Δομές Ευρετηρίου: Διάρθρωση Διάλεξης

- Εισαγωγή - κίνητρο
- Ανεστραμμένα Αρχεία (Inverted files)
- Δένδρα Καταλήξεων (Suffix trees)
- Αρχεία Υπογραφών (Signature files)
- Σειρακή Αναζήτηση σε Κείμενο (Sequential Text Searching)
- Απάντηση Επερωτήσεων "Ταιριάσματος Προτύπου" (Answering Pattern-Matching Queries)

Σημαντικότερες θέματα

1. Δομές Ευρετηρίου
2. Αναζήτηση σε Κείμενο
3. Αναζήτηση με δόμημα
4. Αναζήτηση με τοποθετημένες λέξεις

Αρχεία Υπογραφών (Signature Files)

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

Συγκεκριμένα

| b=3 | B=6 |
| Block 1 | Block 2 | Block 3 | Block 4 |
| Text | This is a text | A text has many words | Words are made from letters |
| Text Signature | 000101 | 110101 | 100100 | 101101 |

Αρχεία Υπογραφών: Παράδειγμα

| b=3 | B=6 |
| Text | This is a text | A text has many words | Words are made from letters |
| Text Signature | 000101 | 110101 | 100100 | 101101 |

Εστω ότι οι λέξεις του κειμένου είναι 3 λέξεις με 3 λέξεις κάθε λέξης.

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

2/ Compare W with all bit masks Bi of all text blocks
   If (W & Bi = 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 Bi but the word w is not there

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

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>

**Text**

**Signature Function**

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

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

**Διαμόρφωση (Configuration) υπογραφών**

- Σχεδιαστικοί στόχοι:
  - Μείωση της πιθανότητας εμφάνισης false drops
  - Κράτηση του μέγεθους του αρχείου υπογραφών μικρό

\[ \text{Δεν έχουμε κανένα false drop αν } b=1 \text{ και } B=\log_2(V) \]

- Παράμετροι:
  - \( B \) (το μέγεθος των bit mask)
  - \( L \) (\( L<\log_2(V) \) και \( \text{πλήθος του bit που είναι } 1 \) σε κάθε \( h(w) \))

- The (space)-(false drop probability) tradeoff:

\[ \text{10% space overhead } \Rightarrow 2\% \text{ false drop probability} \]
\[ \text{20% space overhead } \Rightarrow 0.046\% \text{ false drop probability} \]

**Αρχεία Υπογραφών: Άλλες Παρατηρήσεις**

- Μέγεθος αρχείου υπογραφών:
  - bit masks of each block plus one pointer for each block

- Συνήρτηση αρχείων υπογραφών:
  - Η προσθήκη/διαγραφή αρχείων αντιμετωπίζεται εύκολα

**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

**Δομές Ευρετηρίου: Διάρθρωση Διάλεξης**

- Εισαγωγή - κίνητρο
  - Inverted files (ανεστραμμένα αρχεία)
  - Suffix trees (δένδρα καταλήξεων)

**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 \( m \).

**Uses:**
- For finding words in a text (if we do not have an index).
- In the case where we have a fragmented index with block addressing.
- In the case where we have a signature file to ensure that a match is not a false drop.

---

**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**

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

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**

**Knuth-Morris-Pratt & Boyer-Moore**

- 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 **shift** the window by the difference between the current mismatch and the length of the pattern.

---

- 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 **shift** the window by the difference between the current mismatch and the length of the pattern.
• 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-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

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

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

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

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

KMP: the next table

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

\[
\begin{array}{cccccccccccc}
  j & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\
p[j] & a & b & r & a & c & a & d & a & b & r & a \\
\text{next}[j] & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 4 \\
\end{array}
\]
**Exploiting 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><code>p[j]</code></td>
<td>abracadabra</td>
<td>abracadabra</td>
<td>abracadabra</td>
<td>abracadabra</td>
<td>abracadabra</td>
<td>abracadabra</td>
<td>abracadabra</td>
<td>abracadabra</td>
<td>abracadabra</td>
<td>abracadabra</td>
<td>abracadabra</td>
</tr>
</tbody>
</table>
| `next[j]` | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 4

`j-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**

- `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><code>p[j]</code></td>
<td>abracadabra</td>
<td>abracadabra</td>
<td>abracadabra</td>
<td>abracadabra</td>
<td>abracadabra</td>
<td>abracadabra</td>
<td>abracadabra</td>
<td>abracadabra</td>
<td>abracadabra</td>
<td>abracadabra</td>
<td>abracadabra</td>
</tr>
</tbody>
</table>
| `next[j]` | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 4

`j-next[j]-1` = 0 1 2 3 5 7 8 9 10 7

**Example: match until 3rd char**

- `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><code>p[j]</code></td>
<td>abracadabra</td>
<td>abracadabra</td>
<td>abracadabra</td>
<td>abracadabra</td>
<td>abracadabra</td>
<td>abracadabra</td>
<td>abracadabra</td>
<td>abracadabra</td>
<td>abracadabra</td>
<td>abracadabra</td>
<td>abracadabra</td>
</tr>
</tbody>
</table>
| `next[j]` | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 4

`j-next[j]-1` = 0 1 2 3 5 7 8 9 10 7

**Example: match until 7th char**

- `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><code>p[j]</code></td>
<td>abracadabra</td>
<td>abracadabra</td>
<td>abracadabra</td>
<td>abracadabra</td>
<td>abracadabra</td>
<td>abracadabra</td>
<td>abracadabra</td>
<td>abracadabra</td>
<td>abracadabra</td>
<td>abracadabra</td>
<td>abracadabra</td>
</tr>
</tbody>
</table>
| `next[j]` | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 4

`j-next[j]-1` = 0 1 2 3 5 7 8 9 10 7
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>0</td>
<td>1</td>
<td>0</td>
<td>1</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>4</td>
<td>5</td>
<td>6</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 (and at least $n$).
- We shouldn’t however forget the cost for building the next table.
- The overall it is $O(m+n)$
- 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 $(Q, q_0, \Sigma, \delta, A)$, 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 $\phi(w) \in A$
Example (II)

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

Example (III)

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

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

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

Example (VI)

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

Example (VII)

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
Example (VIII)

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

Example (IX)

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

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

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

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, δ, P)

n := length(T)
q := 0 // initial state
for i = 1 to n do
q := δ(q, T[i]) // transition to the next state
if q = m then // if we reached the state m
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 $k$)

Compute-Transition-Function($P$, $\Sigma$)

$m := \text{length}(P)$

for $q = 0$ to $m$ do

for each character $a \in \Sigma$ do

$k := 1 + \min(m, q+1)$

repeat

$k := k - 1$

until $P_k$ is a suffix of $P, a$

$\delta(q, a) := k$

od

od

Boyer-Moore (BM)

Motivation

- KMP yields genuine benefits only if a mismatch as preceded by a partial match of some length
  - only in this case is the pattern slides more than 1 position
- Unfortunately, this is the exception rather than the rule
  - mismatches occur much more seldom than the KMP algorithm.
- The idea
  - start comparing characters at the end of the pattern rather than at the beginning
  - like in KMP, a pattern is pre-processed

Sequential Text Searching

Synopsis

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

- 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
    - $O(n \log m)$ running time (worst case)
    - $O(n \log m \log n)$ average time

Other string searching algorithms

- Rabin-Karp
- Shift-Or (it is sketched in Book)
  - ...
  - and many others...
  - ...
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
- Κ 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: 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 = “surgery”, P = “survey”, k=2
We will use a mxn matrix C
one row for each char of P, one column for each char of T
Searching Allowing Errors: Solution using Dynamic Programming (III)

T = “surgery”, P = “survey”, k=2

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

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

T

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

Searching Allowing Errors: Solution using Dynamic Programming (IV)

C[i,j] := C[i-1,j-1] if P[i]=T[j]
Else C[i,j] := 1 + min of:
• C[i-1,j]  // i-1 chars consumed P, j chars consumed of T
• C[i,j-1]  // i chars consumed P, j-1 chars consumed of T
• C[i-1,j]  // i chars consumed P, j-1 chars consumed of T

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>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</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>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>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>
Solution using Dynamic Programming: Example

- T = "surgery", P = "survey", k=2

<table>
<thead>
<tr>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<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>0</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>0</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>v</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>3</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>2</td>
<td>1</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.
- \(\text{Παρατήρηση: η πολυπλοκότητα είναι ανεξάρτητη του } k\)

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
Searching Allowing Errors: Solution with a Nondeterministic Automaton

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

- Horizontal arrows represent matching a document.
- Vertical arrows represent insertions into pattern.
- Solid diagonal arrows represent replacements.
- Dashed diagonal arrows represent deletion in the pattern (ε: empty).

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 $m$ is the size of the regular expression $\Pi \chi$, regex $= b^* (b \mid b^* a)$
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. $O(m^{2^m})$.

\[ b^* (b | b^* a) = (b b^* b | b b^* a) = (b b^* | b b^* a) \]

Pattern Matching Using Inverted Files

- Προηγουμένως είδαμε πως μπορούμε να αποτιμήσουμε επερωτήσεις με κριτήρια τύπου Edit Distance,RegExp, ομαδοποιώντας στα κείμενα.
- Τι κάνουμε αν έχουμε ήδη ένα Inverted File?
  - Ψάχνουμε το Λεξιλόγιο αντί των κειμένων (αρκετά μικρότερο σε μέγεθος)
  - Βρίσκουμε τις λέξεις που ταιριάζουν
  - Συγχωνεύουμε τις λίστες εμφάνισεων (occurrence lists) των λέξεων που ταίριαξαν.
- If block addressing is used, the search must be completed with a sequential search over the blocks.

Index terms

<table>
<thead>
<tr>
<th>Index terms</th>
<th>computer</th>
<th>database</th>
<th>...</th>
<th>science</th>
</tr>
</thead>
<tbody>
<tr>
<td>system</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Pattern Matching Using Suffix Trees

- Τι κάνουμε αν έχουμε ήδη ένα Suffix Tree?
  - Μπορούμε να αποτιμήσουμε τις επερωτήσεις εκεί, αντί στα κείμενα;

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.

"letter" < q < "many"
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.

\[ q = ma^* \]

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)

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