text{od}
\end{align*}
\]

The naive string matcher needs **worst case** running time \(O((n-m+1) m)\)
For \(n = 2m\) this is \(O(n^2)\)
Its **average case** is \(O(n)\) (since on random text a mismatch is found after \(O(1)\) comparisons on average)
The naive string matcher is not optimal, since string matching can be done in time \(O(m + n)\)
Knuth-Morris-Pratt & Boyer-Moore

- Faster algorithms that base on moving (scanning) window

- General idea:
  - They employ a **window** of length \( m \) which is **slid** over the text.
  - It is **checked** whether the text in the window is equal to the pattern (if it is, the window position is reported as a match).
  - Then, the window is **shifted forward**.

- They algorithms differ in the way they check and slide (move) the window.
It does not try all window positions as BF does. Instead, it reuses information from previous checks.

Knuth-Morris-Pratt (KMP) [1970]

- The pattern $p$ is preprocessed to build a table called $next$.

- The $next$ table at position $j$ says which is the longest proper prefix of $p[1..j-1]$ which is also a suffix and the characters following prefix and suffix are different.

- Hence $j\text{-next}[j]-1$ window positions can be safely skipped if the characters up to $j-1$ matched and the $j$-th did not.
**KMP: the next table**

\[ \text{next}[j] = \text{longest proper prefix of } p[1..j-1] \text{ which is also a suffix and the characters following prefix and suffix are different} \]

<table>
<thead>
<tr>
<th>j</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>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>p[j]</td>
<td>a</td>
<td>b</td>
<td>r</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>d</td>
<td>a</td>
<td>b</td>
<td>r</td>
<td>a</td>
</tr>
<tr>
<td>next[j]</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
</tbody>
</table>
KMP: the next table

\[ \text{next}[j] = \text{longest proper prefix of } p[1..j-1] \text{ which is also a suffix and the characters following prefix and suffix are different} \]

<table>
<thead>
<tr>
<th>j</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>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>p[j]</td>
<td>a</td>
<td>b</td>
<td>r</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>d</td>
<td>a</td>
<td>b</td>
<td>r</td>
<td>a</td>
</tr>
<tr>
<td>next[j]</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

CS463 - Information Retrieval Systems
Yannis Tzitzikas, U. of Crete
### KMP: the next table

**next[j]** = longest proper prefix of \( p[1..j-1] \) which is also a suffix and the characters following prefix and suffix are different

<table>
<thead>
<tr>
<th>j</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>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p[j] )</td>
<td>a</td>
<td>b</td>
<td>r</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>d</td>
<td>a</td>
<td>b</td>
<td>r</td>
<td>a</td>
</tr>
<tr>
<td>next[j]</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
</tbody>
</table>

### KMP: the next table

**next[j]** = longest proper prefix of \( p[1..j-1] \) which is also a suffix and the characters following prefix and suffix are different

<table>
<thead>
<tr>
<th>j</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>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p[j] )</td>
<td>a</td>
<td>b</td>
<td>r</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>d</td>
<td>a</td>
<td>b</td>
<td>r</td>
<td>a</td>
</tr>
<tr>
<td>next[j]</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
</tbody>
</table>
**KMP: the next table**

\[ \text{next}[j] = \text{longest proper prefix of } p[1..j-1] \text{ which is also a suffix and the characters following prefix and suffix are different} \]

<table>
<thead>
<tr>
<th>( j )</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>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p[j] )</td>
<td>a</td>
<td>b</td>
<td>r</td>
<td>a</td>
<td>c</td>
<td>d</td>
<td>a</td>
<td>b</td>
<td>r</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>next[j]</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
</tbody>
</table>
$next[j] = \text{longest proper prefix of } p[1..j-1] \text{ which is also a suffix and the characters following prefix and suffix are different}$
KMP: the next table

\( \text{next}[j] \) = longest proper prefix of \( p[1..j-1] \) which is also a suffix and the characters following prefix and suffix are different.
Exploiting the next table

\[ next[j] = \text{longest proper prefix of } p[1..j-1] \text{ which is also} \]
a suffix and the characters following prefix and suffix are different

\[
\begin{array}{ccccccccccccc}
  j & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\
p[j] & a & b & r & a & c & a & d & a & b & r & a \\
next[j] & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 4 \\
j-next[j]-1 & 0 & 1 & 2 & 3 & 3 & 5 & 5 & 7 & 8 & 9 & 10 & 7 \\
\end{array}
\]

- \[ j \text{-next}[j]-1 \] window positions can be safely skipped if the characters up to \( j-1 \) matched and the \( j \)-th did not.

Example: match until 2nd char

\[
\begin{array}{ccccccccccccc}
  j & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\
p[j] & a & b & r & a & c & a & d & a & b & r & a \\
next[j] & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 4 \\
j-next[j]-1 & 0 & 1 & 2 & 3 & 3 & 5 & 5 & 7 & 8 & 9 & 10 & 7 \\
\end{array}
\]

\[
\begin{array}{cccccccccc}
s & a & a & r & i & c & a & b & r & a & c & a \\
p & a & b & r & a & c & a & d & a & b & r & a \\
 & a & b & r & a & c & a & d & a & b & r & a \\
\end{array}
\]
Example: match until 3rd char

<table>
<thead>
<tr>
<th>j</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>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>p[j]</td>
<td>a</td>
<td>b</td>
<td>r</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>d</td>
<td>a</td>
<td>b</td>
<td>r</td>
<td>a</td>
</tr>
<tr>
<td>next[j]</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>j-next[j]-1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
</tr>
</tbody>
</table>

Example: match until 7th char

<table>
<thead>
<tr>
<th>j</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>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>p[j]</td>
<td>a</td>
<td>b</td>
<td>r</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>d</td>
<td>a</td>
<td>b</td>
<td>r</td>
<td>a</td>
</tr>
<tr>
<td>next[j]</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>j-next[j]-1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>7</td>
</tr>
</tbody>
</table>

CS463 - Information Retrieval Systems
Yannis Tzitzikas, U. of Crete
**Example: pattern matched**

<table>
<thead>
<tr>
<th>j</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>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>p[j]</td>
<td>a</td>
<td>b</td>
<td>r</td>
<td>a</td>
<td>c</td>
<td>a</td>
<td>d</td>
<td>a</td>
<td>b</td>
<td>r</td>
<td>a</td>
</tr>
<tr>
<td>next[j]</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>j-next[j]-1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
</tr>
</tbody>
</table>

**KMP: Complexity**

- Since at each text comparison the window or the text pointer advance by at least one position, the algorithm performs at most \(2n\) comparisons (for the case where \(m=n\)), and at least \(n\).
- The overall complexity is \(O(m+n)\)
  - The worst case is exactly \(n+m\) for finding the 1\(^{st}\) occurrence
- Remarks:
  - We shouldn’t however forget the cost for building the \(next\) table.
  - On average is it not much faster than BF
Finite-Automaton-Matcher

- For every pattern of length $m$ there exists an automaton with $m+1$ states that solves the pattern matching problem.

- *KMP is actually a Finite-Automaton-Matcher*
Finite Automata (επανάληψη)

A deterministic finite automaton $M$ is a 5-tuple $(Q,q_0,A,\Sigma,\delta)$, where
- $Q$ is a finite set of states
- $q_0 \in Q$ is the start state
- $A \subseteq Q$ is a distinguished set of accepting states
- $\Sigma$, is a finite input alphabet,
- $\delta: Q \times \Sigma \rightarrow Q$ is called the transition function of $M$

Let $\phi: \Sigma \rightarrow Q$ be the final-state function defined as:

For the empty string $\varepsilon$ we have: $\phi(\varepsilon) := q_0$
For all $a \in \Sigma$, $w \in \Sigma^*$ define $\phi(wa) := \delta(\phi(w), a)$

$M$ accepts $w$ if and only if: $\phi(w) \in A$

Example (I)

$Q$ is a finite set of states
$q_0 \in Q$ is the start state
$Q$ is a set of accepting states
$\Sigma$: input alphabet
$\delta: Q \times \Sigma \rightarrow Q$: transition function

$p=«abba»$

<table>
<thead>
<tr>
<th>States</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
</tbody>
</table>

input: [a, b, a, b, b, a, b, b, a, a]
Example (II)

Q is a finite set of states

$q_0 \in Q$ is the start state

Q is a set of accepting states

$\Sigma$: input alphabet

$\delta$: $Q \times \Sigma \rightarrow Q$: transition function

<table>
<thead>
<tr>
<th>input</th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>state</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>

Example (III)

Q is a finite set of states

$q_0 \in Q$ is the start state

Q is a set of accepting states

$\Sigma$: input alphabet

$\delta$: $Q \times \Sigma \rightarrow Q$: transition function

<table>
<thead>
<tr>
<th>input</th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>state</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>
Example (IV)

Q is a finite set of states
q₀ ∈ Q is the start state
Q is a set of accepting states
Σ: input alphabet
δ: Q × Σ → Q: transition function

<table>
<thead>
<tr>
<th>input</th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>state</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>

Example (V)

Q is a finite set of states
q₀ ∈ Q is the start state
Q is a set of accepting states
Σ: input alphabet
δ: Q × Σ → Q: transition function

<table>
<thead>
<tr>
<th>input</th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>state</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>
Example (VI)

Q is a finite set of states

$q_0 \in Q$ is the start state

Q is a set of accepting states

$\Sigma$: input alphabet

$\delta$: $Q \times \Sigma \rightarrow Q$: transition function

<table>
<thead>
<tr>
<th>input</th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>state</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

Example (VII)

Q is a finite set of states

$q_0 \in Q$ is the start state

Q is a set of accepting states

$\Sigma$: input alphabet

$\delta$: $Q \times \Sigma \rightarrow Q$: transition function

<table>
<thead>
<tr>
<th>input</th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>state</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>
Example (VIII)

Q is a finite set of states
q_0 \in Q is the start state
Q is a set of accepting states
\Sigma: input alphabet
\delta: Q \times \Sigma \rightarrow Q: transition function

p=«abba»

Example (IX)

Q is a finite set of states
q_0 \in Q is the start state
Q is a set of accepting states
\Sigma: input alphabet
\delta: Q \times \Sigma \rightarrow Q: transition function

p=«abba»
Example (X)

Q is a finite set of states
q₀ ∈ Q is the start state
Q is a set of accepting states
Σ: input alphabet
δ: Q × Σ → Q: transition function

<table>
<thead>
<tr>
<th>input</th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>state</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

Example (XI)

Q is a finite set of states
q₀ ∈ Q is the start state
Q is a set of accepting states
Σ: input alphabet
δ: Q × Σ → Q: transition function

<table>
<thead>
<tr>
<th>input</th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>state</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>
Finite-Automaton-Matcher

For every pattern $P$ of length $m$ there exists an automaton with $m+1$ states that solves the pattern matching problem with the following algorithm:

Finite-Automaton-Matcher($T, \delta, P$)

$n := \text{length}(T)$
$q := 0$  // initial state

for $i = 1$ to $n$ do

$q := \delta(q, T[i])$  // transition to the next state

if $q = m$ then  // if we reached the state $m$ (which is the final)

return “Pattern occurs at position ” $i-m$

fi

od

Computing the Transition Function:
It is actually the idea of KMP
How to Compute the Transition Function?

- Let $P_k$ denote the first $k$ letter string of $P$ (i.e. the prefix of $P$ with length $k$)

```plaintext
Compute-Transition-Function($P$, $\Sigma$)
  $m := \text{length}(P)$
  for $q = 0$ to $m$ do
    for each character $a \in \Sigma$ do
      $k := 1 + \text{min}(m, q+1)$
      repeat
        $k := k - 1$
      until $P_k$ is a suffix of $P_a$
    $\delta(q, a) := k$
  od
od
```

Boyer-Moore (BM)
Boyer-Moore (BM) [1975]

- **Motivation**
  - KMP yields genuine benefits only if a mismatch as preceded by a partial match of some length
    - *only in this case* the pattern slides more than one position
  - Unfortunately, this is the exception rather than the rule
    - *matches occur much more seldom than mismatches*
- **The idea**
  - start comparing characters at the **end of the pattern** rather than at the beginning
  - like in KMP, a pattern is pre-processed

---

**Boyer-Moore: The idea by an example**

Start comparing at the end

First wrong letter! Do a large shift!

What’s this? There is no “a” in the search pattern. We can shift m+1 letters

An “a” again...

Bingo! Do another large shift!

That’s it! 10 letters compared and ready!
Sequential Text Searching

Synopsis

- **Brute Force Algorithm**
  - O(n^2) running time (worst case)
- **KMP ~ Finite Automaton Matcher**
  - Let a (finite) automaton do the job
    - Cost: cost to construct the automaton plus the cost to “consume” the string s
    - O(m+n) running time (worst case)
    - m: for constructing the next table
    - n: for searching the text
- **BM Algorithm**
  - Bad letters allow us to jump through the text
  - Faster in practice
  - O(nm) running time (worst case)
  - O(n log(m)/m) average time

find the first occurrence (or all occurrences) of a string (or pattern) \( p \) (of length \( m \)) in a string \( s \) (of length \( n \))

Other string searching algorithms

- Rabin-Karp
- Shift-Or (it is sketched in the Modern Information Retrieval Book)
- ...and many others ..
- ..
For more

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Preprocessing time</th>
<th>Matching time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naïve string search algorithm</td>
<td>0 (no preprocessing)</td>
<td>$\Theta(n m)$</td>
</tr>
<tr>
<td>Rabin-Karp string search algorithm</td>
<td>$\Omega(m)$</td>
<td>average $\Theta(n+m)$, worst $\Theta(n m)$</td>
</tr>
<tr>
<td>Finite state automaton based search</td>
<td>$\Theta(m</td>
<td>\Sigma</td>
</tr>
<tr>
<td>Knuth-Morris-Pratt algorithm</td>
<td>$\Theta(m)$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>Boyer-Moore string search algorithm</td>
<td>$\Theta(m +</td>
<td>\Sigma</td>
</tr>
<tr>
<td>Bitap algorithm (shift-or, shift-and, Baeza-Yates-Gonnet)</td>
<td>$\Theta(m +</td>
<td>\Sigma</td>
</tr>
</tbody>
</table>

EXACT STRING MATCHING ALGORITHMS, Christian Charras - Thierry Lecroq,
- http://www-igm.univ-mlv.fr/~lecroq/string/index.html (it includes animations in Java)

Δομές Ευρετηρίου: Διάρθρωση Διάλεξης

- Εισαγωγή - κίνητρο
- Inverted files (ανεστραμμένα αρχεία)
- Suffix trees (δένδρα καταλήξεων)
- Signature files (αρχεία υπογραφών)
- Sequential Text Searching
- Answering Pattern-Matching Queries
Answering Pattern Matching Queries

• Searching Allowing Errors (Levenshtein distance)
• Searching using Regular Expressions

Searching Allowing Errors

• Δεδομένα:
  – Ένα κείμενο (string) T, μήκους n
  – Ένα pattern P μήκους m
  – K επιτρεπόμενα σφάλματα

• Ζητούμενο:
  – Βρες όλες τις θέσεις του κειμένου όπου το pattern P εμφανίζεται με το πολύ k σφάλματα

Remember: Edit (Levenstein) Distance:
Minimum number of character deletions, additions, or replacements needed to make two strings equivalent.

  “misspell” to “mispell” is distance 1
  “misspell” to “mistell” is distance 2
  “misspell” to “misspelling” is distance 3
Searching Allowing Errors

- **Naïve algorithm**
  - Produce all possible strings that could match $P$ (assuming $k$ errors) and search each one of them on $T$

Searching Allowing Errors: Solution using **Dynamic Programming**

- Dynamic Programming is the class of algorithms, which includes the most commonly used algorithms in speech and language processing.

- Among them the **minimum edit distance algorithm for spelling error correction**.

- Intuition:
  - *a large problem can be solved by properly combining the solutions to various subproblems.*
Searching Allowing Errors: Solution using Dynamic Programming (II)

**Problem Statement:** \( T[n] \) text string, \( P[m] \) pattern, \( k \) errors

Example: \( T = \text{“surgery”}, \ P = \text{“survey”}, \ k=2 \)

*To explain the algorithm we will use a \( m \times n \) matrix \( C \)

one row for each char of \( P \), one column for each char of \( T \)

(latter on we shall see that we need less space)

<table>
<thead>
<tr>
<th>( )</th>
<th>( s )</th>
<th>( u )</th>
<th>( r )</th>
<th>( g )</th>
<th>( e )</th>
<th>( r )</th>
<th>( y )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( s )</td>
<td>( 0 )</td>
<td>( 0 )</td>
<td>( 0 )</td>
<td>( 0 )</td>
<td>( 0 )</td>
<td>( 0 )</td>
<td>( 0 )</td>
</tr>
<tr>
<td>( u )</td>
<td>( 1 )</td>
<td>( 0 )</td>
<td>( 0 )</td>
<td>( 0 )</td>
<td>( 0 )</td>
<td>( 0 )</td>
<td>( 0 )</td>
</tr>
<tr>
<td>( r )</td>
<td>( 2 )</td>
<td>( 0 )</td>
<td>( 0 )</td>
<td>( 0 )</td>
<td>( 0 )</td>
<td>( 0 )</td>
<td>( 0 )</td>
</tr>
<tr>
<td>( v )</td>
<td>( 3 )</td>
<td>( 0 )</td>
<td>( 0 )</td>
<td>( 0 )</td>
<td>( 0 )</td>
<td>( 0 )</td>
<td>( 0 )</td>
</tr>
<tr>
<td>( e )</td>
<td>( 4 )</td>
<td>( 0 )</td>
<td>( 0 )</td>
<td>( 0 )</td>
<td>( 0 )</td>
<td>( 0 )</td>
<td>( 0 )</td>
</tr>
<tr>
<td>( y )</td>
<td>( 5 )</td>
<td>( 0 )</td>
<td>( 0 )</td>
<td>( 0 )</td>
<td>( 0 )</td>
<td>( 0 )</td>
<td>( 0 )</td>
</tr>
</tbody>
</table>

Searching Allowing Errors: Solution using Dynamic Programming (III)

\( T = \text{“surgery”}, \ P = \text{“survey”}, \ k=2 \)

οι γραμμές του \( C \) εκφράζουν πόσα γράμματα του pattern έχουμε ήδη καταναλώσει (στη 0-γραμμή τίποτα, στη \( m \)-γραμμή ολόκληρο το pattern)

\( C[0,j] := 0 \) for every column \( j \)

(no letter of \( P \) has been consumed)

\( C[i,0] := i \) for every row \( i \)

(i chars of \( P \) have been consumed, pointer of \( T \) at 0. So \( i \) errors (insertions) so far)
Searching Allowing Errors: Solution using **Dynamic Programming** (IV)

\[
\text{if } P[i] = T[j] \text{ THEN } C[i,j] := C[i-1,j-1] \\
// \text{ εγινε match αρα τα “λάθη” ήταν όσα και πριν}
\]

**Else** \(C[i,j] := 1 + \text{ min of:}

- \(C[i-1,j]\) \\
  - // i-1 chars consumed P, j chars consumed of T \\
  - // ~delete a char from T

- \(C[i,j-1]\) \\
  - // i chars consumed P, j-1 chars consumed of T \\
  - // ~ delete a char from P

- \(C[i-1,j-1]\) \\
  - // i-1 chars consumed P, j-1 chars consumed of T \\
  - // ~ character replacement

---

**Searching Allowing Errors: Solution using Dynamic Programming: Example**

- **T** = “surgery”, **P** = “survey”, k=2

<table>
<thead>
<tr>
<th></th>
<th>s</th>
<th>u</th>
<th>r</th>
<th>g</th>
<th>e</th>
<th>r</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>s</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>u</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>r</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>v</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>e</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>y</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>
Solution using **Dynamic Programming**: Example

- \( T = \text{“surgery”}, P = \text{“survey”}, k=2 \)

<table>
<thead>
<tr>
<th></th>
<th>s</th>
<th>u</th>
<th>r</th>
<th>g</th>
<th>e</th>
<th>r</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>u</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>r</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>v</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>e</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>y</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>s</th>
<th>u</th>
<th>r</th>
<th>g</th>
<th>e</th>
<th>r</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>u</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>r</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>v</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>e</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>y</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>
Solution using **Dynamic Programming**: Example

- T = “surgery”, P = “survey”, k=2

<table>
<thead>
<tr>
<th></th>
<th>s</th>
<th>u</th>
<th>r</th>
<th>g</th>
<th>e</th>
<th>r</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>s</strong></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td><strong>u</strong></td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td><strong>r</strong></td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td><strong>v</strong></td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td><strong>e</strong></td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td><strong>y</strong></td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

Solution using **Dynamic Programming**: Example

- T = “surgery”, P = “survey”, k=2

<table>
<thead>
<tr>
<th></th>
<th>s</th>
<th>u</th>
<th>r</th>
<th>g</th>
<th>e</th>
<th>r</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>s</strong></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td><strong>u</strong></td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td><strong>r</strong></td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td><strong>v</strong></td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td><strong>e</strong></td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td><strong>y</strong></td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

1 +
Solution using **Dynamic Programming: Example**

- T = “surgery”, P = “survey”, k=2

<table>
<thead>
<tr>
<th>T</th>
<th>s</th>
<th>u</th>
<th>r</th>
<th>g</th>
<th>e</th>
<th>r</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>s</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>u</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>r</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>v</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>e</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>y</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

1 +

---

Solution using **Dynamic Programming: Example**

- T = “surgery”, P = “survey”, k=2

<table>
<thead>
<tr>
<th>T</th>
<th>s</th>
<th>u</th>
<th>r</th>
<th>g</th>
<th>e</th>
<th>r</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>s</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>u</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>r</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>v</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>e</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>y</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

1 +
Solution using **Dynamic Programming**: Example

- \(T = \text{“surgery”}, \ P = \text{“survey”}, \ k=2\)

<table>
<thead>
<tr>
<th></th>
<th>s</th>
<th>u</th>
<th>r</th>
<th>g</th>
<th>e</th>
<th>r</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>(1)</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>(2)</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>(3)</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>(4)</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>(5)</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>(6)</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

Bold entries indicate matching positions.

- Cost: \(O(mn)\) time where \(m\) and \(n\) are the lengths of the two strings being compared.

- Παρατήρηση: η πολυπλοκότητα είναι ανεξάρτητη του \(k\)
Solution using **Dynamic Programming**: Example

- \( T = "surgery", P = "survey", k=2 \)

  \[
  \begin{array}{cccccccc}
  & s & u & r & g & e & r & y \\
  \hline
  s & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
  u & 1 & 0 & 1 & 1 & 1 & 1 & 1 \\
  r & 2 & 1 & 0 & 1 & 2 & 2 & 2 \\
  v & 3 & 2 & 1 & 0 & 1 & 2 & 2 \\
  e & 4 & 3 & 2 & 1 & 1 & 2 & 3 \\
  y & 5 & 4 & 3 & 2 & 2 & 1 & 3 \\
  \end{array}
  \]

- **Cost**: \( O(mn) \) time where \( m \) and \( n \) are the lengths of the two strings being compared.
- **\( O(m) \) space** as we need to keep only the previous column stored
  - So we don't have to keep a \( mxn \) matrix

Εφαρμογή στο groogle

![grOOGLE](image)
Searching Allowing Errors
Solution with a Nondeterministic Automaton

- At each iteration, a new text character is read and automaton changes its state.
- Every row denotes the number of errors seen
  - (0 for the first row, 1 for the second, and so on)
- Every column represents matching to pattern up to a given position.
Searching Allowing Errors: Solution with a Nondeterministic Automaton

- **Horizontal** arrows represent matching a document.
- **Vertical** arrows represent insertions into pattern.
- **Solid diagonal** arrows represent replacements (they are unlabelled: this means that they match any character).
- **Dashed diagonal** arrows represent deletion in the pattern ($\varepsilon$: empty).

- Search time is $O(n)$
  - άρα η μέθοδος αυτή είναι πιο αποδοτική από την τεχνική με δυναμικό προγραμματισμό (που ήταν $O(mn)$)
- However, if we convert NDFA into a DFA then it will be huge in size
  - An alternative solution is **BIT-Parallelism**
Searching using **Regular Expressions**

**Classical Approach**

(a) Build a ND Automaton

(b) Convert this automaton to deterministic form

(a) Build a ND Automaton

Size $O(m)$ where $m$ the size of the regular expression

\[ \Pi.\chi. \text{regex} = b \ b^* \ (b \ | \ b^* \ a) \]
Searching using Regular Expressions (II)

(b) Convert this automaton to deterministic form

- It can search any regular expression in $O(n)$ time where $n$ the size of text
- However, its size and construction time can be exponential in $m$, i.e. $O(m^{2^m})$.

$$b \ b^* \ (b \ | \ b^* \ a) = (b \ b^* \ b \ | \ b \ b^* \ b^* \ a) = (b \ b \ b^* \ | \ b \ b^* a)$$

![Automaton Diagrams]

Bit-Parallelism to avoid constructing the deterministic automaton (NFA Simulation)

Pattern Marching Queries and Index Structures
Pattern Matching Using **Inverted Files**

- Previously, we learned how to evaluate queries with criteria such as **Edit Distance**, **RegExpr**, referring to the documents.
- What if we already have an **Inverted File**?
  - We search the **Lexicon** instead of the documents (much smaller in size)
  - We find the matching words
  - We merge the lists of occurrences (occurrence lists) of the matching words.

If **block addressing** is used, the search must be completed with a sequential search over the blocks.

- Technique of inverted files is not able to efficiently find approximate matches or regular expressions that span many words.

<table>
<thead>
<tr>
<th>Index terms</th>
<th>D₁, 3</th>
<th>D₂, 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>computer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>database</td>
<td></td>
<td></td>
</tr>
<tr>
<td>science</td>
<td></td>
<td></td>
</tr>
<tr>
<td>system</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

Pattern Matching Using **Suffix Trees**

- What if we already have a **Suffix Tree**?
- We can evaluate queries here instead of the documents;

**Suffix Trie**

If **block addressing** is used, the search must be completed with a sequential search over the blocks.
Pattern Matching Using **Suffix Trees (II)**

If the suffix trees index all text positions (not just word beginnings) it can search for words, prefixes, suffixes and sub-stings with the same search algorithm and cost described for word search.

Indexing all text positions normally makes the suffix array size **10 times or more the text size**.

![Suffix Tree Example](image)

cacao

versus

Pattern Matching Using **Suffix Trees (III)**

- **Range queries** are easily solved by just searching both extremes in the trie and then collecting all the leaves lie in the middle.

“letter” < q < “many”

![Range Query Example](image)
Pattern Matching Using **Suffix Trees (IV)**

- **Regular expressions** can be searched in the suffix tree. The algorithm simply simulates sequential searching of the regular expression

\[ q = ma^* \]

[Diagram of a suffix tree]

Some Software Packages for String Searching

- A recent software package that implements several recently emerged string matching algorithms (code available in C++) is available at:
  - [http://flamingo.ics.uci.edu/releases/1.0/](http://flamingo.ics.uci.edu/releases/1.0/)

- **StringSearch**: Searching algorithms written in Java (includes implementations of the Boyer-Moore and the Shift-Or (bit-parallel) algorithms). These algorithms are easily five to ten times faster than the naïve implementation found in java.lang.String. Available at
  - [http://johannburkard.de/software/stringsearch/](http://johannburkard.de/software/stringsearch/)

- **Algorithm FJC in Java**
  - [http://www.sfu.ca/~cjennings/fjs/index.html](http://www.sfu.ca/~cjennings/fjs/index.html)
References

- Some slides were based on the slides of
  - Christian Schindelhauer (University of Paderborn)

Δομές Ευρετηρίου: Διάρθρωση Διάλεξης

- Εισαγωγή - κίνητρο
- Inverted files (ανεστραμμένα αρχεία)
- Suffix trees (δένδρα καταλήξεων)
- Signature files (αρχεία υπογραφών)
- Sequential Text Searching
- Answering Pattern-Matching Queries
  - directly on documents
    - Searching Allowing Errors
    - Searching using Regular Expressions
  - on indices (inverted files and suffix trees)