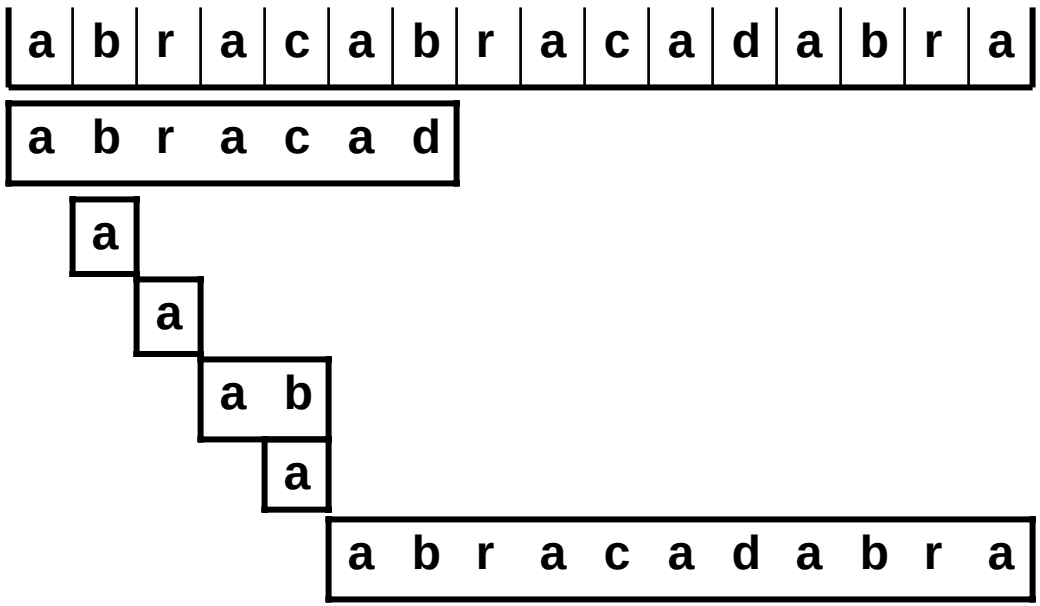
Text Searching

- Thus far, we've seen how to pre-process texts (stemming, "stopping" words, compressing, etc)
- We've also seen how to build, process, improve and assess the quality of queries and the returned answer sets
- We also saw how to effectively index the texts
- Now, how do we search text ? It goes beyond searching the indices, specially in the cases where the indices are not enough to answer queries (e.g., proximity queries)

Sequential Searching

- If no index is used this is the only option, however sometimes (e.g., when blocking is used) it is necessary even if an index exists
- The problem is “simple”
  - Given a pattern x of length m and a text y of length n (n >> m) find all positions in y where x occurs
- There is much more work on this than we can cover, including many theoretical results. Thus we will discover some well known approaches

Brute Force



BRUTE_FORCE_MATCHER(x,m,y,n)

1. **For p from 0 up to n-m**
2. **loop if y[p..p+m-1]=x**
3. **then report p**

- The name says it all …
- Starting from all possible initial positions (i.e., all positions), check whether the pattern could start at that position
- It takes O(mn) time in the worst case:
  - For instance when $\alpha^{m-1}b$ is searched in α^n for any two symbols α,b.
- O(n) in the average case – not that bad
- The most important thing is that it suggests the use of a sliding window over the text. The idea is to see whether we can see the pattern through the window

The Karp-Rabin Algorithm

- Hashing: a simple method for avoiding a quadratic number of symbol comparisons in most practical situations.
- At each position p of the window on the text, check if the part of the text delimited by the window y[p . . p + m – 1] "looks like" x.
- Use a hash function to check the resemblance
- The hash function should be highly discriminating for strings.
- The function should also have the following properties:
  - To be efficiently computable;
  - Easy computation of the value associated with the next part of the text from the value associated with the current part.
- If Σ is the alphabet of input symbols, the hash function h:

$$h(u) = \left(\sum_{i=0}^{|u|-1} u[i] \times d^{|u|-1-i} \right) \bmod q$$

where q and d are two constants. Then for each $v \in \sum^*$ for each symbols $a', a'' \in \sum$, $h(va'')$ is computed from $h(a'v)$ by the formula

$$h(va'') = \left(\left(h(a'v) - a' \times d^{|v|} \right) \times d + a'' \right) \bmod q$$

- Search for pattern x: compare the value h(x) with the hash value associated each part of length m of text y.
- If the two values are equal: necessary to check whether this part of the text is equal to x or not by symbol comparisons

The Karp-Rabin algorithm (cont.)

```
KARP-RABIN-MATCHER(x, m, y, n)
1  r ← 1
2  s ← x[0] mod q
3  t ← 0
4  for i from 1 up to m − 1
5     loop r ← (r × d) mod q
6          s ← (s × d + x[i]) mod q
7          t ← (t × d + y[i − 1]) mod q
8  for p from 0 up to n − m
9     loop t ← (t × d + y[p + m − 1]) mod q
10         if t = s and y[p .. p + m − 1] = x
11            then report p
12         t ← ((c − 1) × q + t − y[p] × r) mod q
```

- The value of symbols ranges from 0 to c-1
- The quantity (c-1) x q is added in line 8 to provide correct computations on positive integers
- Convenient values for *d* are powers of 2:
  - in this case, all the products by *d* can be computed as shifts on integers.
- The value of *q* is generally a large prime (such that the quantities *(q* - 1) x *d* + *c* - 1 and *c* x *q* - 1 do not cause overflows),

p	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
$y[p]$	n	o	␣	d	e	f	e	n	s	e	␣	f	o	r	␣	s	e	n	s	e
$h(y[p..p+4])$	8	8	6	28	9	18	28	26	22	12	17	24	16	0	1	9	—	—	—	—

- Searching for the pattern x = sense in the text y = “no defense for sense”:
  - *c* = 256 (ASCII code), and the values of
  - *q* and *d* are set respectively to 31 and 2.
  - *h(x)* = (115 x 16 + 101 x 8 + 110 x 4 + 115 x 2 + 101) mod 31 = 9.
  - *h(y[4.. 8])=h(y[l5.. 19])= h(x):* two string-to-string comparisons against x are performed.

The Knuth-Morris-Pratt Algorithm

- The first algorithm with linear worst-case behaviour,
- On average it is not much faster than Brute-Force (BF) algorithm.
- Sliding window over the text: however, it does not try all window positions as BF does.
- It reuses information from previous checks.
- After the window is checked, a number of pattern letters were compared to the text window, and they all matched except possibly the last one compared.
- Hence, there is a *prefix* of the pattern that matched the text.
- The algorithm takes advantage of this information to avoid trying window positions which can be deduced not to match.
- The pattern is preprocessed in *O(m)* time and space to compute a function called *ψ.*
- *ψ*(i): the length of the longest proper prefix of $x_{0..i-1}$ which is also a suffix and the characters following prefix and suffix are different.

$$\psi[i] = \max\{k \mid (0 \le k < i, x[i-k..i-1] = x[0..k-1], x[k] \ne x[i]) \; or \; (k = -1)\}$$

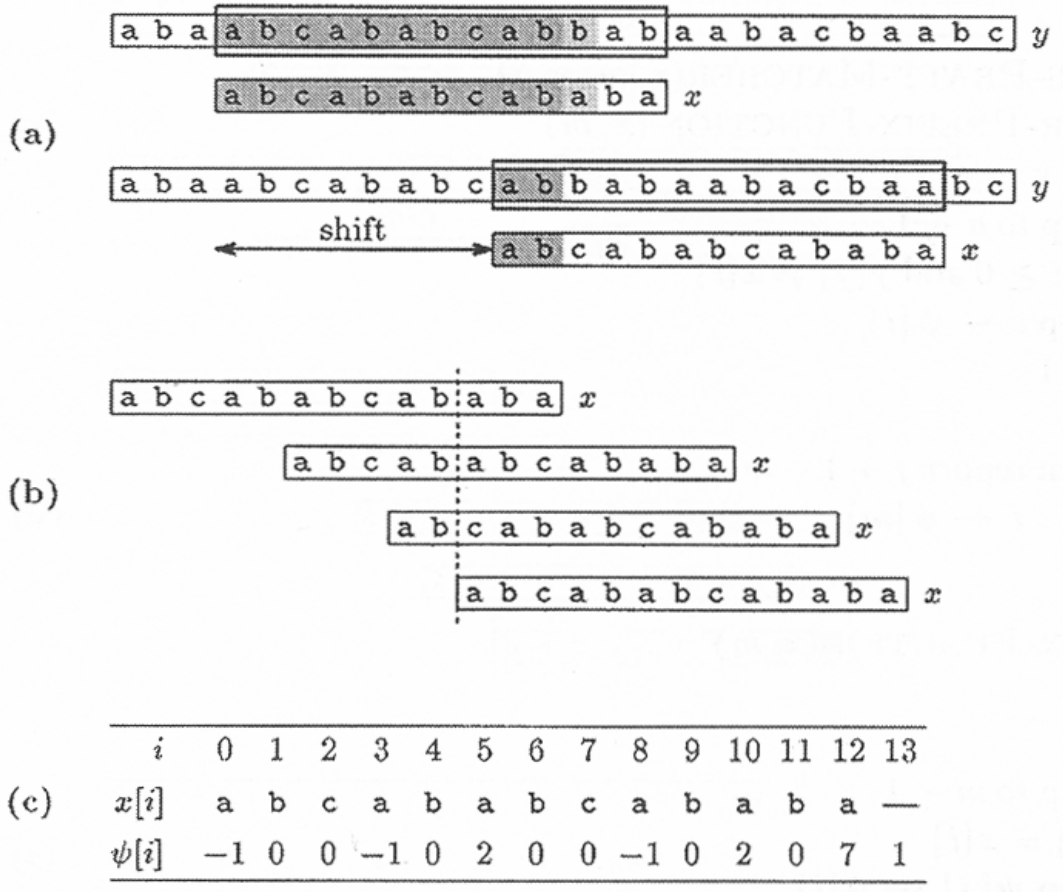
$$\text{where } i \in \{0,1,\ldots m-1\} \text{ and}$$

$$\psi[m] = \max\{k \mid 0 \le k < m, x[m-k..m-1] = x[0..k-1]\}$$

- Hence i - *ψ[i]* window positions can be safely skipped if the characters up to i - 1 matched, and the i-th did not:
  - Crucial observation: this shift depend only on the pattern, because if the text in the window matched up to position i - 1, then that text is equal to the pattern

The Knuth-Morris-Pratt Algorithm (Cont.)

```
KNUTH-MORRIS-PRATT-MATCHER(x, m, y, n)
1  ψ ← BETTER-PREFIX-FUNCTION (x, m)
2  i ← 0
3  for j from 0 up to n − 1
4     loop while i ≥ 0 and y[j] ≠ x[i]
5           loop i ← ψ[i]
6        i ← i + 1
7        if i = m
8           then report j + 1 − m
9                i ← ψ[m]
```



i	0	1	2	3	4	5	6	7	8	9	10	11	12	13
$x[i]$	a	b	c	a	b	a	b	c	a	b	a	b	a	—
$\psi[i]$	−1	0	0	−1	0	2	0	0	−1	0	2	0	7	1

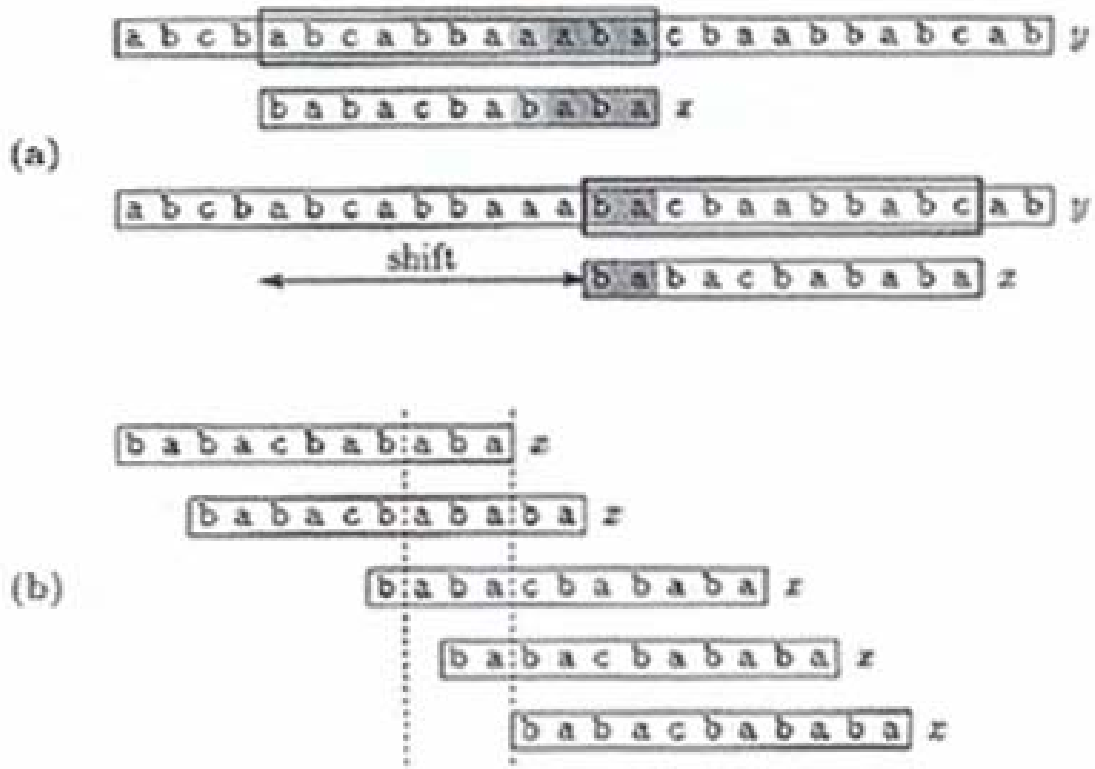
- Searching for the pattern *x* = abcababcababa.
- (a) The window on the text *y* at position 3. A mismatch occurs at position 10 on *x.* The matching symbols are shown darkly shaded, and the current analyzed symbols lightly shaded. We should shift the window 8 positions to the right. The string-to-string comparison resumes at position 2 on the pattern.
- (b) The prefixes of *x* that are suffixes of *x* [0.. 9] = abcabacbab are right-aligned along the discontinuous vertical line. String *x[0..* 4] = abcab is a suffix of *x* [0 . . 9], but is followed by symbol a which is identical to *x* [10]. String *x*[0..1] is the expected prefix, since it is followed by symbol c.
- (c) The values of the function *ψ* for pattern *x.*

The Boyer-Moore Algorithm

- The most efficient string-matching algorithm in usual applications.
- A simplified version of it, or the entire algorithm, is often implemented in text editors for the "search" and "substitute" commands.
- The scan operation proceeds from right to left in the window on the text, instead of left to right as in the Knuth-Morris-Pratt algorithm.
- In case of a mismatch, the algorithm uses two functions to shift the window:
  - good-suffix shift function
  - bad-symbol shift function.
- Good-suffix shift function:
  - Let p be the current (left) position of the window on the text. Assume that a mismatch occurs between symbols y[p + i] and x[i] for some i, 0 ≤ i ≤ m-1:
    - y[p + i] ≠ x[i] and y[p + i + 1.. p + m - 1] = x [i + 1 . . m - 1].
  - Good-suffix shift: align y [p + i + 1 . . p + m - 1] with its rightmost occurrence x [k + 1 . . m - 1 - i + k] in x preceded by a symbol x[k] different from x[i] to avoid an immediate mismatch.
  - If no such occurrence exists, align the longest suffix of y[p + i + 1 … p + m - 1] with a matching prefix of x. The good-suffix shift function β is defined by

$$\beta[i] = \min\{i - k \mid (0 \le k < i,\ x[k+1 \ldots m-1-i+k] = x[i+1 \ldots m-1],\ x[k] \ne x[i])$$
$$or\ (i-m < k < 0,\ x = x[i-k \ldots m-1]x[m-i+k \ldots m-1])\ or\ (k = i-m)\}$$

The Boyer-Moore Algorithm



i	0	1	2	3	4	5	6	7	8	9	10
$x[i]$	b	a	b	a	c	b	a	b	a	b	a
$\beta[i]$	7	7	7	7	7	7	2	9	4	11	1

- The good-suffix shift in the Boyer-Moore algorithm when searching for the pattern x = babacbababa.
- (a) The window on the text is at position 4. The string-to-string comparison, which proceeds from right to left, stops with a mismatch at position 7 on x.
- The window is shifted 9 positions to the right to avoid an immediate mismatch.
- (b) Indeed, the string x [8 . . 10] = aba is repeated three times in x, but is preceded each time by symbol x [7] = b. The expected matching substring in x is then the prefix ba of x. The substrings of x identical with aba and the prefixes of x ending with a suffix of aba are right-aligned along the rightmost discontinuous vertical line.
- (c) The values of the shift function β for pattern x.

The Boyer-Moore Algorithm (cont.)

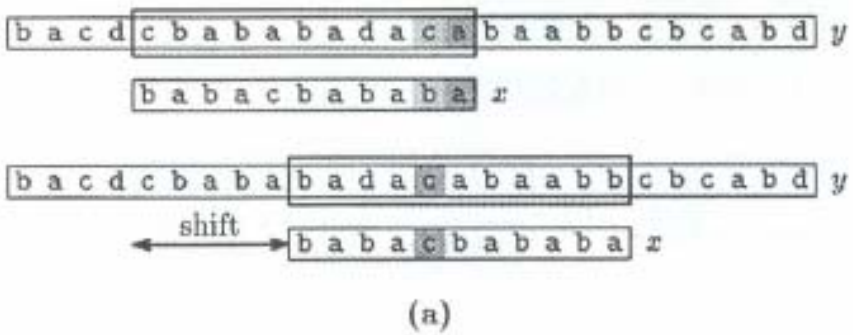
- Bad-symbol shift function:
  - Text symbol y[p +i] that causes a mismatch.
  - Assume that this symbol occurs in x [0…i- 1].
  - let k be the position of the rightmost occurrence of y[p + i] in x[0.. i - 1].
  - The window can be shifted i - k positions to the right without missing an occurrence of x in y. Assume now that
  - If symbol y[p + i] does not occur in x[0…i-1], no occurrence of x in y can overlap the position p + i on the text, and thus, the window can be shifted i + 1 positions to the right.
  - Let δ be the table indexed on alphabet Σ and {0,..,m-1}, and defined for each symbol α Σ and i by

$$\delta[a,i] = \min\{m\} \cup \{m-1-j \mid 0 \le j < i, x[j] = a\}$$

Searching for the pattern *x* = babacbababa.

a) The window on the text at position 4.
Mismatch at position 9 on *x.*
y[13] = c -> shift the window 5 positions to the right.
Notice that if the unexpected symbol were a or d, the applied shift would have been 1 and 10 respectively.

b) The values of the table *δ* for pattern *X* when alphabet Σ is reduced to { a, b, c, d}.



(a)

a	a	b	c	d
δ[α,10]	2	3	6	11

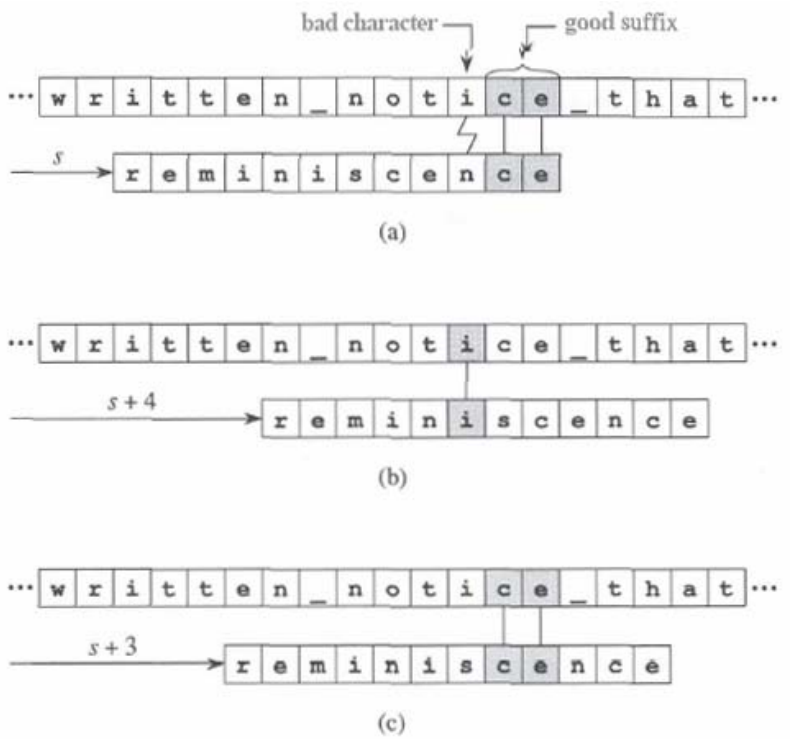
(b)

The Boyer-Moore Algorithm (cont.)

```
BOYER-MOORE-MATCHER(x, m, y, n)
1  β ← GOOD_SUFFIX_FUNCTION(x, m)
2  δ ← BAD_SYMBOL_FUNCTION(x, m)
3  p ← 0
4  while p ≤ n − m
5     loop i ← m − 1
6          while i ≥ 0 and y[p + i] = x[i]
7             loop i ← i − 1
8          if i ≥ 0
9             then p ← p + max{β[i], δ[y[p + i],i] + i − m + 1}
10            else report p
11                 p ← p+β[0]
```

- Pattern preprocessing time and space: $O(m+|\Sigma|)$ where $|\Sigma|$ is the size of the alphabet Σ
- $O(nm + |\Sigma|)$ worst-case run time, $O((n \log m)/m)$ on average
- The worst case is unlikely in English text
- Boyer-Moore's algorithm is significantly faster than the brute-force algorithm on English text

Another example of the Boyer-Moore heuristics.



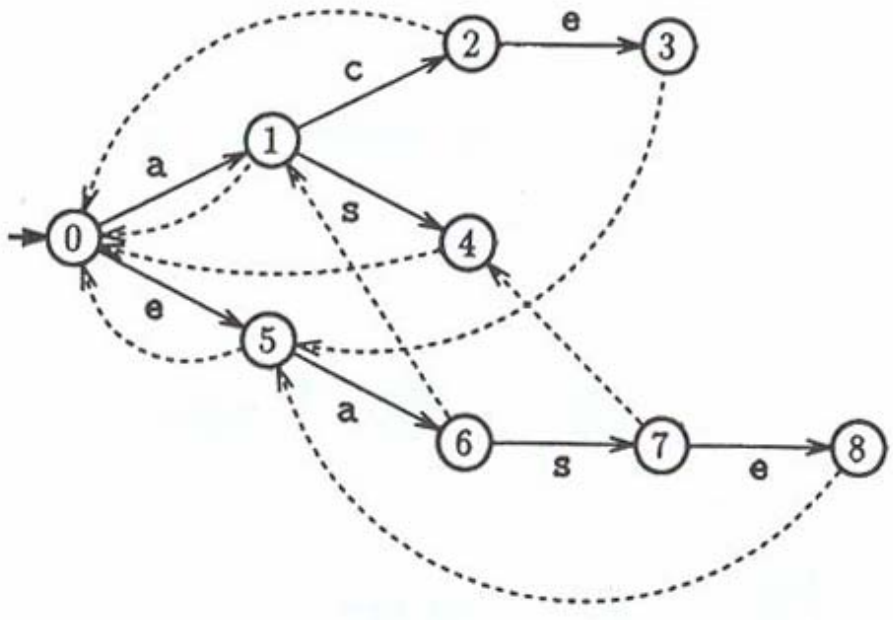
(a) Matching the pattern “reminiscence” against a text by comparing characters in a right-to-left manner. The shift s is invalid. Although a "good suffix" “ce” of the pattern matched correctly against the corresponding characters in the text (matching characters are shown shaded), the "bad character" “i”, which didn't match the corresponding character “n” in the pattern, was discovered in the text.

(b) The bad-symbol heuristic proposes moving the pattern to the right, if possible, by the amount that guarantees that the bad text character will match the rightmost occurrence of the bad character in the pattern. In this example, moving the pattern 4 positions to the right causes the bad text character “i” in the text Io match the rightmost “i” in the pattern, at position 6. If the bad character doesn't occur in the pattern, then the pattern may be moved completely past the bad character in the text.

(c) With the good suffix heuristic, the pattern is moved to the right by the least amount that guarantees that any pattern characters that align with the good suffix “ce” previously found in the text will match those suffix characters. In this example, moving the pattern 3 positions to the right satisfies this condition. Since the good suffix heuristic proposes a movement of 3 positions, which is smaller than the 4-position proposal of the bad-character heuristic, the Boyer-Moore algorithm increases the shift by 4.

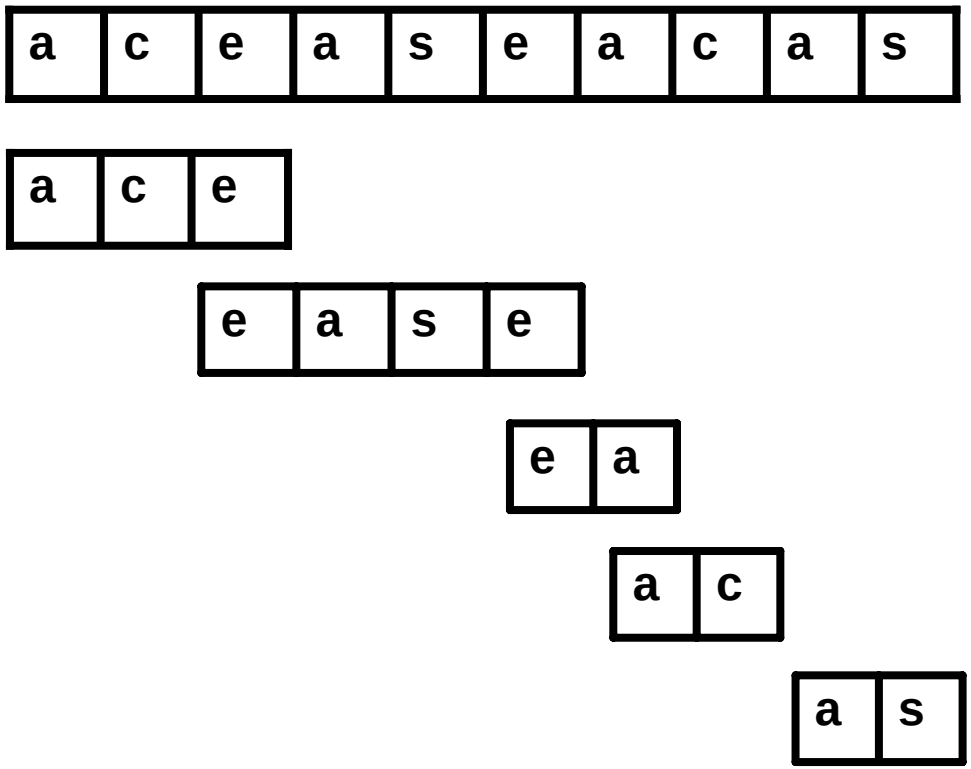
The Aho-Corasick algorithm

Aho-Corasick trie example for the set ‘ace’, ‘as’ and ‘ease’ showing all failure transitions



- An extension of Knuth-Morris-Pratt in matching a set of patterns.
- The patterns are arranged in a trie-like data structure.
- Each trie node corresponds to matching a prefix of some pattern(s).
- There is also a set of *failure* transitions:
  - Those transitions go between nodes of the trie.
  - A transition from a node representing the prefix z to a node representing a prefix y:
    - y is the longest prefix in the set of patterns which is also a proper suffix of z.

The Aho-Corasick algorithm (cont.)



Finite Automaton

- **Definition 3** A finite automaton is a 5-tuple, $M = (Q, q_0, A, \Sigma, \delta)$, where
- Q is a finite set of states.
- $q_0 \in Q$ is the start state.
- $A \subseteq Q$ is the distinguished set of accepting states.
- Σ is a finite input alphabet.
- $\delta : Q \times \Sigma \rightarrow Q$ is the transition function of M.
- A finite automaton M induces a function $\varphi : \Sigma* \rightarrow Q$

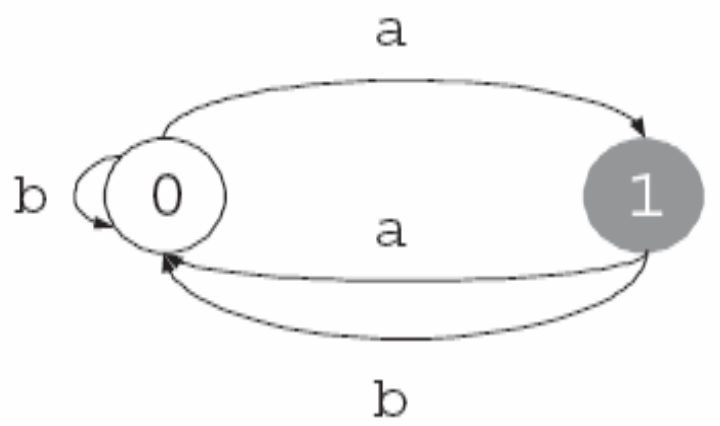
(called final-state function) such that

  - $\varphi(\varepsilon) = q_0$
  - $\varphi(wa) = \delta(\varphi(w), a)$ for $w \in \Sigma^*$, $a \in \Sigma$
- $\varphi(w)$ is the state M ends up after scanning the string w
- M accepts w if and only if $\varphi(w) \in A$

Example of finite automaton

- Given an input alphabet Σ = {a, b} and state set Q = {0, 1}, where q_0 = 0 and A = {1}. The transition function is defined as follows:

δ	a	b
0	1	0
1	0	0



- The automaton accepts the string that end in an odd number of a's

Suffix Function

- **Definition.** Given pattern $P[1 \ldots m]$, $\sigma : \Sigma* \rightarrow \{0, 1, \cdot \cdot \cdot, m\}$ as follows:

$$\sigma(x) = \max\{k |\ P_k \text{ is a suffix of } x\}$$

is called a suffix function corresponding to P.

- **Example.** For $P = ab$, we have $\sigma(\varepsilon) = 0$, $\sigma(ccaca) = 1$, $\sigma(ccab) = 2$.
- **Properties:**
  - If x *is a suffix of* y, then $\sigma(x) \leq \sigma(y)$.
  - $\sigma(x) = m$ iff P *is a suffix of* x.

String-matching Automaton

- The state set Q is {0, 1, · · · ,m}, where
- $q_0 = 0$ and $A = \{m\}$.
- The transition function δ is defined as

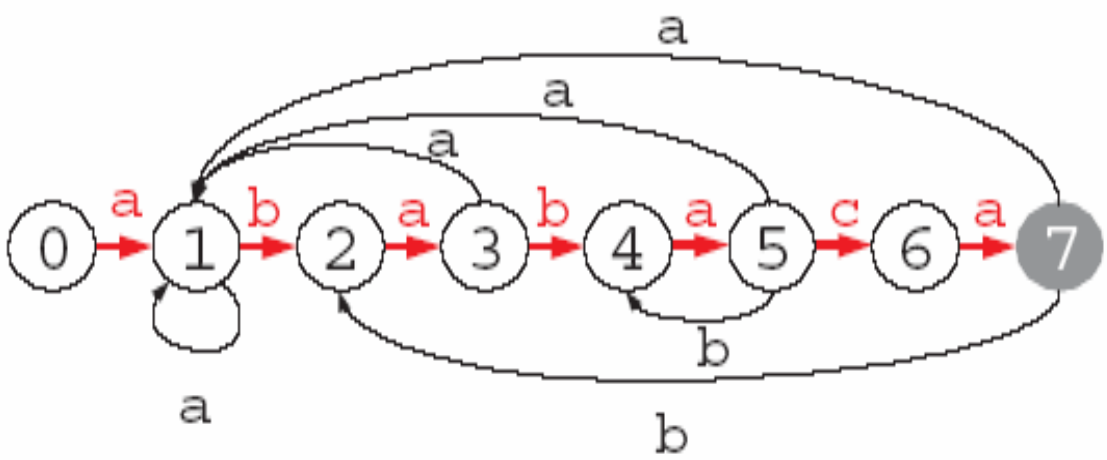
$$\delta(q, a) = \sigma(P_q a)$$

for any state q and character a.

- **Theorem.** If φ is the final-state function of a string-matching automaton for a given pattern P and $T[1 \ldots n]$ is an input text for the automaton, then for $i = 0, 1, \cdots, n$, we have

$$\varphi(T_i) = \sigma(T_i)$$

Example of String-matching Automaton



state	a	b	c	P
0	1	0	0	a
1	1	2	0	b
2	3	0	0	a
3	1	4	0	b
4	5	0	0	a
5	1	4	6	c
6	7	0	0	a
7	1	2	0	

i	—	1	2	3	4	5	6	7	8	9	10	11
$T[i]$	—	a	b	a	b	a	b	a	c	a	b	a
$\phi(T_i)$	0	1	2	3	4	5	4	5	6	7	2	3

Finite Automaton Algorithm

```
FINITE-AUTOMATON-MATCHER(T, δ, m)
1  n ← length[T]
2  q ← 0
3  for i ← 1 to n
4       do q ← δ(q, T[i])
5          if q = m
6             then print "occur with shift" i − m
```

```
COMPUTING-TRANSITION-FUNCTION(P, Σ)
1  m ← length[P]
2  for q ← 0 to m
3       do for each character a ∈ Σ
4             do k ← min(m + 1, q + 2)
5                repeat k ← k − 1
6                   until P_k is a suffix of P_q a
7                δ(q, a) ← k
8  return δ
```

- The matching time is Θ(n)
- The transition function δ can be computed in $O(m^3|\Sigma|)$ time

Shift-Or Algorithm

- Using bit-parallelism
  - simulate the operation of non-deterministic automaton that searches the pattern in the text.
- algorithm
  - builds a table B which for each character stores a bit mask $b_m \ldots b_1$.
    - The mask in B[c] has the *i*-th bit set to zero iff p_i=c.
  - The state of the search is kept in a machine word $D=d_m \ldots d_1$

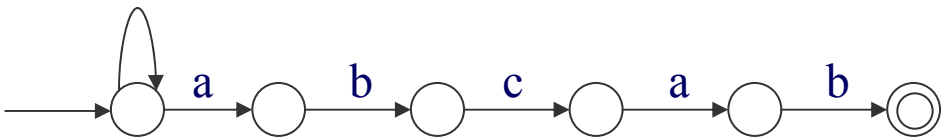
$$D' \leftarrow (D << 1) \mid B[T_j]$$

    - << : shifting all the bits in D one position to the left and setting the right most bit to zero

  - A match is reported whenever d_m is zero.

Shift Or Example(1)

text	d	a	b	c	a	b	c	a
pattern	a	b	c	a	b			



B[a]	0	1	1	0	1
B[b]	1	0	1	1	0
B[c]	1	1	0	1	1
B[*]	1	1	1	1	1

D<<1	1	1	1	1	0
B[d]	1	1	1	1	1
D’	1	1	1	1	1

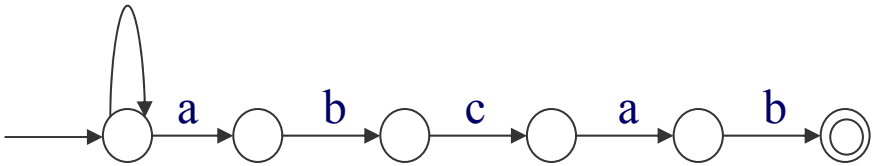
D<<1	1	1	1	1	0
B[a]	1	0	1	1	0
D’	1	1	1	1	0

D<<1	1	1	1	0	0
B[b]	0	1	1	0	1
D’	1	1	1	0	1

D<<1	1	1	0	1	0
B[c]	1	1	0	1	1
D’	1	1	0	1	1

Shift-Or Example(2)

text	d	a	b	c	a	b	c	a
pattern	a	b	c	a	b			



B[a]	0	1	1	0	1
B[b]	1	0	1	1	0
B[c]	1	1	0	1	1
B[*]	1	1	1	1	1

match

D<<1	1	0	1	1	0
B[a]	1	0	1	1	0
D’	1	0	1	1	0

D<<1	0	1	1	0	0
B[b]	0	1	1	0	1
D’	0	1	1	0	1

D<<1	1	1	0	1	0
B[c]	1	1	0	1	1
D’	1	1	0	1	1

D<<1	1	0	1	1	0
B[a]	1	0	1	1	0
D’	1	0	1	1	0

String Matching Allowing Errors

- Approximate string matching:
  - given a string $P=p_1p_2\ldots p_m$ of length m, a string $T=t_1t_2\ldots t_n$ of length n, match P to T with minimum number of errors (Levenshtein distance).
  - The classical solution to approximate string matching is based on dynamic programming:
    - Matrix C[0..m, 0..n] where C[i,j] represents the minimum number of errors needed to match $P_{1..i}$ to $T_{1..j}$

$$C[i,j]=\begin{cases}0 \text{ if } i=0, j=0\\ C[i-1,0]+D(p_i) \text{ if } j=0\\ C[0,j-1]+I(t_j) \text{ if } i=0\\ \min\{C[i-1,j-1]+S(p_i,t_j), C[i-1,j]+D(p_i), C[i,j-1]+I(t_j)\}\end{cases}$$

where $S(p_i,t_j)$ is the cost of substituting p_i with t_j, $D(p_i)$ the cost of deleting p_i and $I(t_j)$ the cost of inserting t_j.

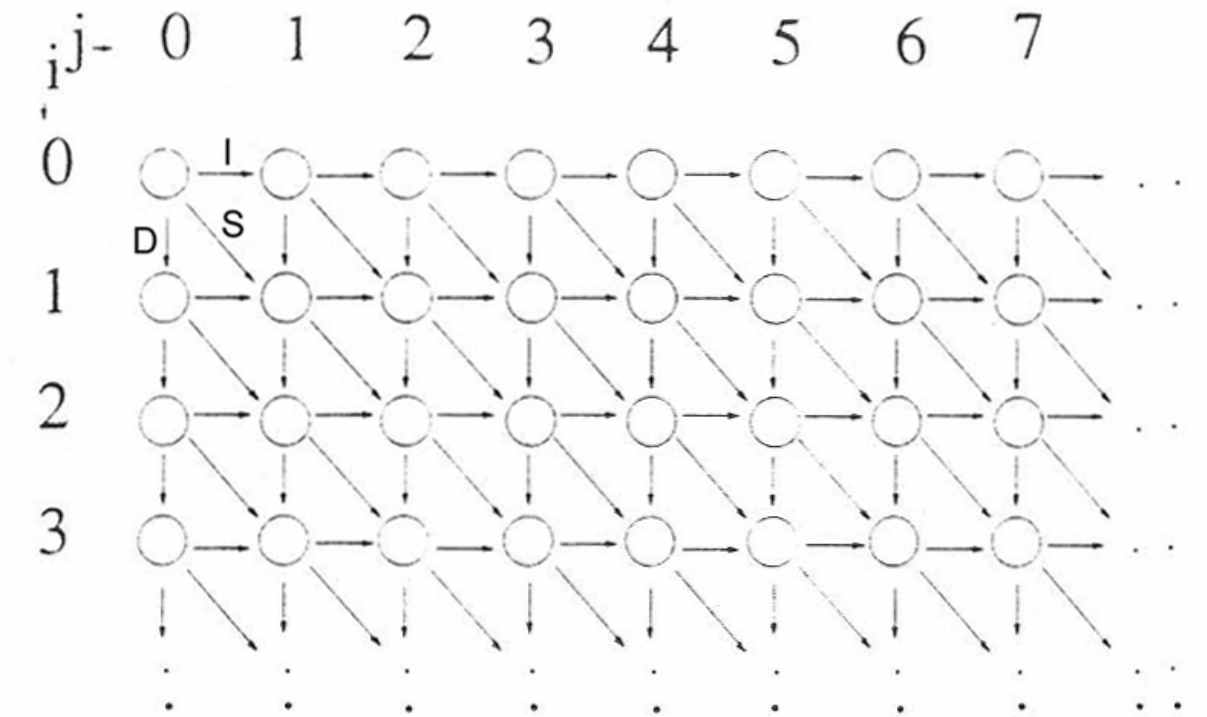
Common assumption: $D(p_i)=I(t_j)=1$, $S(p_i,t_j)=\begin{cases}1 \textit{ if } p_i\neq t_j\\ 0 \textit{ otherwise}\end{cases}$,.

String Matching Allowing Errors (Cont.)

		s	u	r	g	e	r	y
	0	1	2	3	4	5	6	7
s	1	**0**	1	2	3	4	5	6
u	2	1	**0**	1	2	3	4	5
r	3	2	1	**0**	1	2	3	4
v	4	3	2	1	**1**	2	3	4
e	5	4	3	2	2	**1**	**2**	3
y	6	5	4	3	3	2	2	**2**

- The dynamic programming algorithm to compute the edit distance between “survey” and “surgery”. The bold entries show the path to the final result
- O(mn) the running time of the algorithm
- Only O(min(m,n)) the space requirement: only the previous column or row must be stored in order to compute the new column or row respectively.

String Matching Allowing Errors (cont.)



- The dependencies in the dynamic programming recurrence may be represented by a lattice graph
- The vertex in position (i,j) of the lattice graph represents entry (i,j) of the cost matrix.
- Each edge of the lattice graph is assigned a weight equal to the cost of the corresponding edit operation.
- The weights are obtained as follows:
  - The weight of an edge of type <(i-1,j) ,(i,j)> is $D(p_i)$
  - An edge of type <(i-1,j-1) ,(j-1)> has weight $S(p_i,t_j)$
  - The weight of an edge of type <(i ,j-1) ,(i,j)> is $I(t_j)$
- Cost(i,j) is the length of a shortest path from vertex (0,0) to vertex(i,j)
- A match is reported at text positions j such that C[m,j]<=k
- O(mn) the execution time of the algorithm
- O(m) space: only the previous column of the matrix is needed.

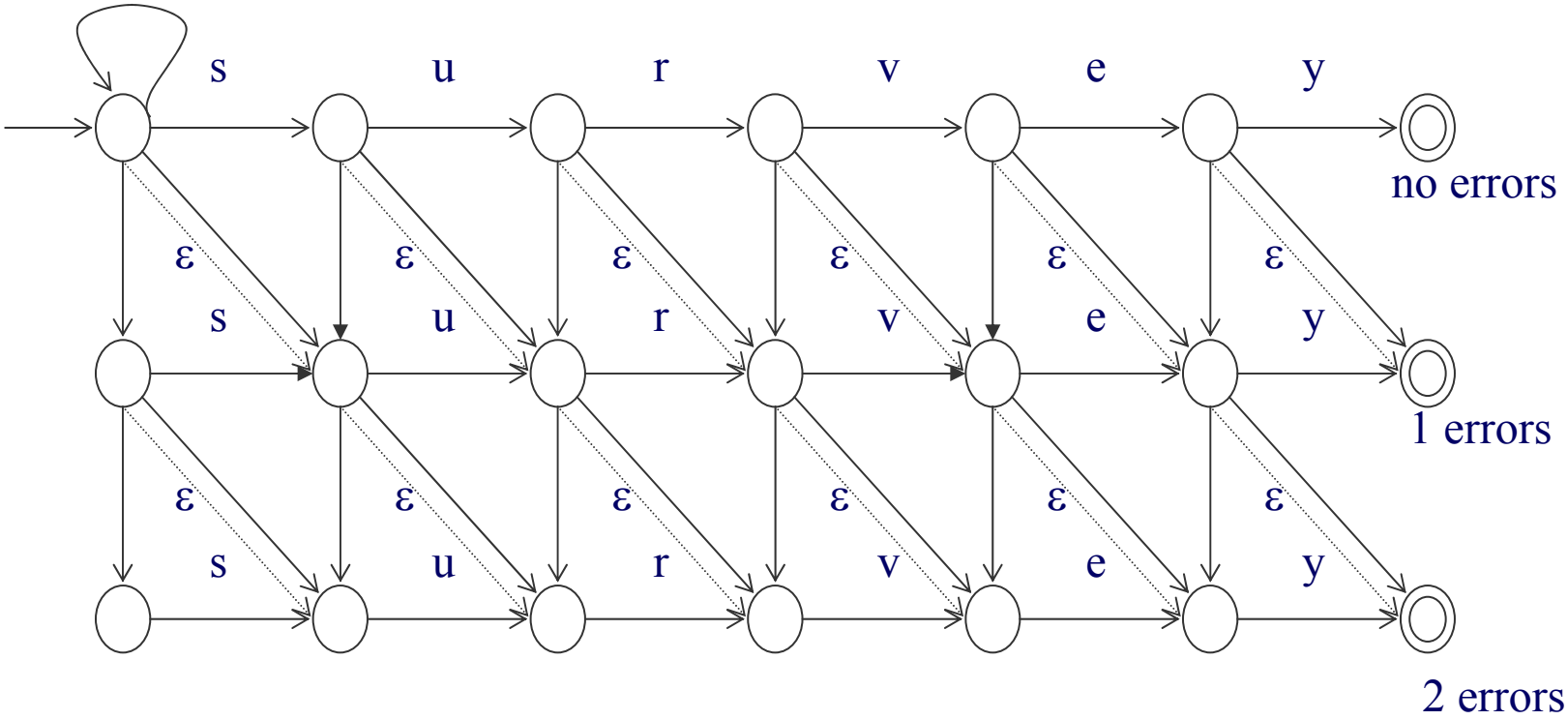
Text Searching allowing errors

- How to adapt the previous algorithm to search a short pattern P[1..m] in a long text T[1..n]?
- Algorithm is basically the same:
  - Set C[0,j]=0 for all j=0…n → begin matching after the first j characters of text.
  - All other expressions are the same

		m	i	n	o	r	␣	s	u	r	g	e	r	y
	0	**0**	**0**	**0**	**0**	**0**	**0**	0	0	0	0	0	0	0
s	1	1	1	1	1	1	1	**0**	1	1	1	1	1	1
u	2	2	2	2	2	2	2	1	**0**	1	2	2	2	2
r	3	3	3	3	3	2	3	2	1	**0**	1	2	2	3
v	4	4	4	4	4	3	4	3	2	1	**1**	2	3	3
e	5	5	5	5	5	4	5	4	3	2	2	**1**	**2**	3
y	6	6	6	6	6	5	6	5	4	3	3	**2**	**2**	**2**

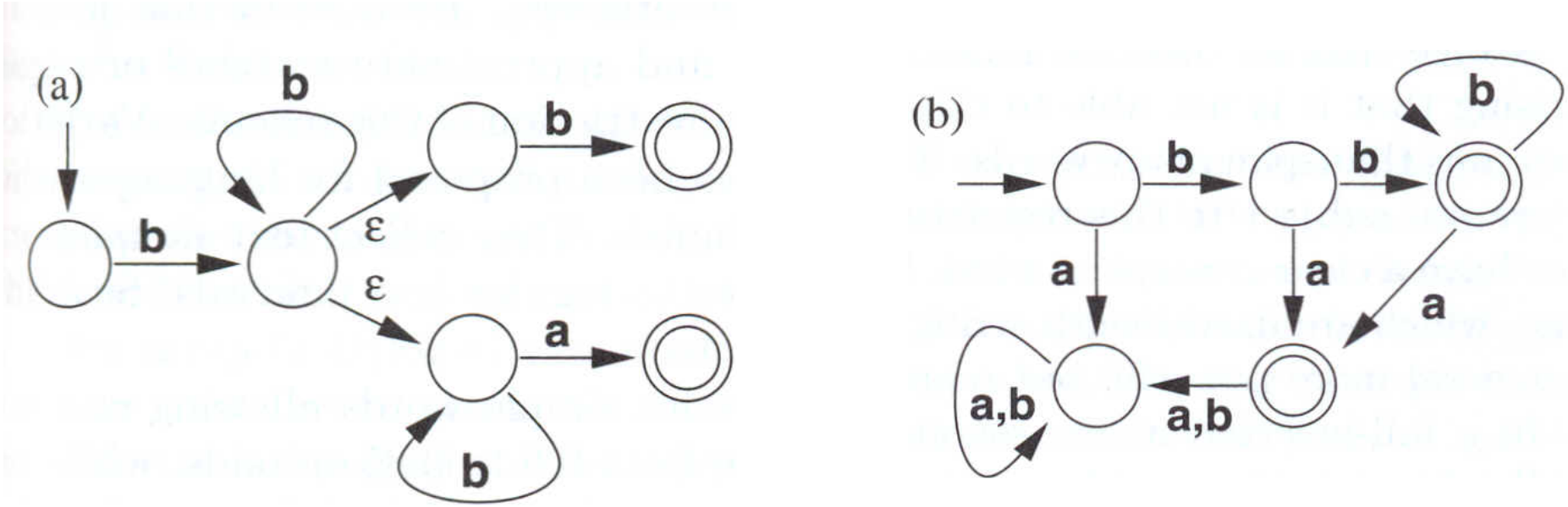
• The dynamic programming algorithm to search “survey” in the text “minor surgery” with two errors.
• Bold entries indicate matching text positions

Text Searching allowing errors (cont.)



- An non-deterministic finite automaton (NFA) for approximate string matching of the pattern ‘survey’ with two errors
- Unlabeled transitions match any character
- Horizontal arrows represent matching a character
- Vertical arrows represent insertions into the pattern
- Solid diagonal arrows represent replacements
- Dashed diagonal arrows represent deletions in the pattern

Regular Expressions



The non-deterministic *(a)* and deterministic *(b)* automata for the regular expression bb*(b | b*a).

- O(m) the size of the non-deterministic finite automaton, where m is the length of the regular expression
- $O(m2^m)$ the size and the construction time of the deterministic finite automaton
- O(n) the time of searching any regular expression in text of n characters

Tradeoff of index space versus word searching time