

# Pokročilá Algoritmizace

Začátek v 11:00

Přednáší: Daniel Průša

[prusa@fel.cvut.cz](mailto:prusa@fel.cvut.cz)



# Text Search

Dictionary NFA and text search

Dictionary DFA and text search

Hamming distance and Dynamic Programming?

Levenshtein distance and Dynamic Programming

Resume






Boyer-Moore text search approach

**Literature:**

Borivoj Melichar, Jan Holub, Tomas Polcar  
TEXT SEARCHING ALGORITHMS VOLUME I.

CTU, FEE, Nov 2005

# Co říkáte na distanční výuku?

- A. Příjemná změna, užívám si to. 
- B. Má to své klady i zápory. 
- C. Hrůza, už aby bylo zase vše v normálu.  
- D. Nic z výše uvedeného nevyjadřuje mé pocity. 

# Kde se právě teď nacházíte?



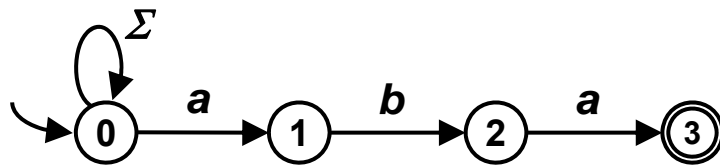
## Bit representation of NFA

Size of transition table  $T$  is  $|Q| \times |\Sigma|$  and each its element  $T[i, k]$  corresponds to state  $q_i \in Q$  and symbol  $a_k \in \Sigma$ .  $T[i, k]$  is vector of length  $|Q|$  and it holds:

$$T[i, k][j] == 1 \Leftrightarrow q_j \in \delta(q_i, a_k).$$

For bit vector  $F$  of final states holds  $F[j] == 1 \Leftrightarrow q_j \in F_A$

## Example



	a	b	z
0	0,1	0	0
1		2	
2	3		
3			

$F$

$$z \in \Sigma - \{a, b\}$$

Automaton A detects pattern *aba* in a text.

$T$	a	b	z
$i=0$	1	1	1
	1	0	0
	0	0	0
	0	0	0
$i=1$	0	0	0
	0	1	0
	0	0	0
	0	0	0
$i=2$	0	0	0
	0	0	0
	1	0	0
	0	0	0
$i=3$	0	0	0
	0	0	0
	0	0	0

Bit representation of automaton A.

$F$

0	0	0	1
---	---	---	---

starting configuration

text symbols: - a c c a b c a a b a

time →

A

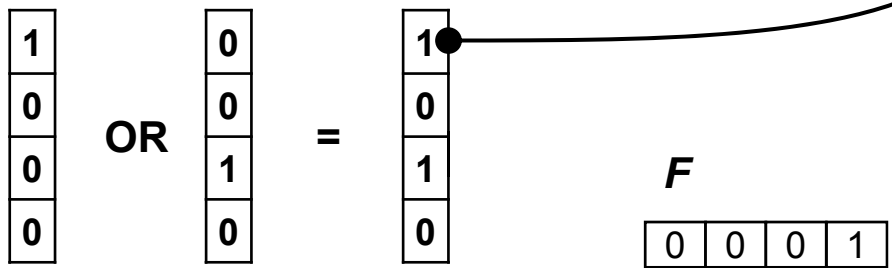
sets of states represented by bit arrays during computation

0	1	1	1	1	1	1	1	1	1	1	1
1	0	1	0	0	1	0	0	1	1	0	1
2	0	0	0	0	0	1	0	0	0	1	0
3	0	0	0	0	0	0	0	0	0	0	1
sets of states	{0}	{0,1}	{0}	{0}	{0,1}	{0,2}	{0}	{0,1}	{0,1}	{0,2}	{0,1,3}

T

	a	b	z
i=0	1	1	0
	1	0	0
	0	0	0
	0	0	0
i=1	0	0	0
	0	0	0
	0	1	0
	0	0	0
i=2	0	0	0
	0	0	0
	1	0	0
i=3	0	0	0
	0	0	0
	0	0	0

**example**  
Automaton is in states {0,1}, it reads symbol b



Simulation of work of a NFA without  $\varepsilon$ -transitions  
Basic method, implemented with bit vectors.

Input: Bit table T of transitions, bit vector F of final states,  
number of states Q.size, text in array t (indexed from 1).

Output: Simulated run and output of the automaton.

(notation in format [0101...00] denotes characteristic vector of set of states)

```
S[0] = [100..0]; i = 1;    // init
while( (i <= t.length) && (S[i-1]!=[000...0]) ) {
    for( j=0; j < Q.size; j++ )
        if( (S[i][j] == 1) && (F[j] == 1) )
            print( q[j].final_state_info );
    S[i] = [000...0];
    for( j=0; j < Q.size; j++ )
        if( S[i-1][j]==1 )
            S[i] = S[i] | T[j][t[i]]; // logical "or"
    i++;
}
```

Dictionary over an alphabet  $A$  is a finite set of strings (patterns) from  $A^*$ .  
Dictionary automaton searches the text for any pattern in the given dictionary.

### Recycle older knowledge

1. Dictionary is a finite language.
2. Each finite language is a regular language.
3. Each regular language can be described by a regular expression.
4. Any language described by a regular expression can be searched for in any text using appropriate NFA/DFA.

### Example

Alphabet

$A = \{a, c, d, e, g, h, i, l, m, n, o, q, r, s, t, u, v, y\}$

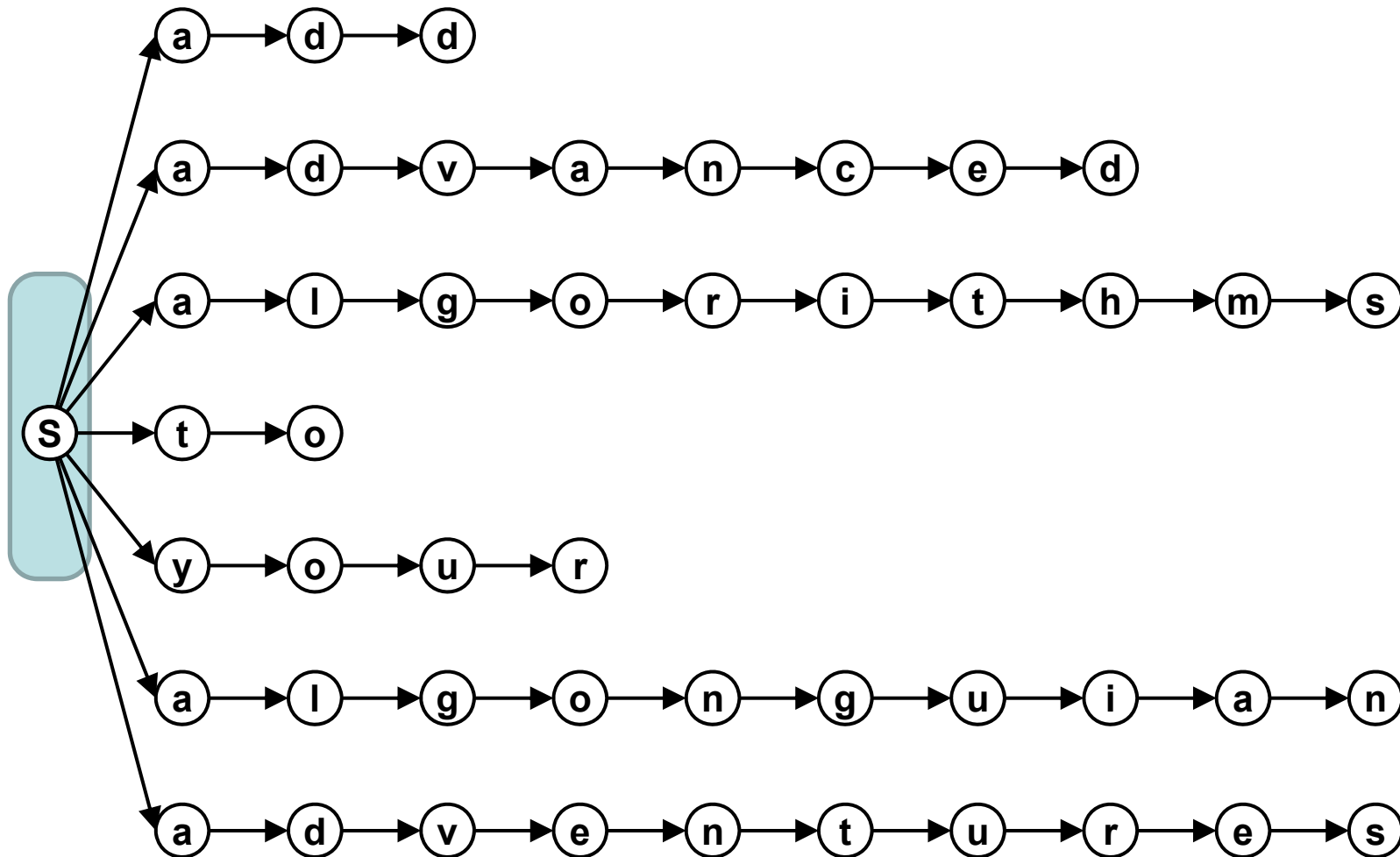
Dictionary

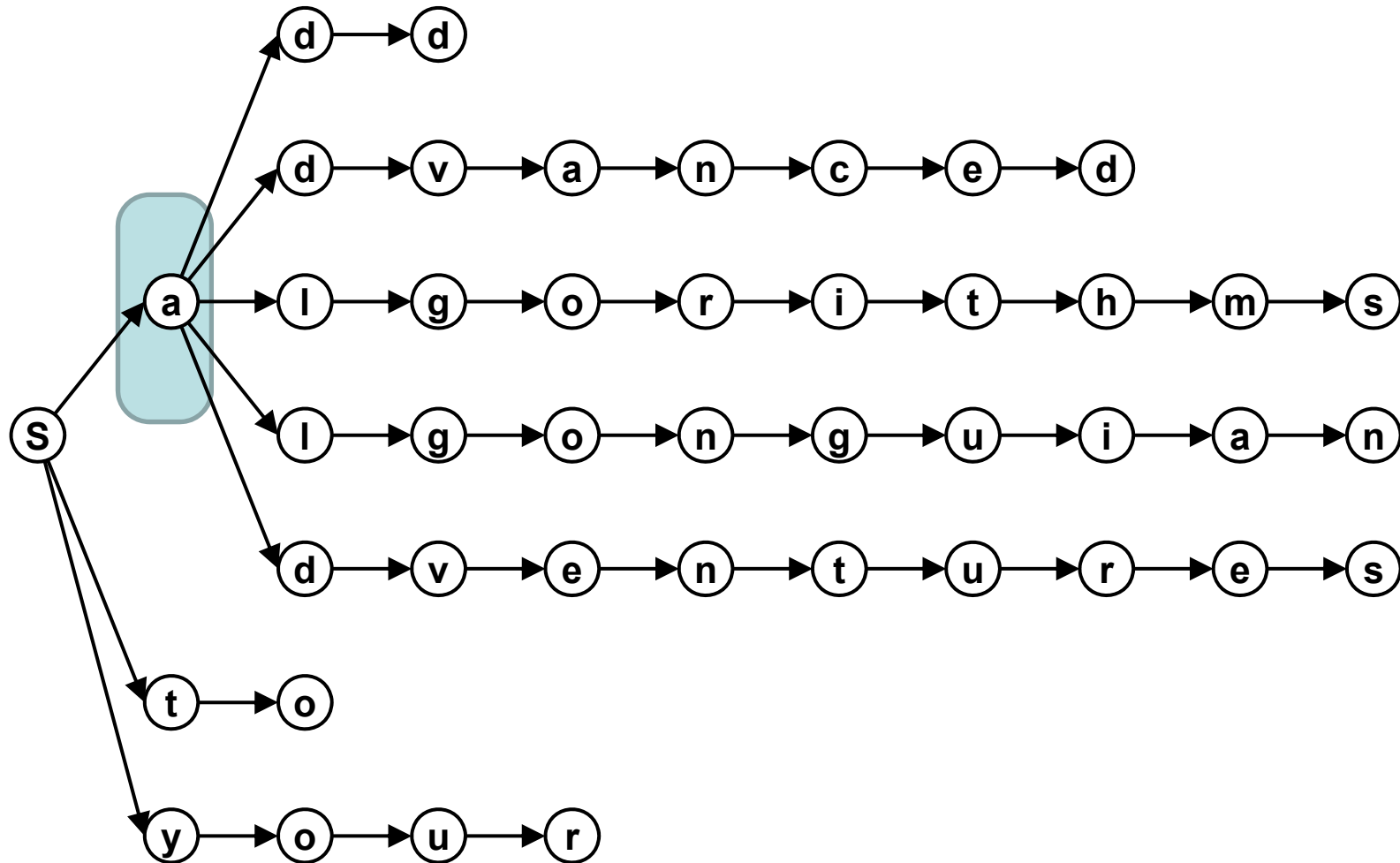
$D = \{\text{"add"}, \text{"advanced"}, \text{"algorithms"}, \text{"to"}, \text{"your"}, \text{"algonquian"}, \text{"adventures"}\}$

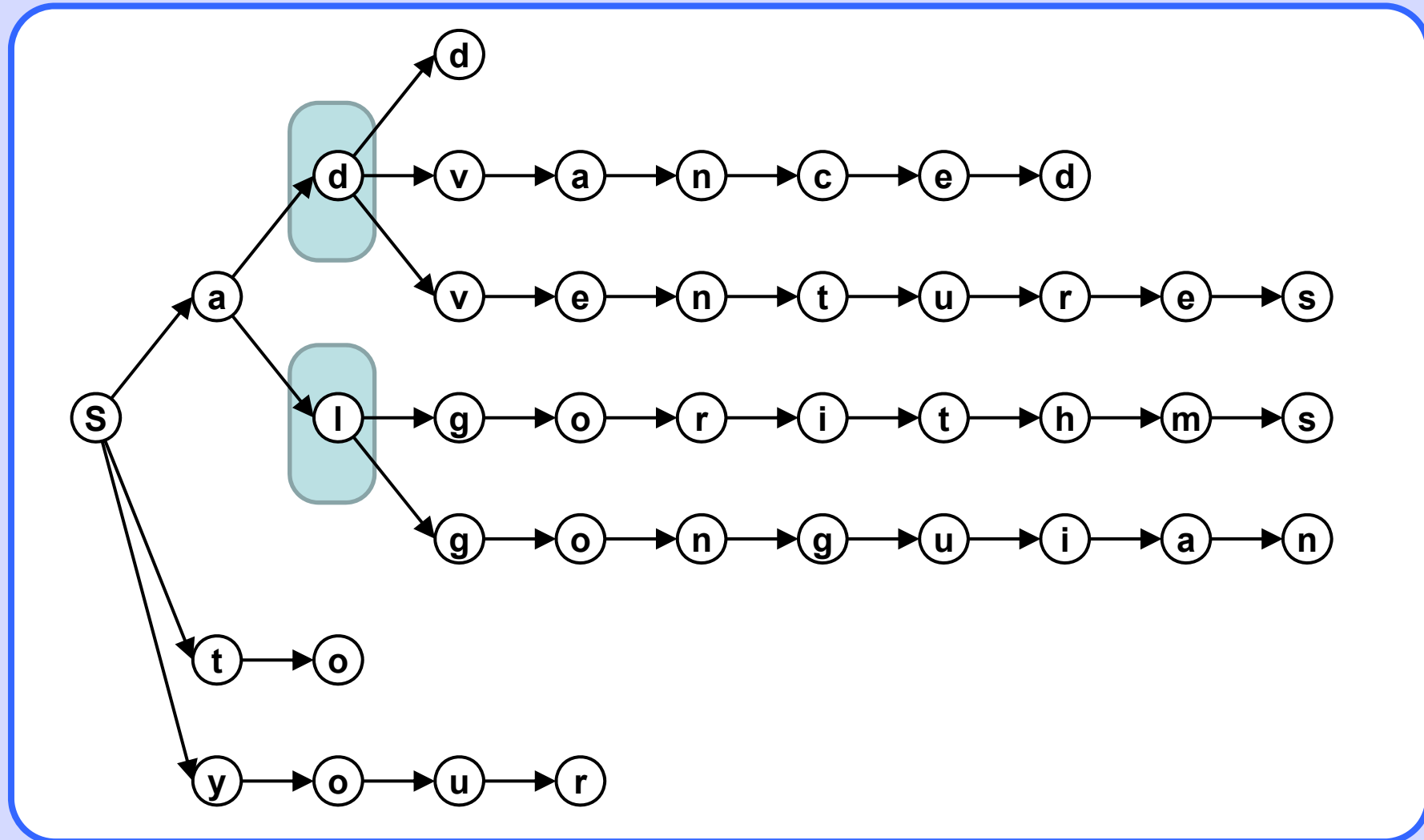
The *Algonquian* are one of the most populous and widespread North American native language groups.

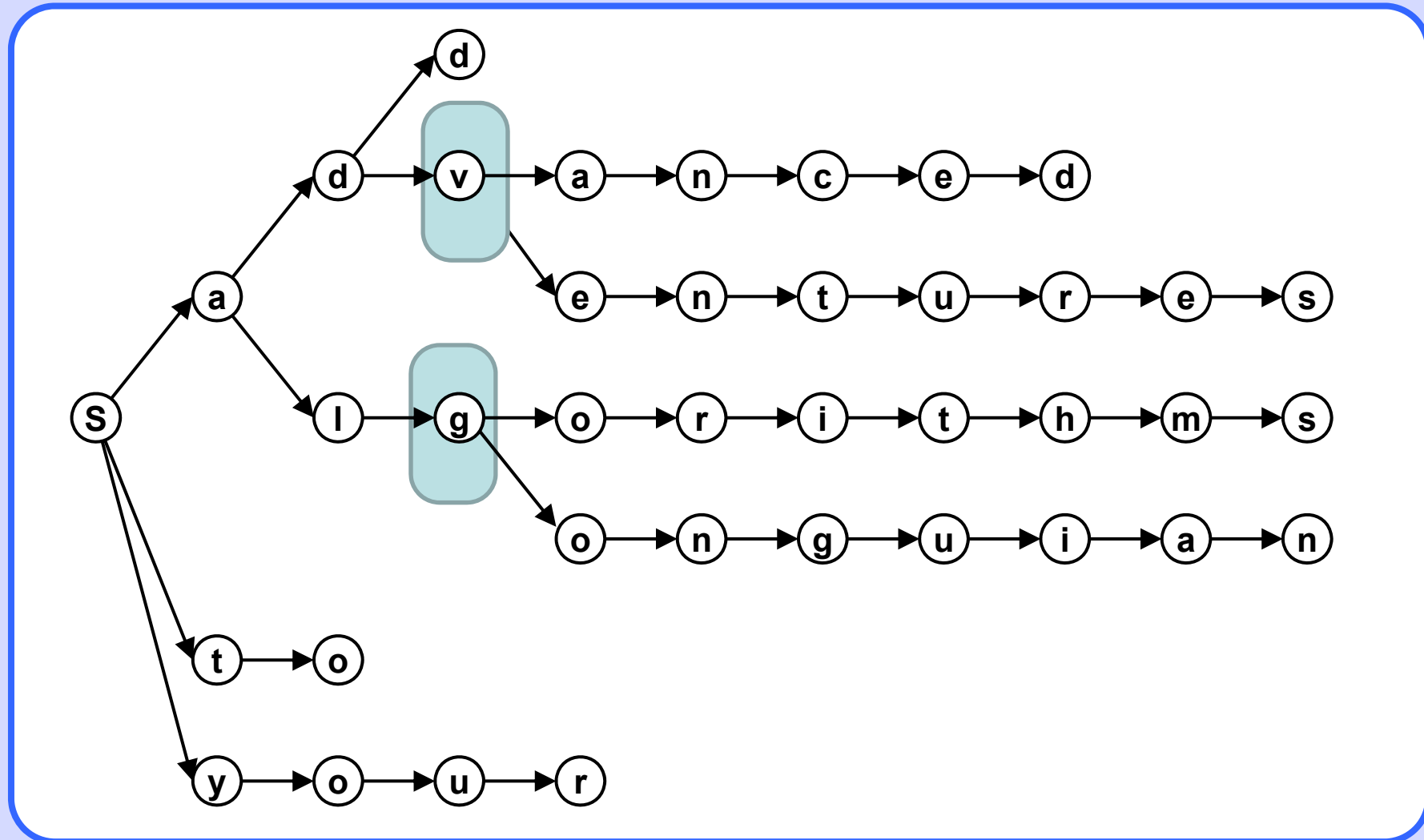


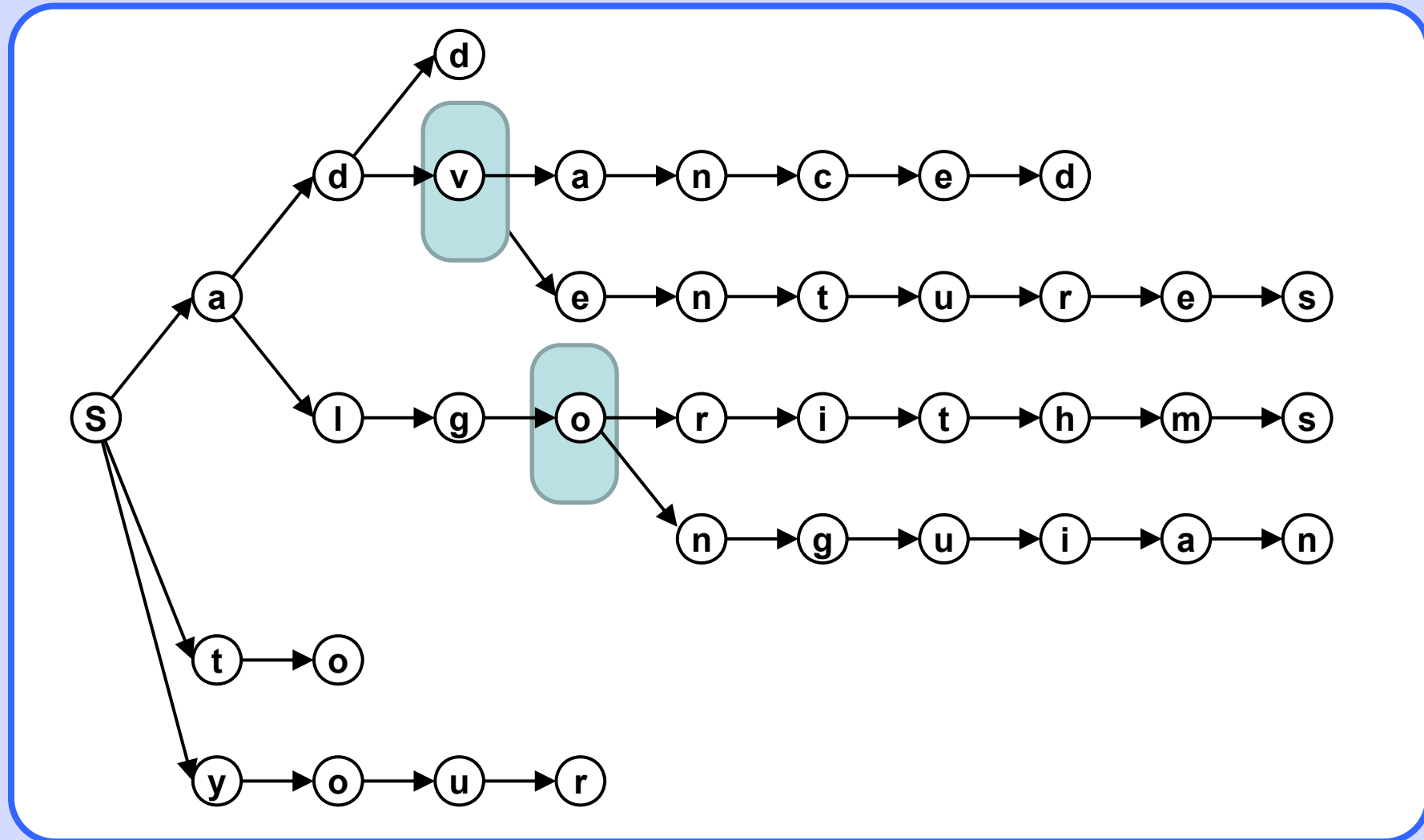
Merge repeatedly into a single state any two states A and B such that path from S to A and from S to B are of equal length and contain equal sequence of transition labels. You may find e.g. BFS/DFS to be useful.





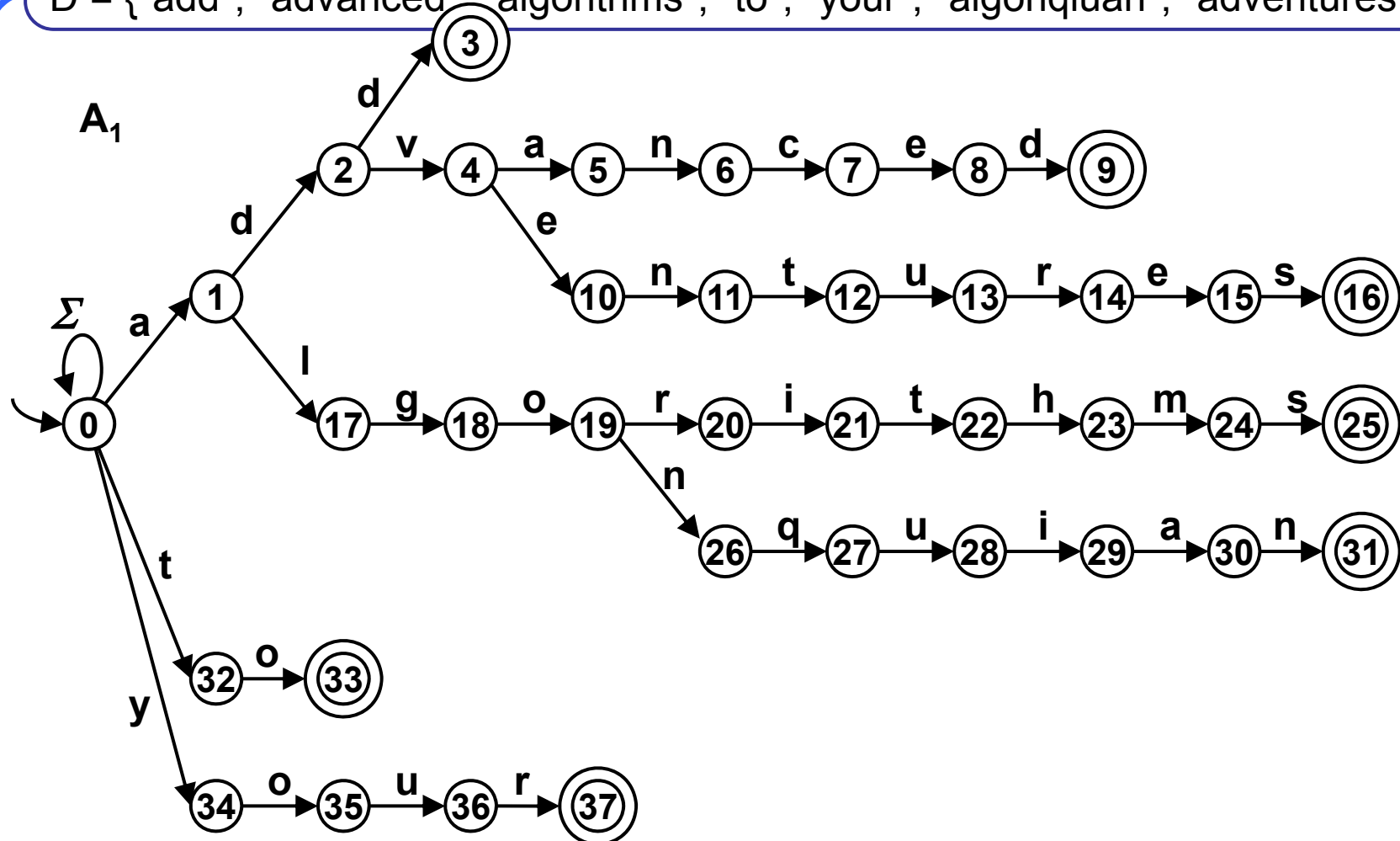


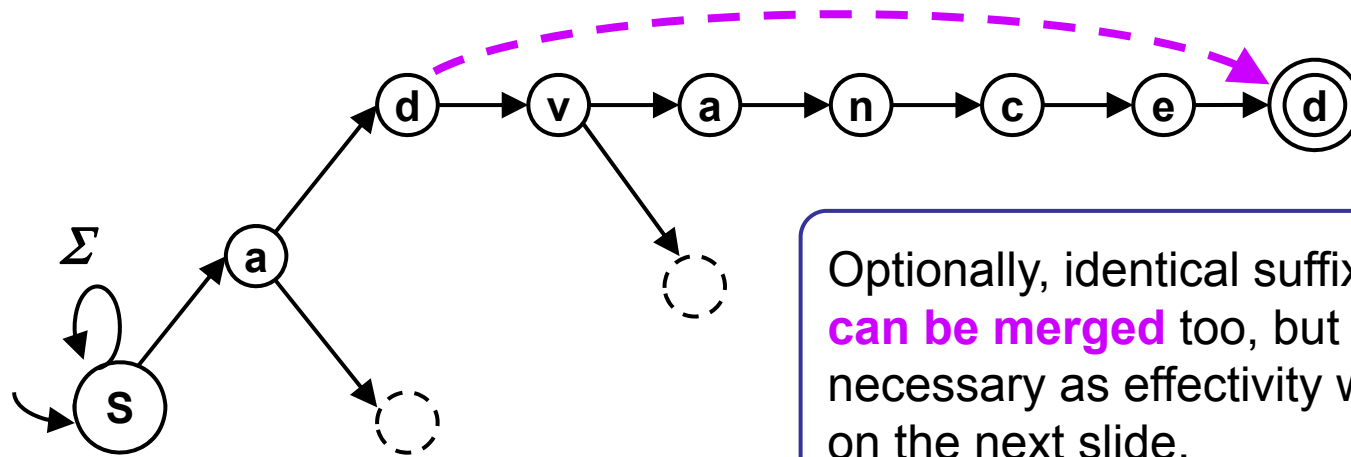




Search NFA for dictionary

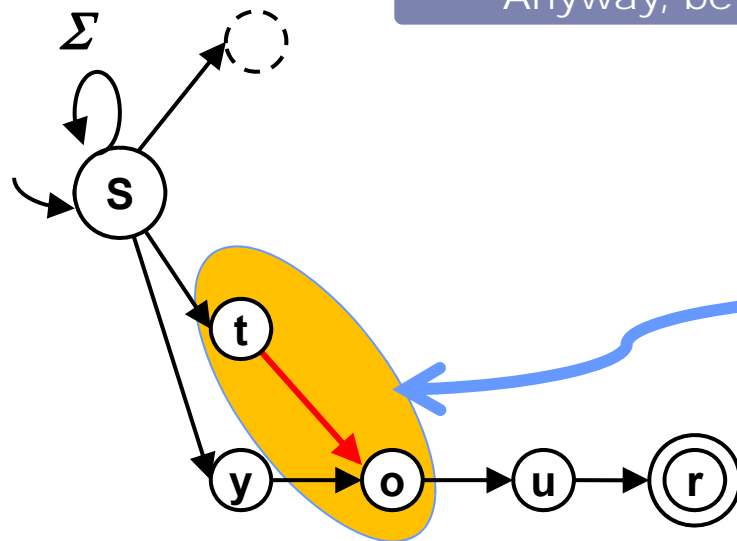
$D = \{ \text{"add", "advanced", "algorithms", "to", "your", "algonquian", "adventures"} \}$





Optionally, identical suffixes **can be merged** too, but it is not necessary as effectivity will be granted on the next slide.

Anyway, be careful.



This is a wrong construction. It would incorrectly add word "tour" to the dictionary.

The transition diagram of a dictionary NFA, like  $A_1$  in the previous example, is a **directed tree** with the start state **in the root**.  
The **only loop** is the self-loop **in the start state** labeled by the whole alphabet.  
This NFA has an useful property:

### Effectivity

**Transforming dictionary NFA of this shape to DFA  
does not increase the number of states.**

### Example

The transition diagram of the resulting DFA has 38 states (same as NFA) and 684 transitions. It would not fit nicely into one slide, therefore we present only the transition table... :

Homework: Draw it!



	a	c	d	e	g	h	i	l	m	n	o	q	r	s	t	u	v	y
0	0,1	0	0	0	0	0	0	0	0	0	0	0	0	0	0,32	0	0	0,34
0,1	0,1	0	0,2	0	0	0	0	0,17	0	0	0	0	0	0	0,32	0	0	0,34
0,32	0,1	0	0	0	0	0	0	0	0	0	0,33	0	0	0	0,32	0	0	0,34
0,34	0,1	0	0	0	0	0	0	0	0	0,35	0	0	0	0	0,32	0	0	0,34
0,2	0,1	0	0,3	0	0	0	0	0	0	0	0	0	0	0	0,32	0	0,4	0,34
0,17	0,1	0	0	0	0	0	0	0	0	0	0	0	0	0	0,32	0	0	0,34
0,33	0,1	0	0	0	0	0	0	0	0	0	0	0	0	0	0,32	0	0	0,34
0,35	0,1	0	0	0	0	0	0	0	0	0	0	0	0	0	0,32	0,36	0	0,34
0,3	0,1	0	0	0	0	0	0	0	0	0	0	0	0	0	0,32	0	0	0,34
0,4	0,1,5	0	0	0	0	0	0	0	0	0	0	0	0	0	0,32	0	0	0,34
0,18	0,1	0	0	0	0	0	0	0	0	0	0,19	0	0	0	0,32	0	0	0,34
0,36	0,1	0	0	0	0	0	0	0	0	0	0	0	0,37	0	0,32	0	0	0,34
0,1,5	0,1	0	0	0,2	0	0	0	0,17	0	0,6	0	0	0	0	0,32	0	0	0,34

F

F

Transition table of DFA  $A_2$  equivalent to dictionary NFA  $A_1$ .

Continue...

... continued

	a	c	d	e	g	h	i	l	m	n	o	q	r	s	t	u	v	y
0,10	0,1	0	0	0	0	0	0	0	0	0,11	0	0	0	0	0,32	0	0	0,34
0,19	0,1	0	0	0	0	0	0	0	0	0,26	0	0	0,20	0	0,32	0	0	0,34
0,37	0,1	0	0	0	0	0	0	0	0	0	0	0	0	0	0,32	0	0	0,34
0,6	0,1	0,7	0	0	0	0	0	0	0	0	0	0	0	0	0,32	0	0	0,34
0,11	0,1	0	0	0	0	0	0	0	0	0	0	0	0	0	0,12, 32	0	0	0,34
0,26	0,1	0	0	0	0	0	0	0	0	0	0	0,27	0	0	0,32	0	0	0,34
0,20	0,1	0	0	0	0	0	0,21	0	0	0	0	0	0	0	0,32	0	0	0,34
0,7	0,1	0	0	0,8	0	0	0	0	0	0	0	0	0	0	0,32	0	0	0,34
0,12, 32	0,1	0	0	0	0	0	0	0	0	0	0,33	0	0	0	0,32	0,13	0	0,34
0,27	0,1	0	0	0	0	0	0	0	0	0	0	0	0	0	0,32	0,28	0	0,34
0,21	0,1	0	0	0	0	0	0	0	0	0	0	0	0	0	0,22, 32	0	0	0,34
0,8	0,1	0	0,9	0	0	0	0	0	0	0	0	0	0	0	0,32	0	0	0,34
0,13	0,1	0	0	0	0	0	0	0	0	0	0	0	0,14	0	0,32	0	0	0,34

F

continue...

... continued

	a	c	d	e	g	h	i	l	m	n	o	q	r	s	t	u	v	y	
0,28	0,1	0	0	0	0	0	0,29	0	0	0	0	0	0	0	0,32	0	0	0,34	
0,22, 32	0,1	0	0	0	0	0,23	0	0	0	0	0,33	0	0	0	0,32	0	0	0,34	
0,9	0,1	0	0	0	0	0	0	0	0	0	0	0	0	0	0,32	0	0	0,34	F
0,14	0,1	0	0	0,15	0	0	0	0	0	0	0	0	0	0	0,32	0	0	0,34	
0,23	0,1	0	0	0	0	0	0	0	0,24	0	0	0	0	0	0,32	0	0	0,34	
0,29	0,1, 30	0	0	0	0	0	0	0	0	0	0	0	0	0	0,32	0	0	0,34	
0,15	0,1	0	0	0	0	0	0	0	0	0	0	0	0	0,16	0,32	0	0	0,34	
0,24	0,1	0	0	0	0	0	0	0	0	0	0	0	0	0,25	0,32	0	0	0,34	
0,1, 30	0,1	0	0,2	0	0	0	0	0,17	0	0,31	0	0	0	0	0,32	0	0	0,34	
0,16	0,1	0	0	0	0	0	0	0	0	0	0	0	0	0	0,32	0	0	0,34	F
0,25	0,1	0	0	0	0	0	0	0	0	0	0	0	0	0	0,32	0	0	0,34	F
0,31	0,1	0	0	0	0	0	0	0	0	0	0	0	0	0	0,32	0	0	0,34	F

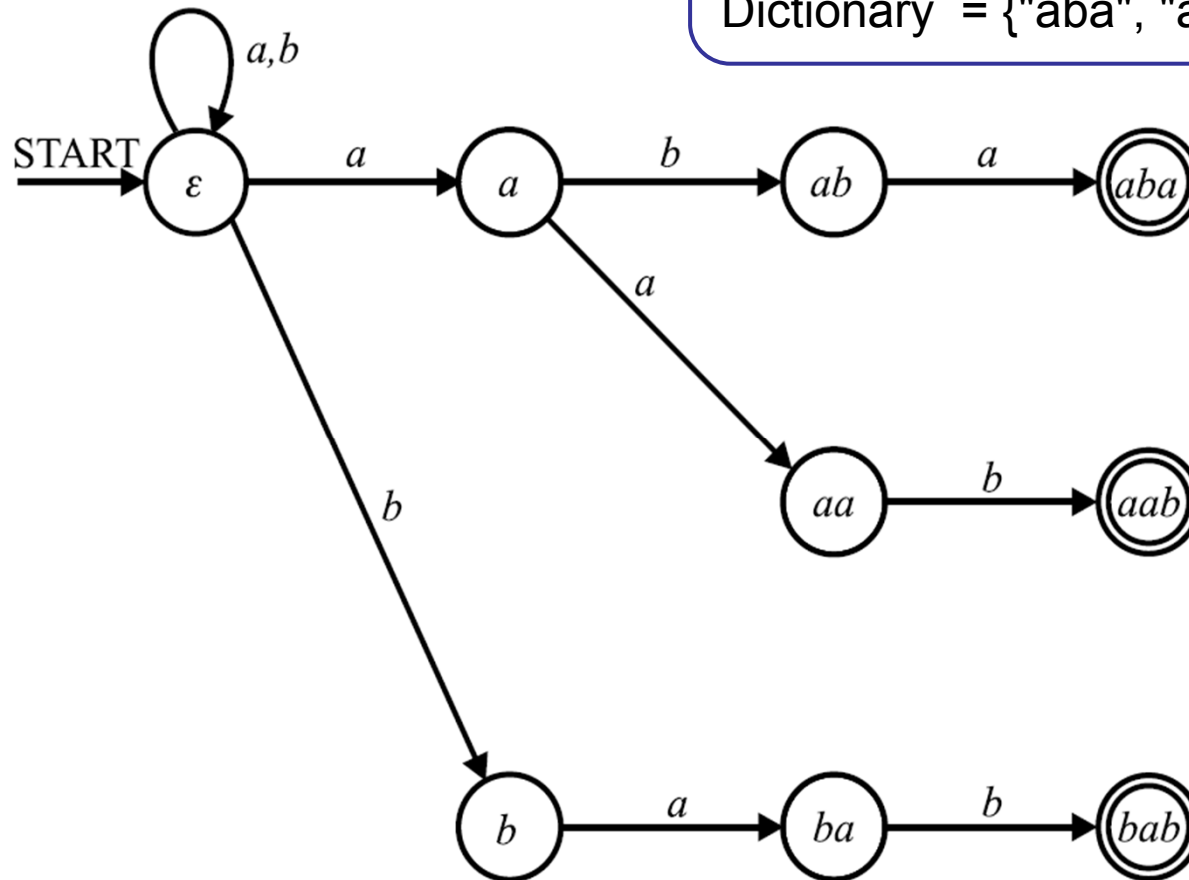
... finished.

Example of dictionary automaton whose transition diagram fits to one slide.

NFA

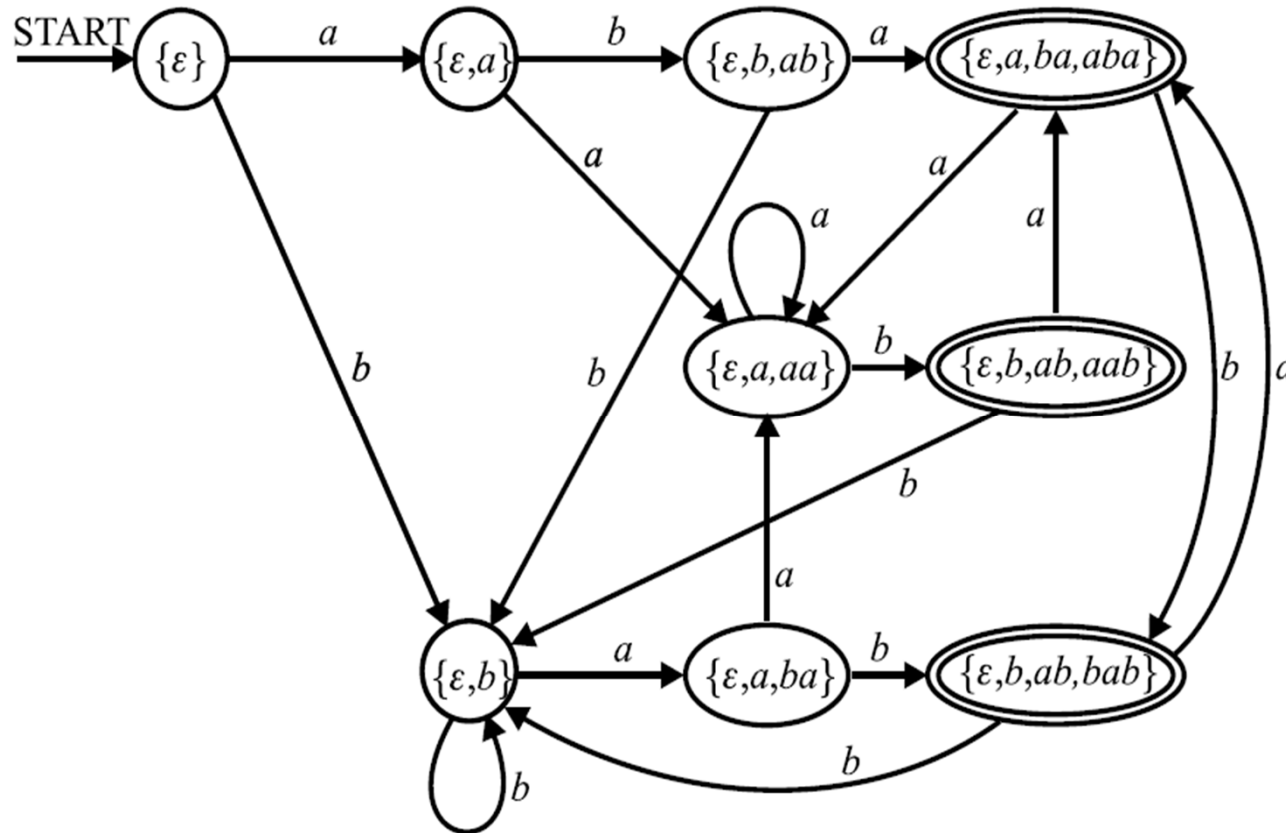
Alphabet = {a, b}

Dictionary = {"aba", "aab", "bab"}

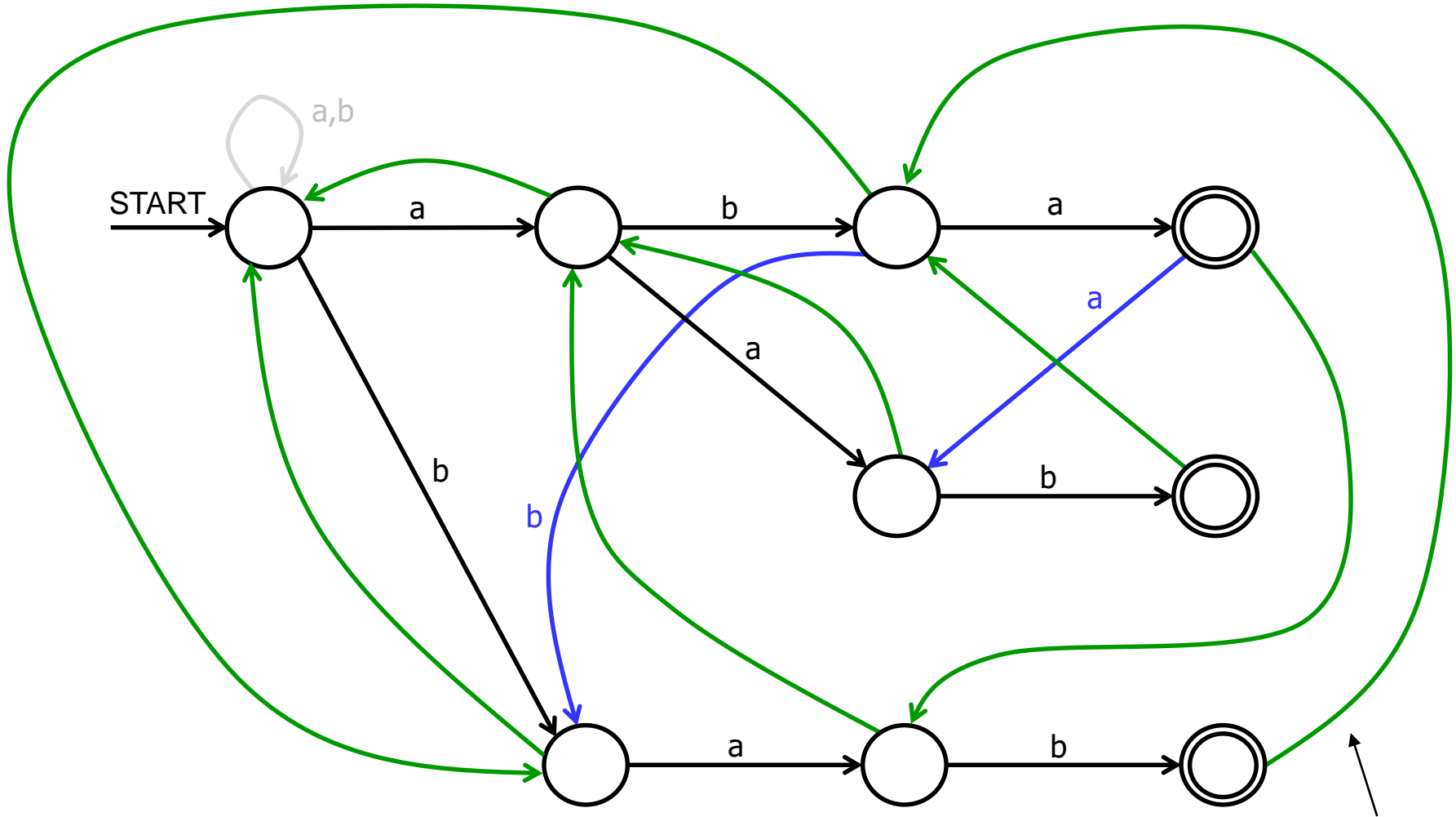


Alphabet = {a, b}  
 Dictionary = {"aba", "aab", "bab"}

DFA



# Aho-Corasick algorithm



Link to the longest proper suffix of "bab" contained in the prefix tree.

**DP approach to text search considering Hamming distance**

Alphabet  $\{a,b,c,d\}$ , pattern  $P$ : adbbca, text  $T$ : adcabcaabadbcca.

For each alignment  $P$  with  $T$  determine

Hamming distance between  $P$  and  $t[k-m+1], t[k-m+2], \dots, t[k]$

<b>T:</b>	t[1]	...	t[k-m+1]	...	t[k-1]	t[k]	...	t[n]
<b>P:</b>			p[1]	...	p[m-1]	p[m]		

**Method**

Let pattern  $P$  be  $p[1], p[2], \dots, p[m]$ , let text  $T$  be  $t[1], t[2], \dots, t[n]$ .

Create dynamic programming table  $D[m+1][n+1]$ , whose elements  $d[i][k]$  are defined as follows:

- $d[0][k] = 0$  // for  $k = 0, \dots, n$
- if  $(p[i] == t[k])$   $d[i][k] = d[i-1][k-1]$   
 else  $d[i][k] = d[i-1][k-1] + 1$  // for  $1 \leq i \leq k, i \leq m, k \leq n$ ,

Fill the table row by row. Element  $d[m][k]$  holds the Hamming distance of  $P$  from the substring  $t[k-m+1], t[k-m+2], \dots, t[k]$ .

Alphabet {a,b,c,d}, pattern P: adbbca, text T: adcabcaababbca.

D	-	a	d	c	a	b	c	a	a	b	a	d	b	b	c	a
-	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
a	-	0	1	1	0	1	1	0	0	1	0	1	1	1	1	0
d	-	-	0	2	2	1	2	2	1	1	2	0	2	2	2	2
b	-	-	-	1	3	2	2	3	3	1	2	3	0	2	3	3
b	-	-	-	-	2	3	3	3	4	3	2	3	3	0	3	4
c	-	-	-	-	-	3	3	4	4	5	4	3	4	4	0	4
a	-	-	-	-	-	-	-	3	4	5	5	5	4	5	5	0

 Highlighted cells represent a match between the text and the pattern.

Though it looks scientifically advanced,  
it is, in fact, only a basic naive approach :-).

Each diagonal corresponds to some alignment of pattern with text  
where mismatches in this alignment are counted one by one.



**DP approach to text search considering Levenshtein distance**

Let pattern  $P$  be  $p[1], p[2], \dots, p[m]$ , let text  $T$  be  $t[1], t[2], \dots, t[n]$ .

Create dynamic programming table  $D[m+1][n+1]$ , whose elements  $d[i][k]$  are defined as follows:

1.  $d[i][0] = i$ ;  $d[0][k] = 0$ , for  $i = 0, \dots, m$ ,  $k = 1, \dots, n$

2. //  $d[i][k]$  is computed using the information about  
 // the minimum possible number of applications of operations  
 // delete, insert, rewrite to the strings shorter by one last character  
 // and followed by at most one edit operation

for  $1 \leq i \leq m$ ,  $1 \leq k \leq n$ :

$d[i][k] = \text{minimum of}$   
 $d[i-1][k] + 1$ , // delete  $p[i]$   
 if(  $i < m$  )  $d[i][k-1] + 1$ , // insert after  $p[i]$   
 $d[i-1][k-1] + (p[i] == t[k]) ? 0 : 1$  ) // leave or rewrite  $p[i]$

Fill the table row by row. The cell  $d[m][k]$  contains the minimum Levenshtein distance of  $P$  from the substring  $S_{x,k} = t[x], t[x+1], \dots, t[k]$ , where  $x \in \{ k-m+1-d[m][k], \dots, k-m+1+d[m][k] \}$  and the particular value of  $x$  is not known.

Alphabet {a,b,c,d}, pattern P: adbbca, text T: adcabcaababbca.

D	-	a	d	c	a	b	c	a	a	b	a	d	b	b	c	a
-	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
a	1	0	1	1	0	1	1	0	0	1	0	1	1	1	1	0
d	2	1	0	1	1	1	2	1	1	1	1	0	1	2	2	1
b	3	2	1	1	2	1	2	2	2	1	2	1	0	1	2	2
b	4	3	2	2	2	2	2	3	3	2	2	2	1	0	1	2
c	5	4	3	2	3	3	2	3	4	3	3	3	2	1	0	1
a	6	5	4	3	2	4	3	2	3	4	3	4	3	2	1	0



Highlighted cells represent a match between the text and the pattern.

$d[i][k] = \text{minimum of } ($   
 $d[i-1][k] + 1,$  // delete p[i]  
 $\text{if}( i < m ) d[i][k-1] + 1,$  // insert after p[i]  
 $d[i-1][k-1] + (p[i] == t[k]) ? 0 : 1 )$  // leave or rewrite p[i]

$\text{Dist}(\text{"BETELGEUSE"}, \text{"BRUXELLES"}) = 6$

		B	E	T	E	L	G	E	U	S	E
	0	1	2	3	4	5	6	7	8	9	10
B	1	0	1	2	3	4	5	6	7	8	9
R	2	1	1	2	3	4	5	6	7	8	9
U	3	2	2	2	3	4	5	6	6	7	8
X	4	3	3	3	3	4	5	6	7	7	8
E	5	4	3	4	3	4	5	5	6	7	7
L	6	5	4	4	4	3	4	5	6	7	8
L	7	6	5	5	5	4	4	5	6	7	8
E	8	7	6	6	5	5	5	4	5	6	7
S	9	8	7	7	6	6	6	5	5	5	6

## Levenshtein distance of strings

Old stuff

$$\text{Dist}(A, B) = |m - n|$$

if  $n = 0$  or  $m = 0$ 

$$\text{Dist}(A, B) = 1 + \min \left( \begin{array}{l} \text{Dist}(A[1..n-1], B[1..m]), \\ \text{Dist}(A[1..n], B[1..m-1]), \\ \text{Dist}(A[1..n-1], B[1..m-1]) \end{array} \right)$$

if  $n > 0$  and  $m > 0$   
and  $A[n] \neq B[m]$ 

$$\text{Dist}(A, B) = \text{Dist}(A[1..n-1], B[1..m-1])$$

if  $n > 0$  and  $m > 0$   
and  $A[n] = B[m]$ 

Calculation corresponds to ... Operation

 $1 + \text{Dist}(A[1..n-1], B[1..m]),$  ... **Insert**(A,  $n-1$ ,  $B[m]$ ) or **Delete**(B,  $m$ )

 $1 + \text{Dist}(A[1..n], B[1..m-1]),$  ... **Insert**(B,  $m-1$ ,  $A[n]$ ) or **Delete**(A,  $n$ )

 $1 + \text{Dist}(A[1..n-1], B[1..m-1])$  ... **Rewrite**(A,  $n$ ,  $B[m]$ ) or **Rewrite**(B,  $m$ ,  $A[n]$ )

## Text search considering Levenshtein distance

New stuff

$$d[i][k] = \text{minimum of } \left( \begin{array}{l} d[i-1][k] + 1, \\ \text{if } (i < m) \ d[i][k-1] + 1, \\ d[i-1][k-1] + (p[i] == t[k])?0:1 \end{array} \right)$$

// Delete p[i]

// Insert after p[i]

// leave or Rewrite p[i]

D	-	a	d	c	a	b	c	a	a	b	a	d	b	b	c	a
-	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
a	1	0	1	1	0	1	1	0	0	1	0	1	1	1	1	0
d	2	1	0	1	1	1	2	1	1	1	1	0	1	2	2	1
b	3	2	1	1	2	1	2	2	2	1	2	1	0	1	2	2
b	4	3	2	2	2	2	2	3	3	2	2	2	1	0	1	2
c	5	4	3	2	3	3	2	3	4	3	3	3	2	1	0	1
a	6	5	4	3	2	4	3	2	3	4	3	4	3	2	1	0

### Challenge

Value  $d[m][k]$  registers only the distance of a substring  $S$  in the text whose end is aligned with  $P$  and it is the minimum distance of all such substrings.

There is no reference in the DP table to the actual length  $S$  i.e. to its start position. To find string

$S = S_x = t[x], t[x+1], \dots, t[k]$ , where  $x \in \{k-m+1-d[m][k], \dots, k-m+1+d[m][k]\}$  we must consider all values of  $x$  and compute Levenshtein distance  $(S_x, P)$  for each  $x$  separately and choose  $x$  which attains minimum.

**Text search using finite automata brings in many possibilities regarding what can be effectively found:**

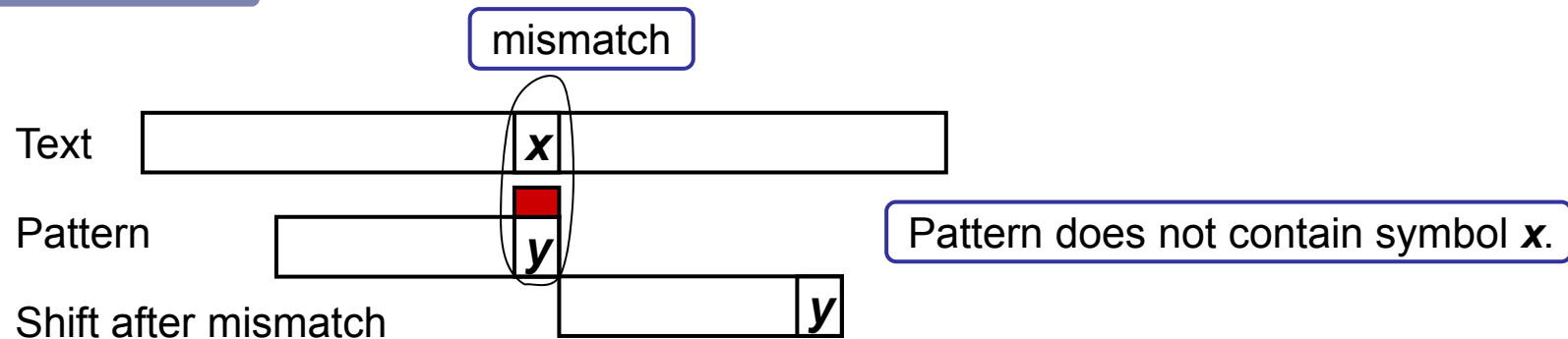
- A.** Any given exact pattern  $P$ . (e.g. *ababccabc*)
- B.** Any word of any language specified by a particular DFA or NFA.  
(Just add the loop labeled by the whole alphabet to the start state.)
- C.** Any string which represents some modification of the pattern  $P$ :  
A string within (or exactly at) a given Hamming distance from  $P$   
A string within (or exactly at) a given Levenshtein/edit distance from  $P$ .
- D.** Any of strings in a given (finite) dictionary.
- E.** Any word of any language described by a regular expression.
- F.** Any union, intersection, concatenation, iteration of any of cases A. - F.
- G.** Any string containing any of cases A. - F. as a subsequence.  
(Just add the loops labeled by the whole alphabet to all states.)

The idea:

Align the pattern with the text and start matching backwards **from the end** of the pattern.

When a mismatch occurs there is a chance that the pattern may be shifted forward by many positions and sometimes by the whole pattern length.

Ideal case



The longer is the pattern the more effective is the search.  
(The bigger the data the faster the algorithm, quite an unusual situation...)

Mismatch at the last position of the pattern.

### Bad Character Shift table (BCS)

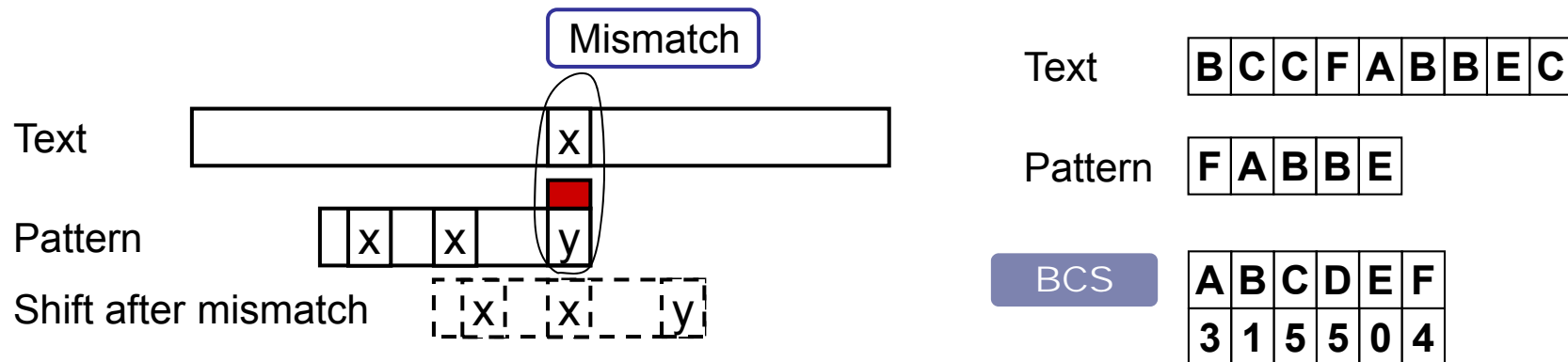
When the last symbol of pattern ( $y$ ) is mismatched with symbol  $x$  in the text shift the pattern to the right to match the first occurrence (from the end) of  $x$  in the pattern with  $x$  in the text.

When the pattern does not contain  $x$  shift it by its whole length.

BCS is indexed by all symbols of alphabet.

For each symbol in the pattern it contains the symbol's minimum distance from the end of the pattern. If the symbol is not in the pattern then the table entry is equal to the pattern length.

### Example



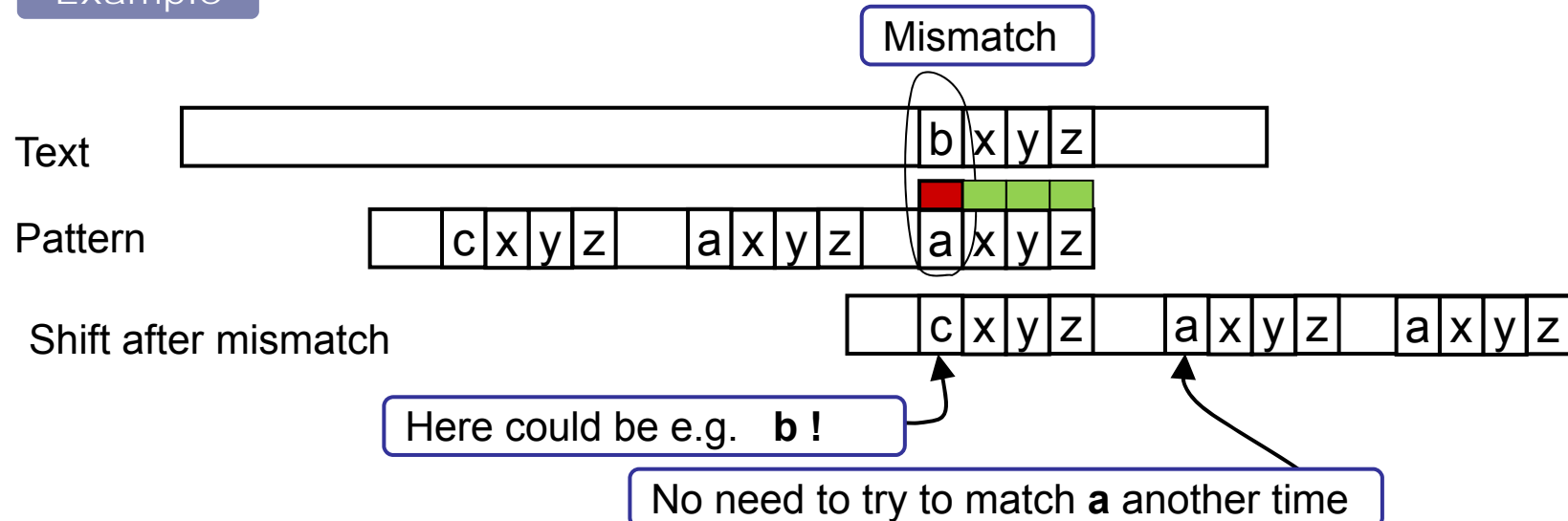


Mismatch after partial match at the end of the pattern.

When a suffix  $S$  of the pattern matches the text and the symbol  $x$  immediately preceding  $S$  mismatches the text then there are three cases:

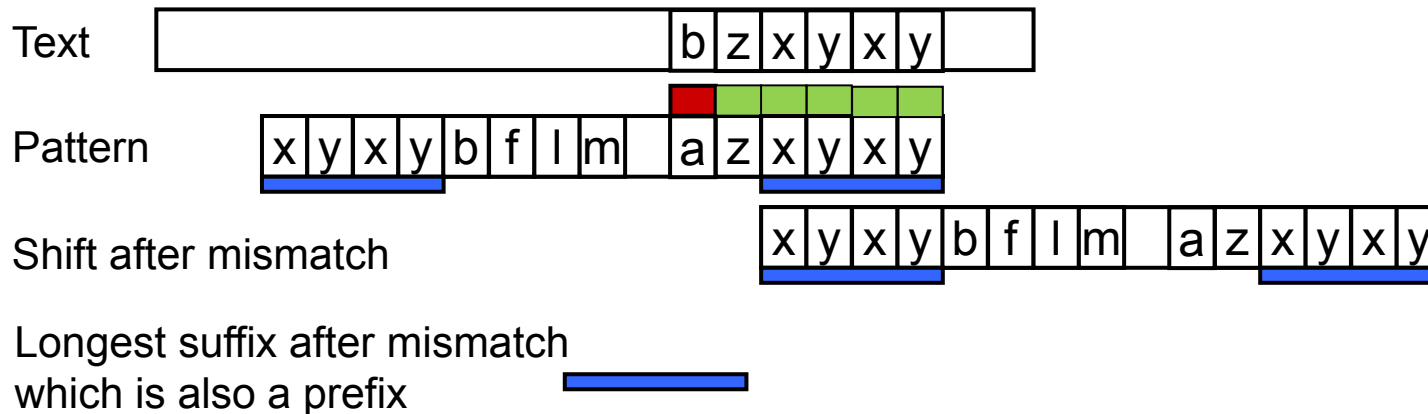
1. The suffix  $S$  occurs more times in the pattern and the other occurrence is not immediately preceded by  $x$ . In this case, shift the pattern so that the nearest described instance of  $S$  matches the text again at the same position. That is, shift the pattern by the distance between these occurrences of suffix  $S$ .

### Example



2. There is a suffix  $W$  whose length does not exceed the length of  $S$  and  $W$  is also a prefix of the pattern. Take the longest possible  $W$  and denote its occurrence at the beginning of the pattern by  $Q$ . Then shift the pattern by the distance between  $Q$  and  $W$ .

## Example



3. Neither case 1. nor case 2. happens. Then shift the pattern by its whole length.

Example is unnecessary

The shift can be calculated for all three cases :

Take suffix S as a separate string and align it with its original position in the pattern. Then keep shifting S to to the left until one of the cases 1., 2., 3. is detected (at least 3. must happen after some time).

Register the distance between the current and the original position of S.

Good Suffix Shift (GS) table contains the shift values for all suffixes S.

## Example

Pattern **A D B A C B A C B A**

Pattern length: 10

Positions indexed from 1,  
0 represents shift after  
complete match.

Apply case 2. after complete match

## GS

position	mismatches	suffix	shift
9	B	A	9
8	C	BA	6
7	A	CBA	9
6	B	ACBA	9
5	C	BACBA	3
4	A	CBACBA	9
3	B	ACBACBA	9
2	D	BACBACBA	9
1	A	DBACBACBA	10
0	-	ADBACBACBA	9

### Example

Pattern

POVALOVAL

BCS

_	A	E	K	L	N	O	P	S	T	V
9	1	9	9	4	9	3	8	9	9	2

GS

0	1	2	3	4	5	6	7	8	9
9	9	9	9	9	4	9	9	9	-
P	O	V	A	L	O	V	A	L	

### Search progress

BCS[P] == 8

GS[5] == 4

GS[6] == 9

Text

