Πανεπιστήμιο Κρήτης, Τμήμα Επιστήμης Υπολογιστών
Άνοιξη 2007

ΗΥ463 - Συστήματα Ανάκτησης Πληροφοριών
Information Retrieval (IR) Systems

Ευρετηριασμός, Αποθήκευση και Οργάνωση Αρχείων Κειμένων (Indexing, Storage and File Organization) II

Γιάννης Τζίτζικας
Διάλεξη : 7
Ημερομηνία : 18-4-2007

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

b=3 (3 words per block) B=6 (bit masks of 6 bits)

	Block 1	Block 2	Block 3	Block 4
Text	This is a <u>text</u>.	A <u>text</u> has <u>many</u>	<u>words</u>. <u>Words</u> are	<u>made</u> from <u>letters</u>.
Text Signature	000101	110101	100100	101101

Signature Function

h(text)= 000101
h(many)= 110000
h(words)=100100
h(made)= 001100
h(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)

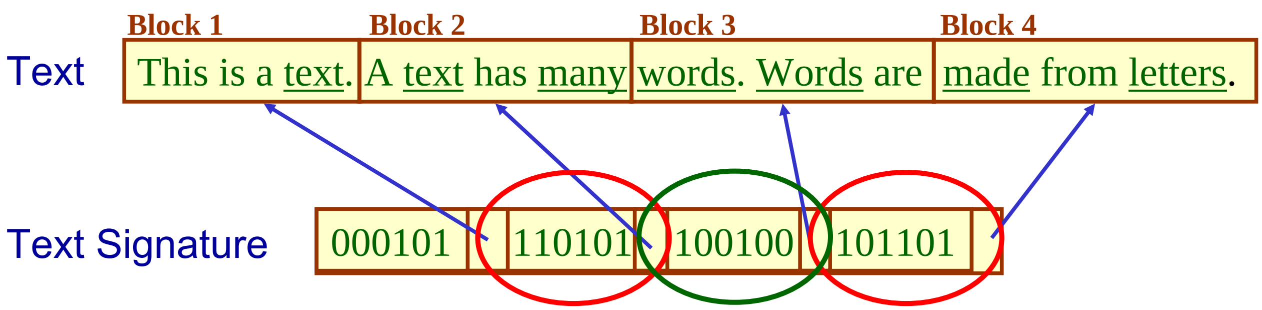
2/ Compare W with all bit masks Bi of all text blocks
If (W **&** Bi = W), the text block i is candidate (may contain the word w)

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=«words», h(«words»)=100100



Signature Function

h(text)= 000101
h(many)= 110000
h(words)=100100
h(made)= 001100
h(letters)=100001

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) to 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

Αρχεία Υπογραφών: Άλλες Παρατηρήσεις

- 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

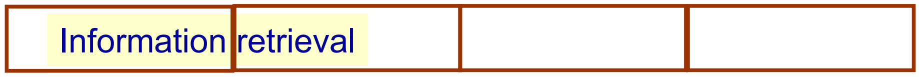
Signature files: Phrase and Proximity Queries

- 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

Phrase/Proximity Queries and **Block Boudaries**

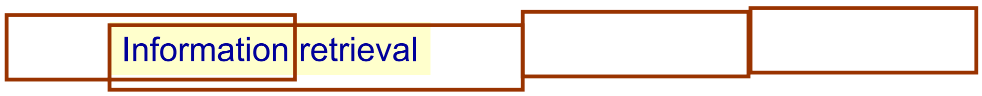
q=<information retrieval>

Text blocks

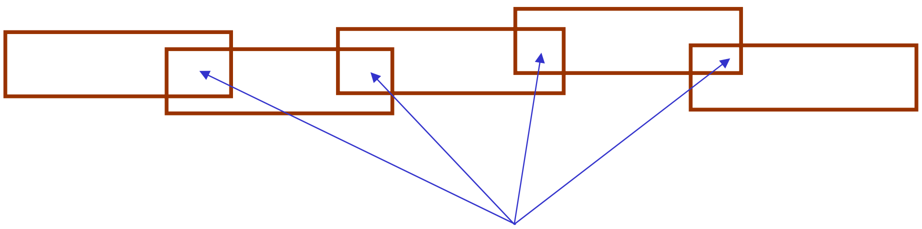


(πρόβλημα! Μπορούμε όμως να το λύσουμε με επικαλυπτόμενα blocks)

Overlapping blocks



For j-proximity queries



j-1 common words

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

Σειριακή Αναζήτηση Κειμένου: Το πρόβλημα

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)***.

Brute-Force Algorithm

```
Naive-String-Matcher(S,P)
n := length(S)
m := length(P)
for i = 0 to n-m do
        if P[1..m] = S[i+1 .. i+m] then
                return "Pattern occurs at position i"
        fi
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

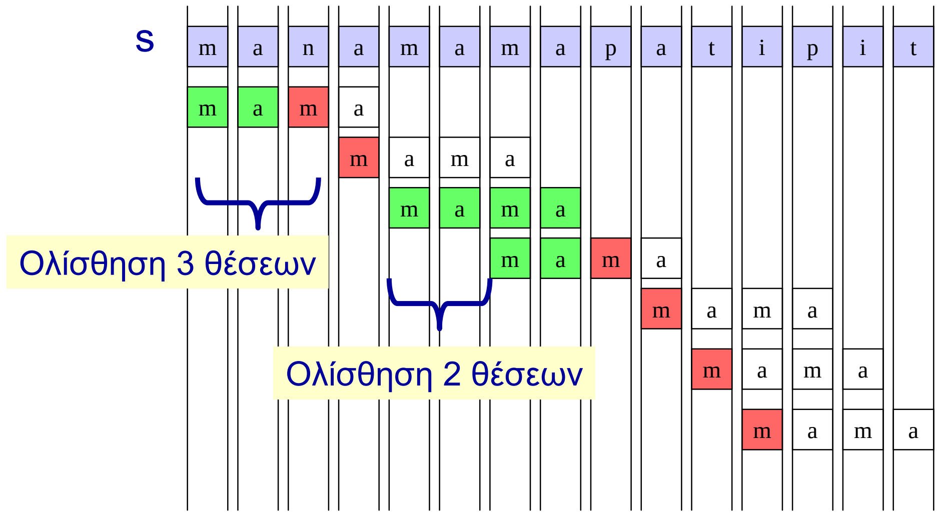
- Πιο γρήγοροι αλγόριθμοι που βασίζονται σε **μετακινούμενο (ολισθαίνον) παράθυρο**

- Γενική ιδέα:
  - 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.

- Οι αλγόριθμοι διαφέρουν στον τρόπο με τον οποίο ελέγχουν και ολισθαίνουν (μετακινούν) το παράθυρο.

Ολίσθηση Παραθύρου: Η γενική ιδέα

p=“mama”



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

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

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

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	$\boxed{0}$	0	0	1	0	1	0	0	0	0	4

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

- *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

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

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	

1

Example: match until 3rd char

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

s	a	b	a	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

2

Example: match until 7th char

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

s	a	b	r	a	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

5

Example: pattern matched

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	<u>7</u>

s	a	b	r	a	c	a	d	a	b	r	a	c							
p	a	b	r	a	c	a	d	a	b	r	a								
								a	b	r	a	c	a	d	a	b	r	a	

7

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

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,Σ,δ), 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 sates**
- Σ, is a finite **input alphabet**,
- $\delta: Q \times \Sigma \rightarrow Q$ is called the **transition function** of M

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

For the empty string ε we have: $\varphi(\varepsilon) := q_0$

For all $a \in \Sigma$, $w \in \Sigma^*$ define $\varphi(wa) := \delta(\varphi(w), a)$

M accepts w if$_{\text{and only i}}$f: $\varphi(w) \in A$

Example (I)

p=«abba»

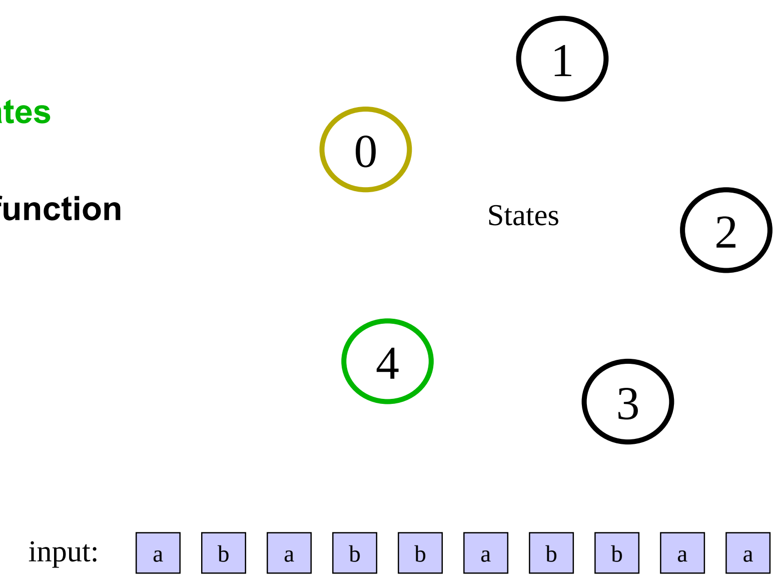
Q is a finite set of states

$q_0 \in Q$ is the start state

Q is a set of accepting sates

Σ: input alphabet

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



Example (II)

Q is a finite set of states

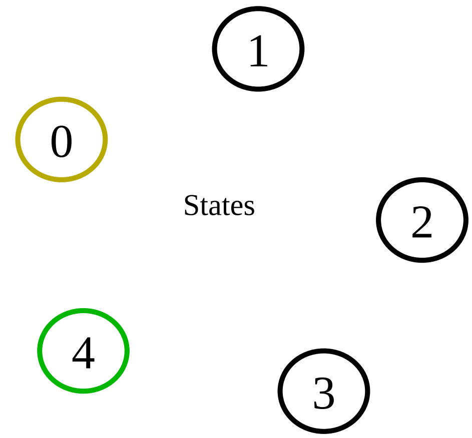
$q_0 \in Q$ is the start state

Q is a set of accepting states

Σ: input alphabet

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

p=«abba»



input / state	a	b
0	1	0
1	1	2
2	1	3
3	4	0
4	1	2

Example (III)

Q is a finite set of states

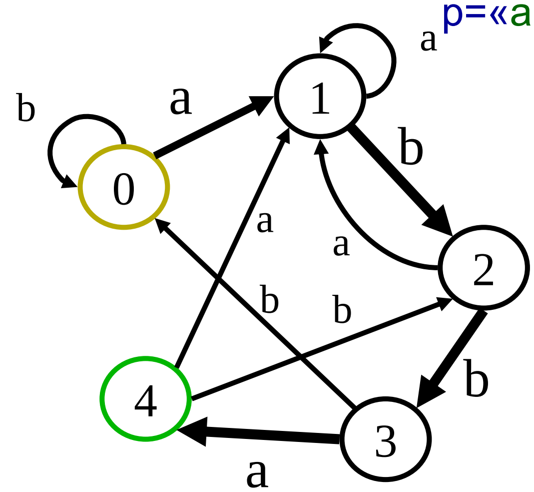
$q_0 \in Q$ is the start state

Q is a set of accepting sates

Σ: input alphabet

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

p=«abba»



input / state	a	b
0	1	0
1	1	2
2	1	3
3	4	0
4	1	2

Example (IV)

Q is a finite set of states

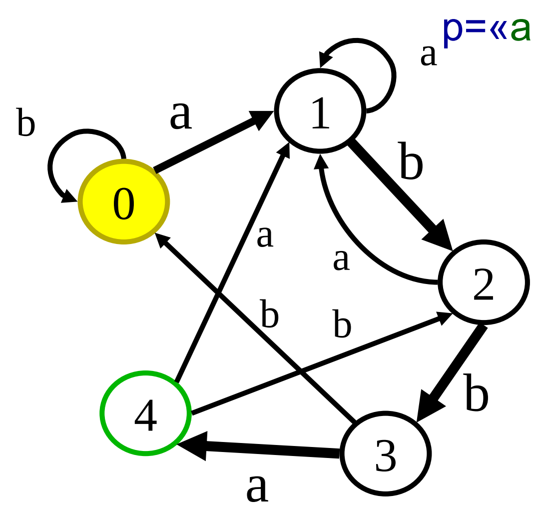
$q_0 \in Q$ is the start state

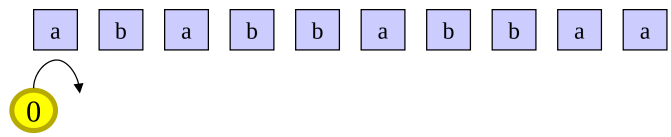
Q is a set of accepting sates

Σ: input alphabet

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

state \ input	a	b
0	1	0
1	1	2
2	1	3
3	4	0
4	1	2





Example (V)

Q is a finite set of states

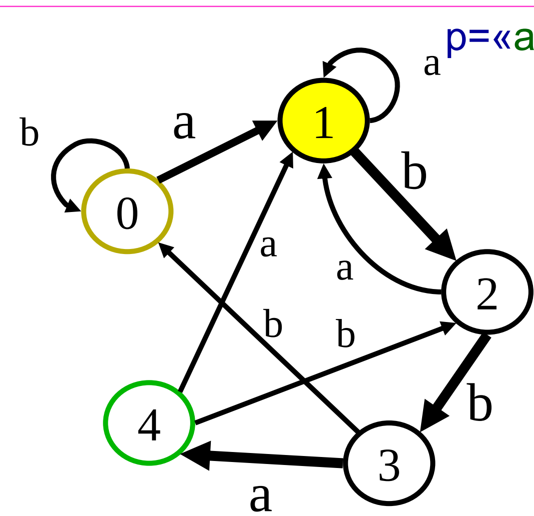
$q_0 \in Q$ is the start state

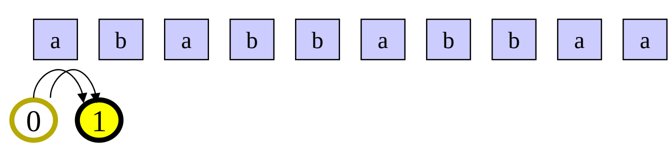
Q is a set of accepting sates

Σ: input alphabet

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

state \ input	a	b
0	1	0
1	1	2
2	1	3
3	4	0
4	1	2





Example (VI)

Q is a finite set of states

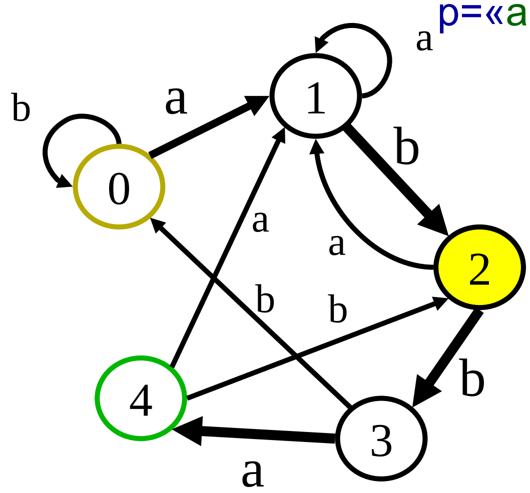
$q_0 \in Q$ is the start state

Q is a set of accepting sates

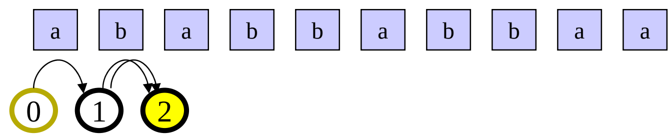
Σ: input alphabet

δ: Q × Σ → Q: transition function

p=«abba»



input	a	b
state 0	1	0
1	1	2
2	1	3
3	4	0
4	1	2



Example (VII)

Q is a finite set of states

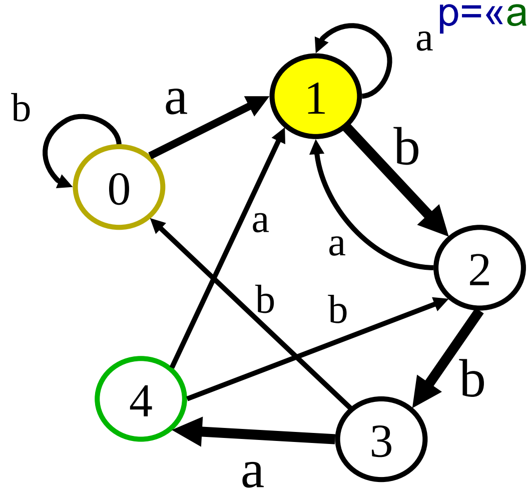
$q_0 \in Q$ is the start state

Q is a set of accepting sates

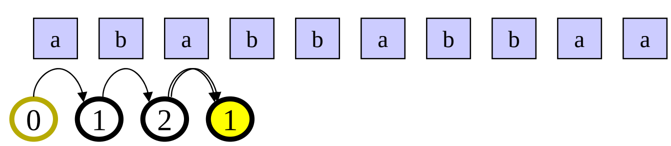
Σ: input alphabet

δ: Q × Σ → Q: transition function

p=«abba»



input	a	b
state 0	1	0
1	1	2
2	1	3
3	4	0
4	1	2



Example (VIII)

Q is a finite set of states

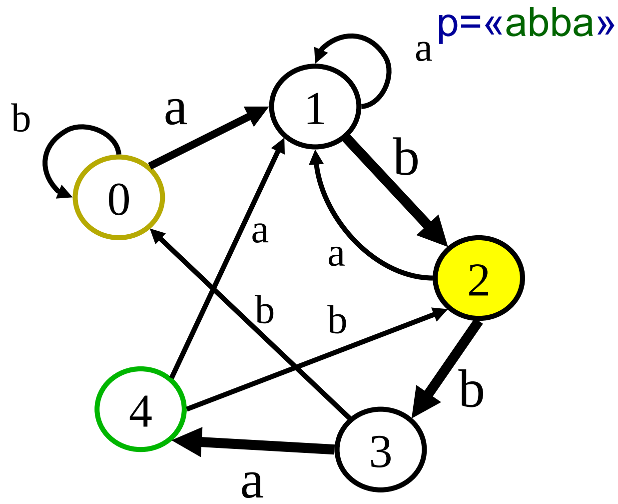
$q_0 \in Q$ is the start state

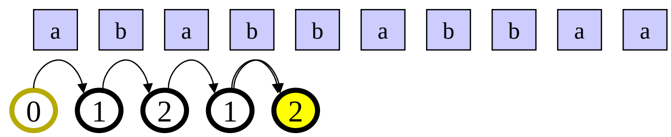
Q is a set of accepting sates

Σ: input alphabet

δ: Q × Σ → Q: transition function

input \ state	a	b
0	1	0
1	1	2
2	1	3
3	4	0
4	1	2





Example (IX)

Q is a finite set of states

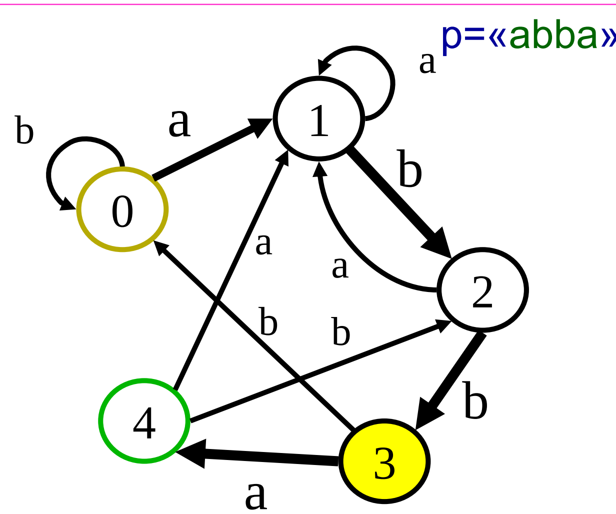
$q_0 \in Q$ is the start state

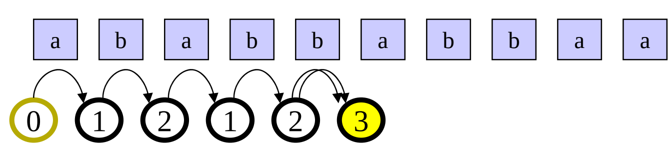
Q is a set of accepting sates

Σ: input alphabet

δ: Q × Σ → Q: transition function

input \ state	a	b
0	1	0
1	1	2
2	1	3
3	4	0
4	1	2





Example (X)

Q is a finite set of states

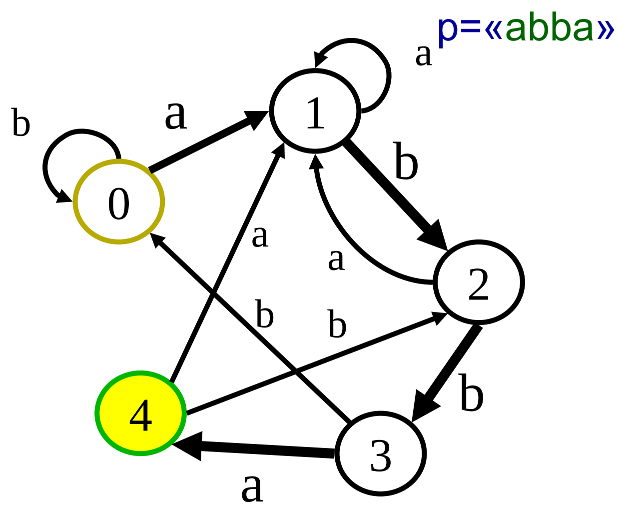
$q_0 \in Q$ is the start state

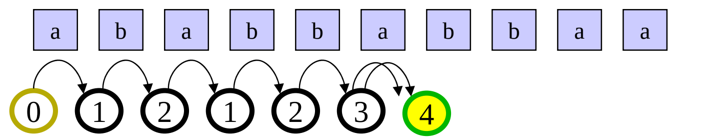
Q is a set of accepting sates

Σ: input alphabet

δ: Q × Σ → Q: transition function

input \ state	a	b
state 0	1	0
1	1	2
2	1	3
3	4	0
4	1	2





Example (XI)

Q is a finite set of states

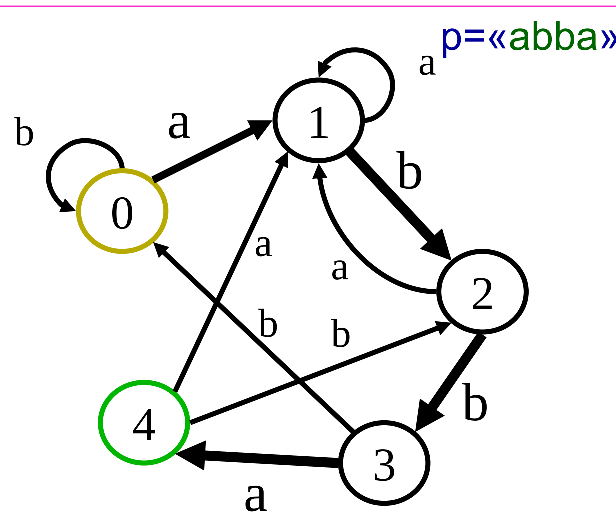
$q_0 \in Q$ is the start state

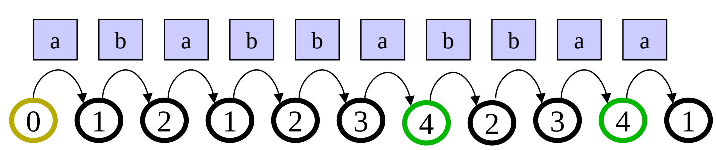
Q is a set of accepting sates

Σ: input alphabet

δ: Q × Σ → Q: transition function

input \ state	a	b
state 0	1	0
1	1	2
2	1	3
3	4	0
4	1	2



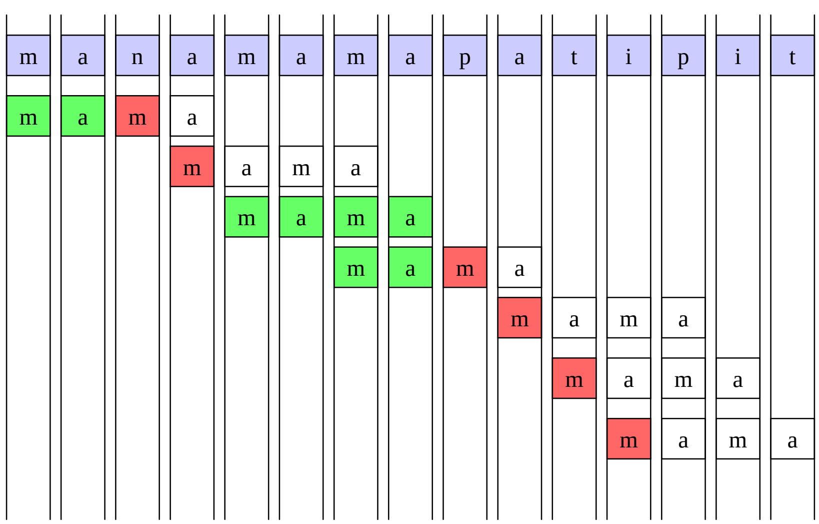


Finite-Automaton-Matcher

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, Σ)

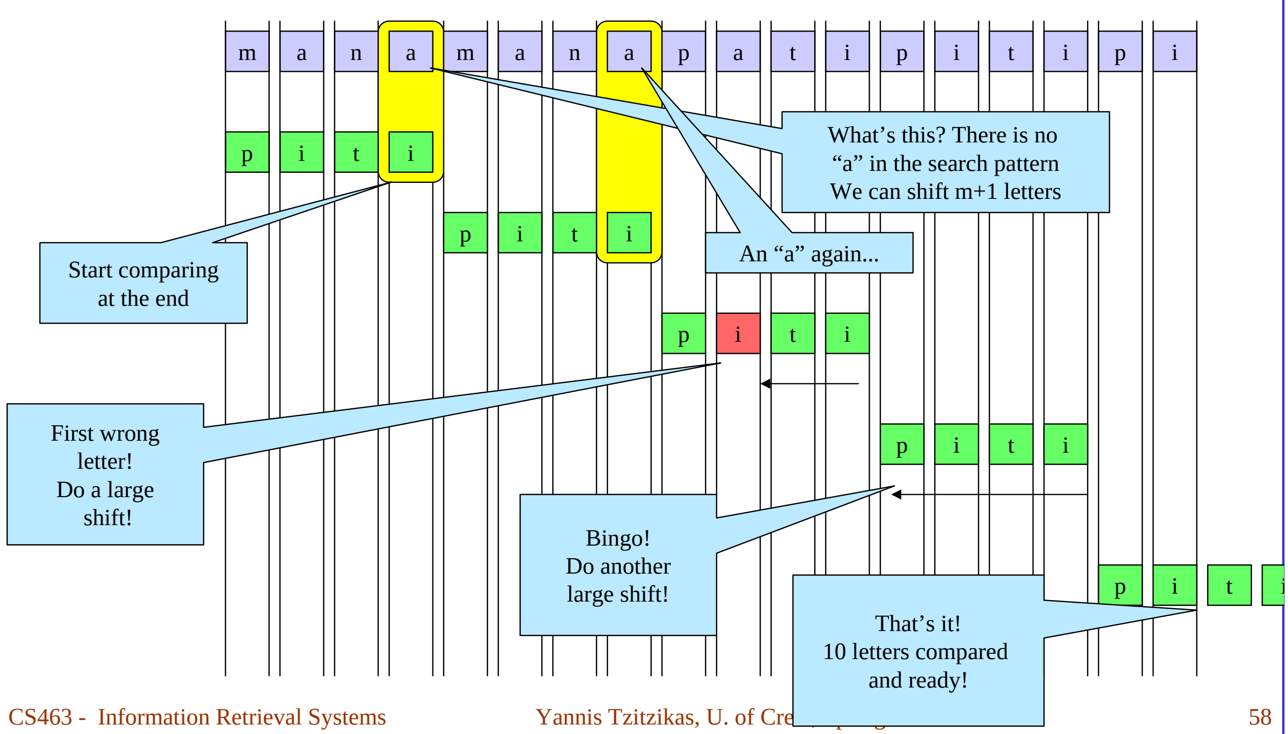
```
m := length(P)
for q = 0 to m do
    for each character a ∈ Σ do
      k := 1+min(m,q+1)
      repeat
          k := k-1
      until P_k is a suffix of P_q a
      δ(q,a) := k
    od
od
```


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 is the pattern slides more than 1 position
  - Unfortunately, this is the exception rather than the rule
    - mathes 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



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(nm) running time (worst case)
  - O(n log(m)/m) 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**
  - **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 = “surgery”, P = “survey”, k=**2**

We will use a m x n matrix C

one row for each char of P, one column for each char of T

T

P

		s	u	r	g	e	r	y
s								
u								
r								
v								
e								
y								

Searching Allowing Errors: Solution using **Dynamic Programming (III)**

T = “surgery”, P = “survey”, k=**2**

οι γραμμές του C εκφράζουν πόσα γράμματα του pattern έχουμε ήδη καταναλώσει (στην 0 γραμμή τίποτα, στην 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)

T

P

		s	u	r	g	e	r	y
	0	0	0	0	0	0	0	0
s	1							
u	2							
r	3							
v	4							
e	5							
y	6							

Searching Allowing Errors: Solution using **Dynamic Programming (IV)**

C[i,j] :=   C[i-1,j-1],   **if** P[i]=T[j]

// εγινε match άρα τα “λάθη” ήταν όσα και πριν

Else C[i,j] := **1** + *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**

T

P

		s	u	r	g	e	r	y
	0	0	0	0	0	0	0	0
s	1	0	1	1	1	1	1	1
u	2	1	0	1	2	2	2	2
r	3	2	1	0	1	2	2	3
v	4	3	2	1	1	2	3	3
e	5	4	3	2	2	1	2	3
y	6	5	4	3	3	**2**	**2**	**2**

Solution using **Dynamic Programming: Example**

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

T

P

		s	u	r	g	e	r	y
	0	0	0	0	0	0	0	0
s	1	0	1	1	1	1	1	1
u	2	1	0	1	2	2	2	2
r	3	2	1	0	1	2	2	3
v	4	3	2	1	1	2	3	3
e	5	4	3	2	2	1	2	3
y	6	5	4	3	3	**2**	**2**	**2**

Solution using **Dynamic Programming: Example**

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

T

P

		s	u	r	g	e	r	y
	0	0	0	0	0	0	0	0
s	1	0	1	1	1	1	1	1
u	2	1	0	1	2	2	2	2
r	3	2	1	0	1	2	2	3
v	4	3	2	1	1	2	3	3
e	5	4	3	2	2	1	2	3
y	6	5	4	3	3	**2**	**2**	**2**

Solution using **Dynamic Programming: Example**

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

T

P

		s	u	r	g	e	r	y
	0	0	0	0	0	0	0	0
s	1	0	1	1	1	1	1	1
u	2	1	0	1	2	2	2	2
r	3	2	1	0	1	2	2	3
v	4	3	2	1	1	2	3	3
e	5	4	3	2	2	1	2	3
y	6	5	4	3	3	**2**	**2**	**2**

Solution using **Dynamic Programming: Example**

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

T

P

		s	u	r	g	e	r	y
	0	0	0	0	0	0	0	0
s	1	0	1	1	1	1	1	1
u	2	1	0	1	2	2	2	2
r	3	2	1	0	1	2	2	3
v	4	3	2	1	1	2	3	3
e	5	4	3	2	2	1	2	3
y	6	5	4	3	3	2	2	2

1 + $\boxed{}$

Solution using **Dynamic Programming: Example**

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

T

P

		s	u	r	g	e	r	y
	0	0	0	0	0	0	0	0
s	1	0	1	1	1	1	1	1
u	2	1	0	1	2	2	2	2
r	3	2	1	0	1	2	2	3
v	4	3	2	1	1	2	3	3
e	5	4	3	2	2	1	2	3
y	6	5	4	3	3	2	2	2

1 + $\boxed{}$

Solution using **Dynamic Programming: Example**

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

T

P

		s	u	r	g	e	r	y
	0	0	0	0	0	0	0	0
s	1	0	1	1	1	1	1	1
u	2	1	0	1	2	2	2	2
r	3	2	1	0	1	2	2	3
v	4	3	2	1	1	2	3	3
e	5	4	3	2	2	1	2	3
y	6	5	4	3	3	2	2	2

1 + $\boxed{}$

Solution using **Dynamic Programming: Example**

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

T

P

		s	u	r	g	e	r	y
	0	0	0	0	0	0	0	0
s	1	0	1	1	1	1	1	1
u	2	1	0	1	2	2	2	2
r	3	2	1	0	1	2	2	3
v	4	3	2	1	1	2	3	3
e	5	4	3	2	2	1	2	3
y	6	5	4	3	3	2	2	2

Solution using **Dynamic Programming: Example**

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

T

P

		s	u	r	g	e	r	y
	0	0	0	0	0	0	0	0
s	1	0	1	1	1	1	1	1
u	2	1	0	1	2	2	2	2
r	3	2	1	0	1	2	2	3
v	4	3	2	1	1	2	3	3
e	5	4	3	2	2	1	2	3
y	6	5	4	3	3	**2**	**2**	**2**

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 κ

Solution using **Dynamic Programming: Example**

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

T

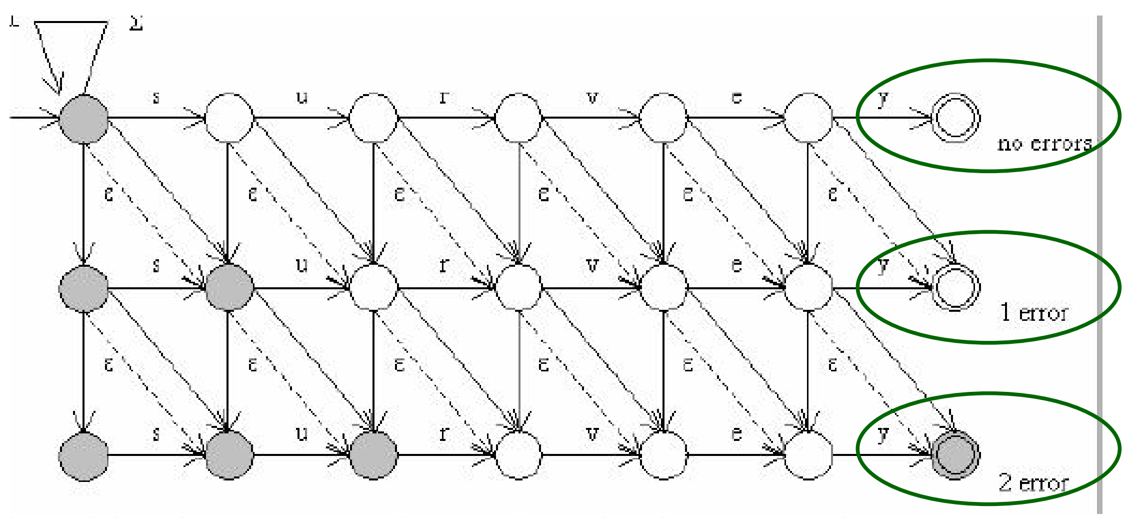
P

		s	u	r	g	e	r	y
	0	0	0	0	0	0	0	0
s	1	0	1	1	1	1	1	1
u	2	1	0	1	2	2	2	2
r	3	2	1	0	1	2	2	3
v	4	3	2	1	1	2	3	3
e	5	4	3	2	2	1	2	3
y	6	5	4	3	3	**2**	**2**	**2**

- 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

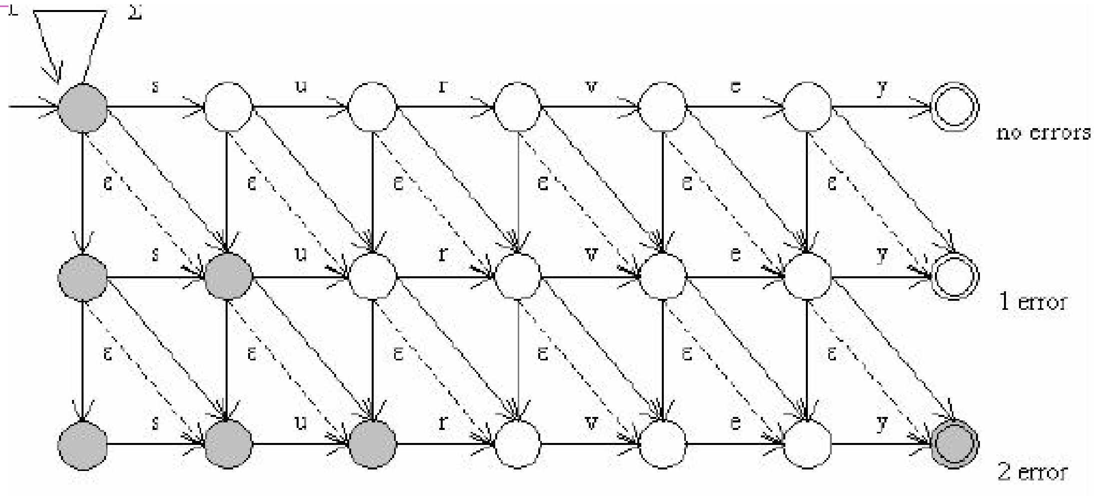
Searching Allowing Errors: Solution with a Nondeterministic Automaton



- 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 represents 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 (ε: empty).

Searching Allowing Errors: Solution with a Nondeterministic Automaton

- 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

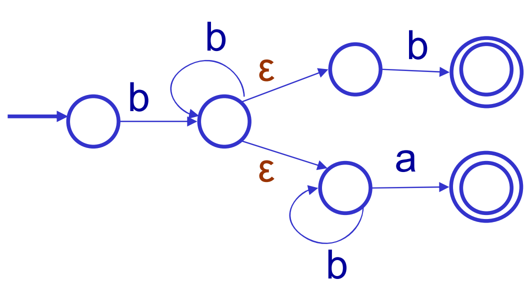
Searching using Regular Expressions

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

Π.χ. regex = b b* (b | b* a)

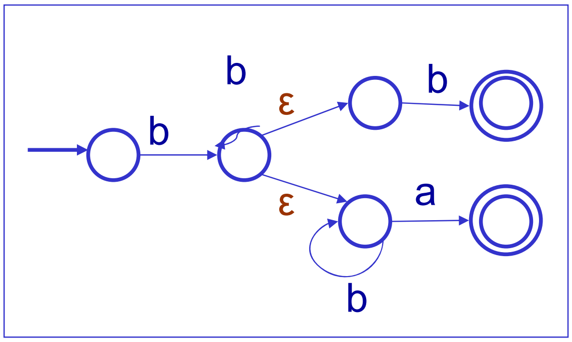


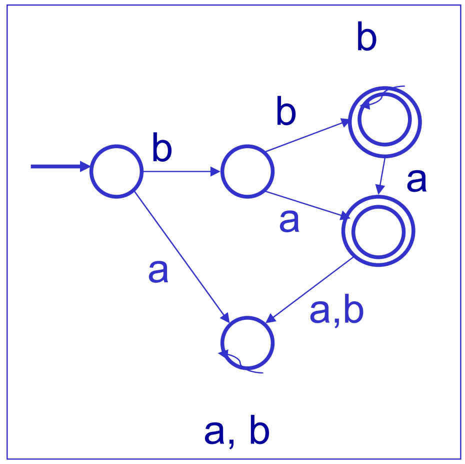
Searching using Regular Expressions (II)

(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)



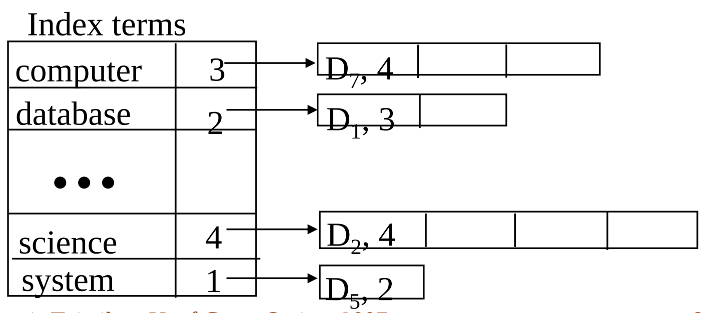


Bit-Parallelism to avoid constructing the deterministic automaton (NFA Simulation)

Pattern Marching Queries and Index Structures

Pattern Matching Using **Inverted Files**

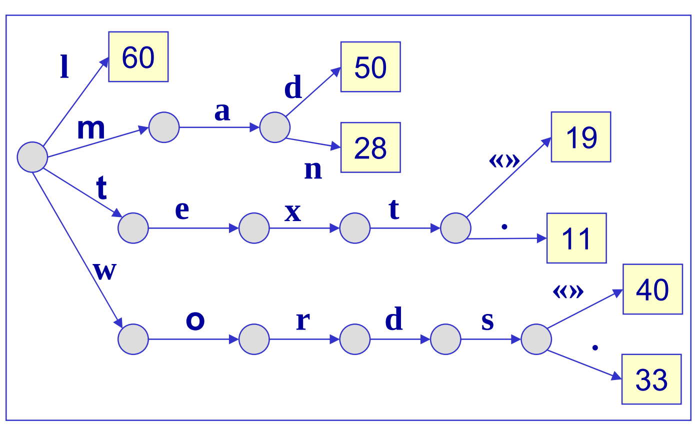
- Προηγουμένως είδαμε πως μπορούμε να αποτιμήσουμε επερωτήσεις με κριτήρια τύπου Edit Distance, RegExpr, ανατρέχοντας στα κείμενα.
- Τι κάνουμε αν έχουμε ήδη ένα Inverted File ?
  - Ψάχνουμε το Λεξιλόγιο αντί των κειμένων (αρκετά μικρότερο σε μέγεθος)
  - Βρίσκουμε τις λέξεις που ταιριάζουν
  - Συγχωνεύουμε τις λίστες εμφανίσεων (occurrence lists) των λέξεων που ταίριαξαν.
- 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.



Pattern Matching Using **Suffix Trees**

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

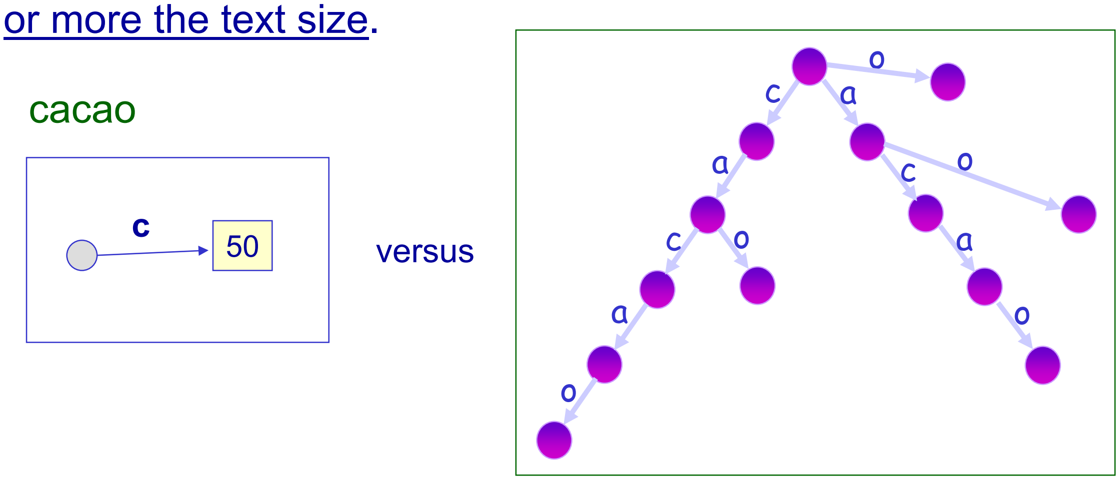
Suffix Trie



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 <u>same search algorithm and cost</u> described for word search.

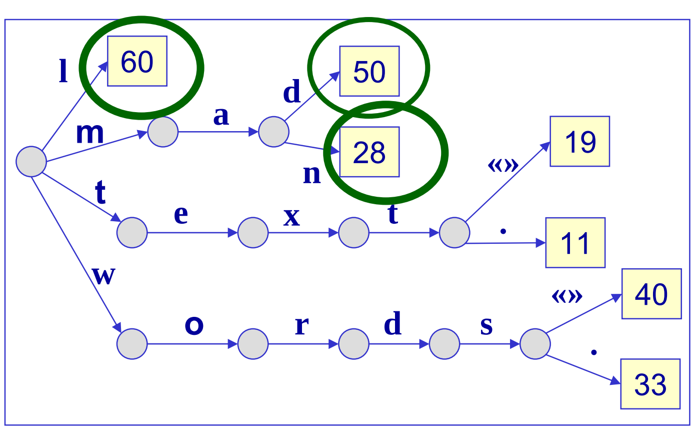
Indexing all text positions normally makes the suffix array size <u>10 times or more the text size</u>.



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

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

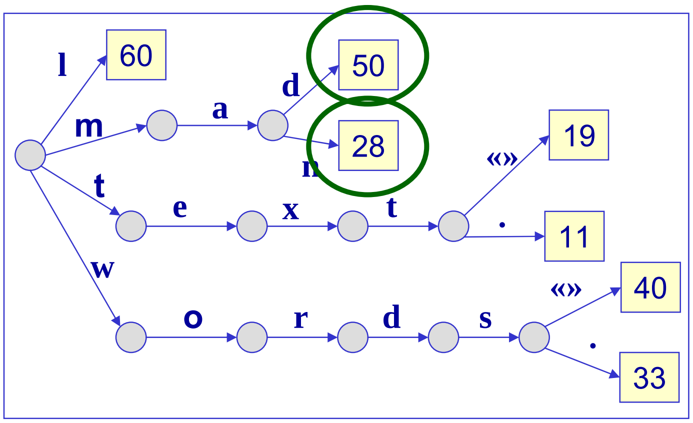
"letter" < q < "many"



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)
- A recent software package that implements several string matching algorithm
  - http://flamingo.ics.uci.edu/releases/1.0/