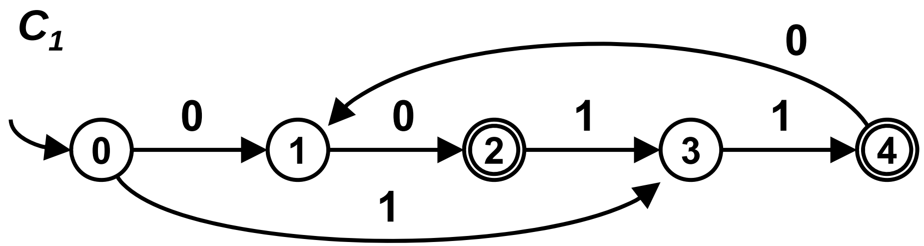
Text Search

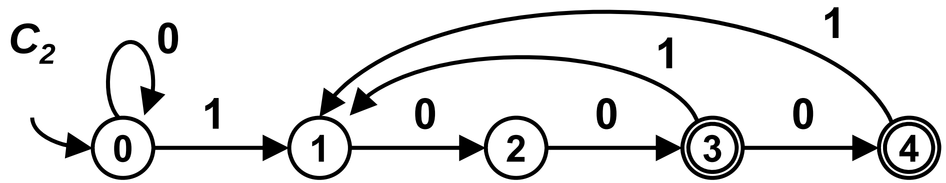
Marko Berezovský
Radek Mařík
PAL 2012

- Automata Examples
- Operations on Regular Languages
- Automata Reperesenting Operations on Regular Languages
- Hamming and Levenshtein Distance
- Approximate Text Search
- Automaton Bit Arrays Simulation

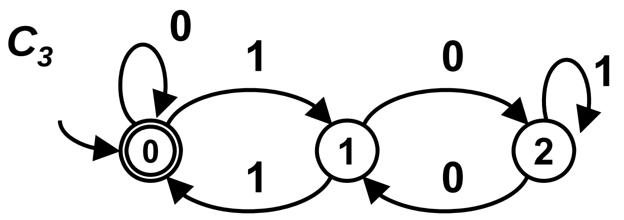
Automaton C_1 accepts union of sets
L_1 = {00, 0011, 001100, 00110011, 0011001100, ...}
$\cup$ {11, 1100, 110011, 11001100, 1100110011, ...}.



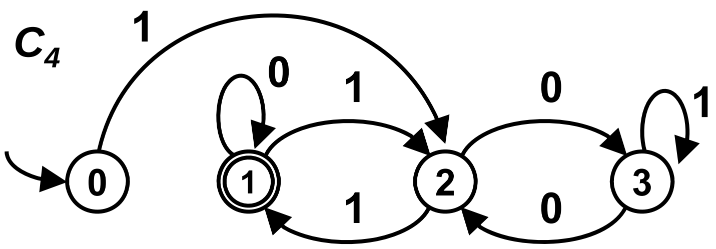
Automaton C_2 accepts language L_2 over Σ = {0, 1}, in each word of L_2 :
-- there is at least one symbol 1,
-- each symbol 1 is followed by exactly two or three symbols 0.



Automaton C_3 accepts all binary nonnegative integers divisible by 3, any number of leading zeros may be included.



Automaton C_4 accepts all binary positive integers divisible by 3, no leading zeros are allowed.



Operations on regular languages revisited

Let L_1 and L_2 be any languages. Then

$L_1 \cup L_2$ is **union** of L_1 and L_2. It is a set of all words which are in L_1 or L_2.

$L_1 \cap L_2$ is **intersection** of L_1 and L_2. It is a set of all words which are simultaneously in L_1 and L_2.

$L_1.L_2$ is **concatenation** of L_1 and L_2. It is a set of all words w for which holds $w = w_1w_2$ (concatenation of words w_1 and w_2), where $w_1 \in L_1$ and $w_2 \in L_2$.

L_1^* is Kleene **star** or Kleene **closure** or **iteration** of language L_1. It is a set of all words which are concatenations of any number (incl. zero) of any words of L_1 in any order.

Closure

Whenever L_1 and L_2 are regular languages then $L_1 \cup L_2$, $L_1 \cap L_2$, $L_1.L_2$, L_1^* are regular languages too.

Automata support

When L_1 is regular language accepted by automaton A_1 and L_2 is regular language accepted by automaton A_2 then there also are automata A_3, A_4, A_5, A_6, which accept $L_1 \cup L_2$, $L_1 \cap L_2$, $L_1.L_2$, L_1^*, respectively.

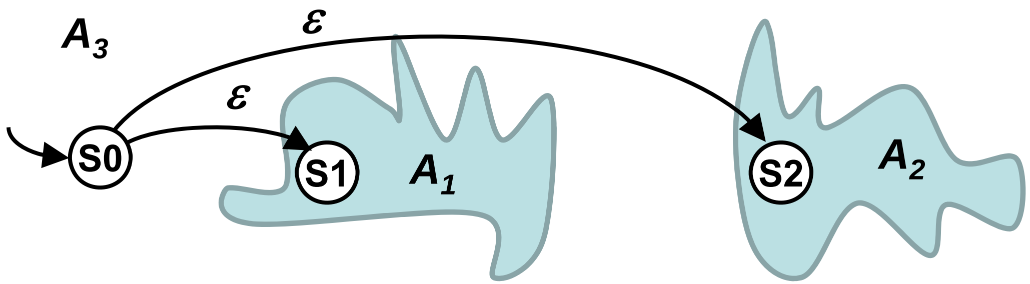
Automaton A_3 accepting union of two regular languages L_1 , L_2 accpted by automata A_1, A_2 respectively.

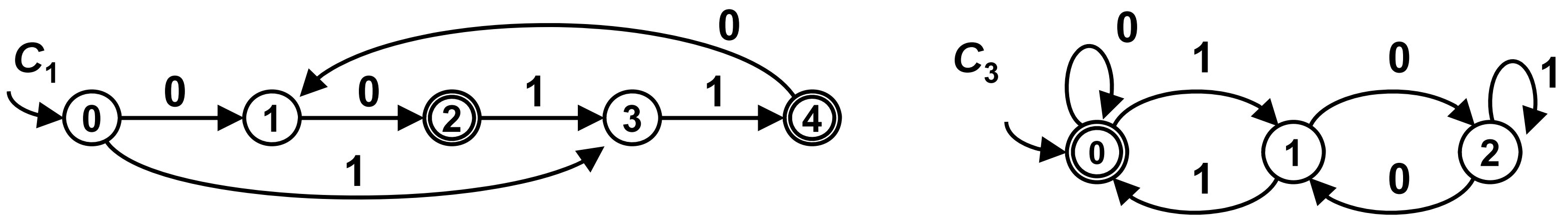
Automaton A_3 is constructed using A_1 and A_2:

Do not change A_1 and A_2.
Create new aditional start state S_0, add ε - transitions from S_0 to start states S_1 and S_2 of A_1 and A_2 respectively.
Define set of final states of A_3 as union of final states of A_1 and A_2.

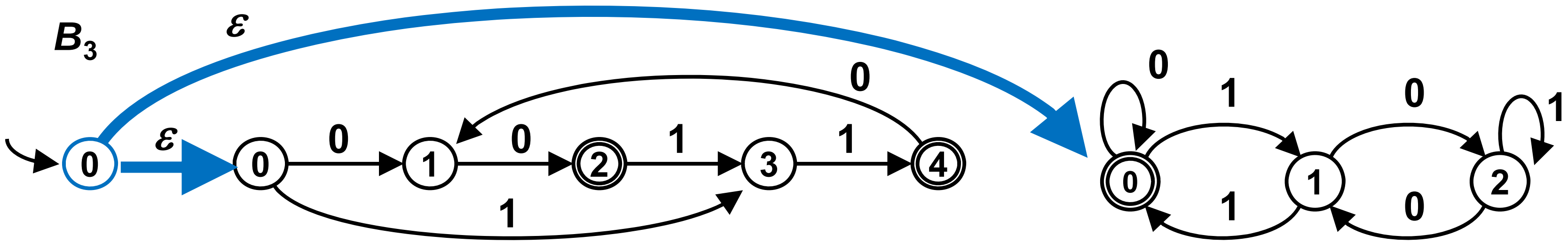
Scheme

.





Automaton B_3 accepts any word from sets
{00, 0011, 001100, 00110011, 0011001100, ...}
{11, 1100, 110011, 11001100, 1100110011, ...}
and also any binary nonnegative integer divisible by 3
with any number of leading zeros



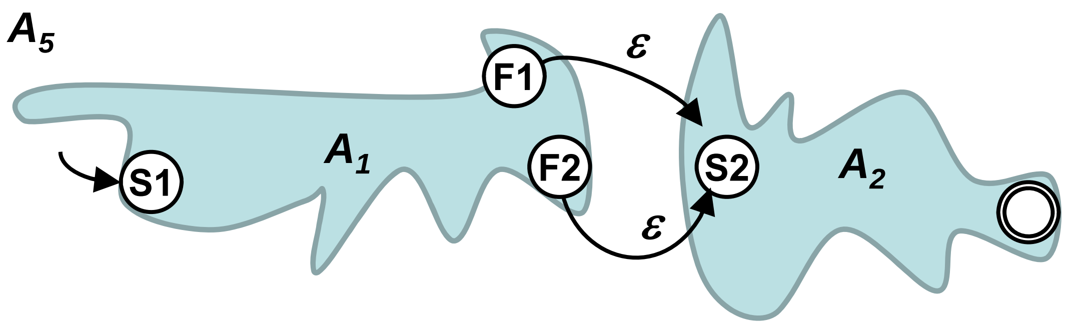
Automaton A_5 accepting concatenation of two regular languages L_1 , L_2 accepted by automata A_1, A_2 respectively.

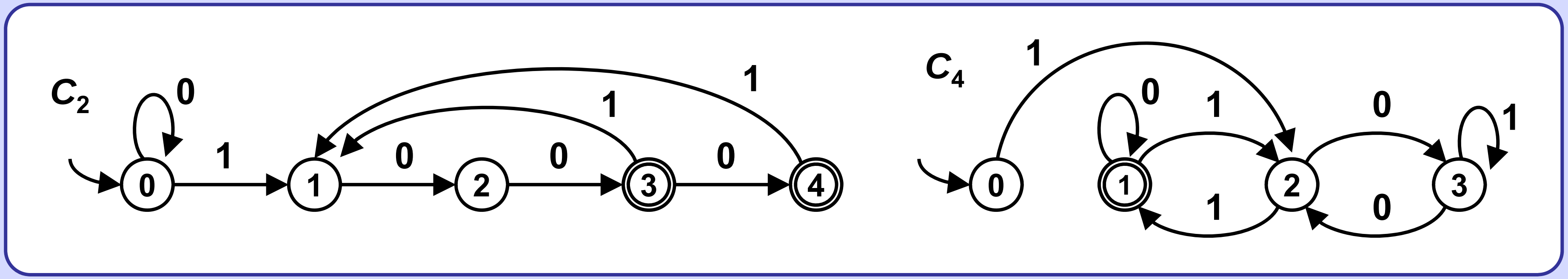
Automaton A_5 is constructed using A_1 and A_2:

Do not change A_1 and A_2.
Add ε - transitions from each final state F_k of A_1 to start state S_2 of A_2.
Define start state of A_5 equal to start state of A_1.
Define set of final states of A_5 as equal to those of A_2.

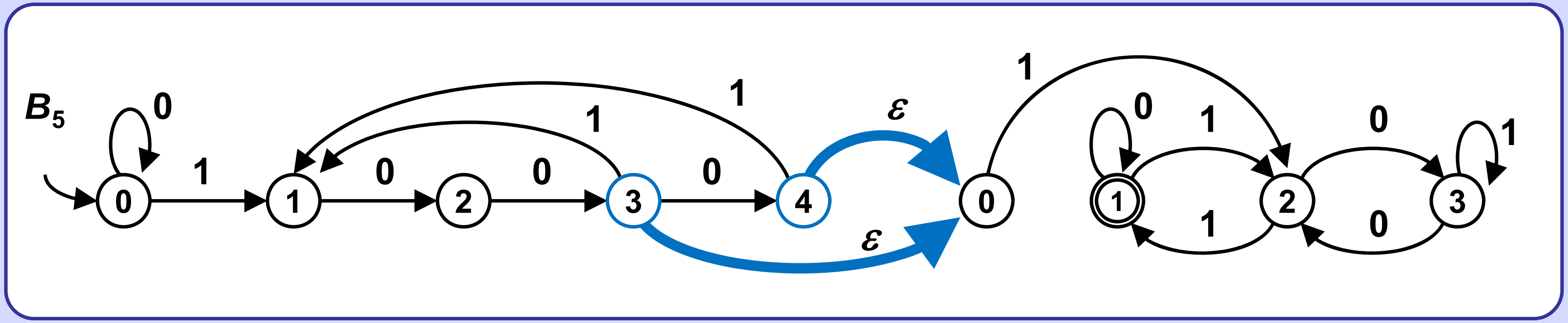
Scheme

.





Automaton B_5 accepts any word over {0, 1} which can be split into two consecutive words w1 and w2, where
word w1 is described by regular expression 0*(100+1000)(100+1000)* ,
word w2 represents binary positive integer divisible by 3 w/o leading 0's.



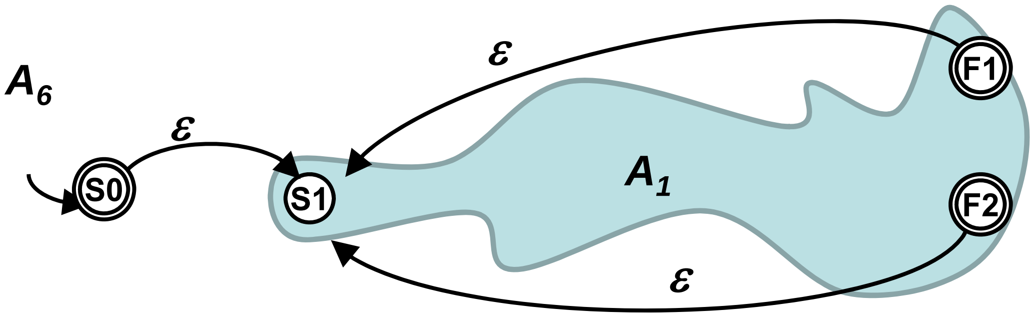
Automaton A_6 accepting iteration of language L_1 accepted by automaton A_1.

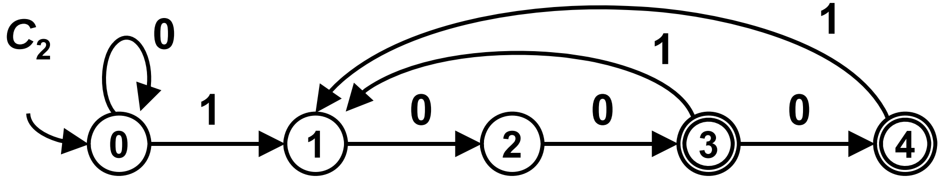
Automaton A_6 is constructed using A_1:

Do not change A_1.
Create new aditional start state S_0 and add ε - transition from S_0 to start state S_1 of A_1
Add ε - transitions from all final states F_k of A_1 to state S_1.
Define start state of A_6 to be S_0.
Define set of final states of A_6 as union of final states F_k and S_0.

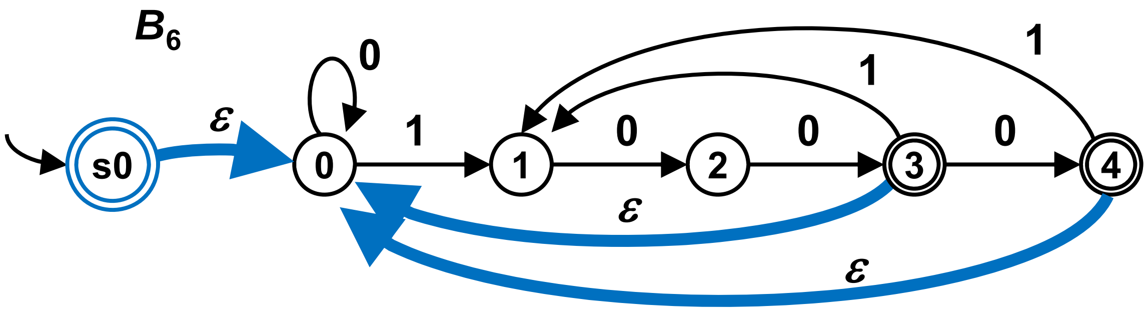
Scheme

.





Automaton B_6 accepts any word created by concatenation and repetition of any words of C_2 including empty word.



Maybe you can find some more telling informal description of the corresponding language?

Automaton A_4 accepting intersection of two regular languages L_1 , L_2 accepted by automata A_1, A_2 respectively.

Automaton A_4 is constructed using A_1 and A_2:

Create Cartesian product $Q_1 \times Q_2$, where Q_1, Q_2 are sets of states of A_1, A_2.

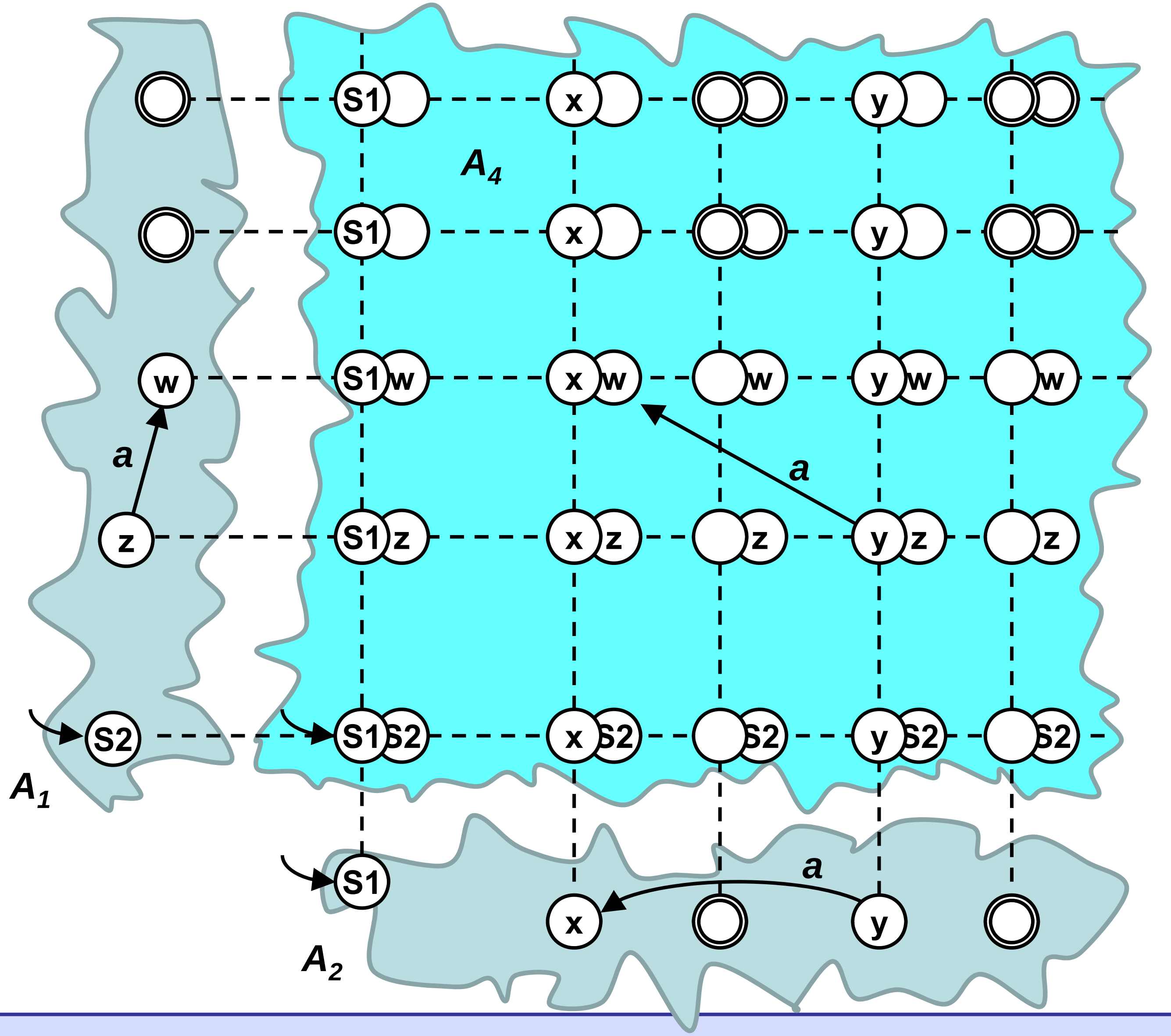
Each state of A_4 will be an ordered pair of states of A_1, A_2.

State (S_1, S_2) will be start state of A_4, when S_1,S_2 are start states of A_1, A_2.

Final states of A_4 will be just those pairs (F, G),
where F is a final state of A_1 and G is a final state of A_2.

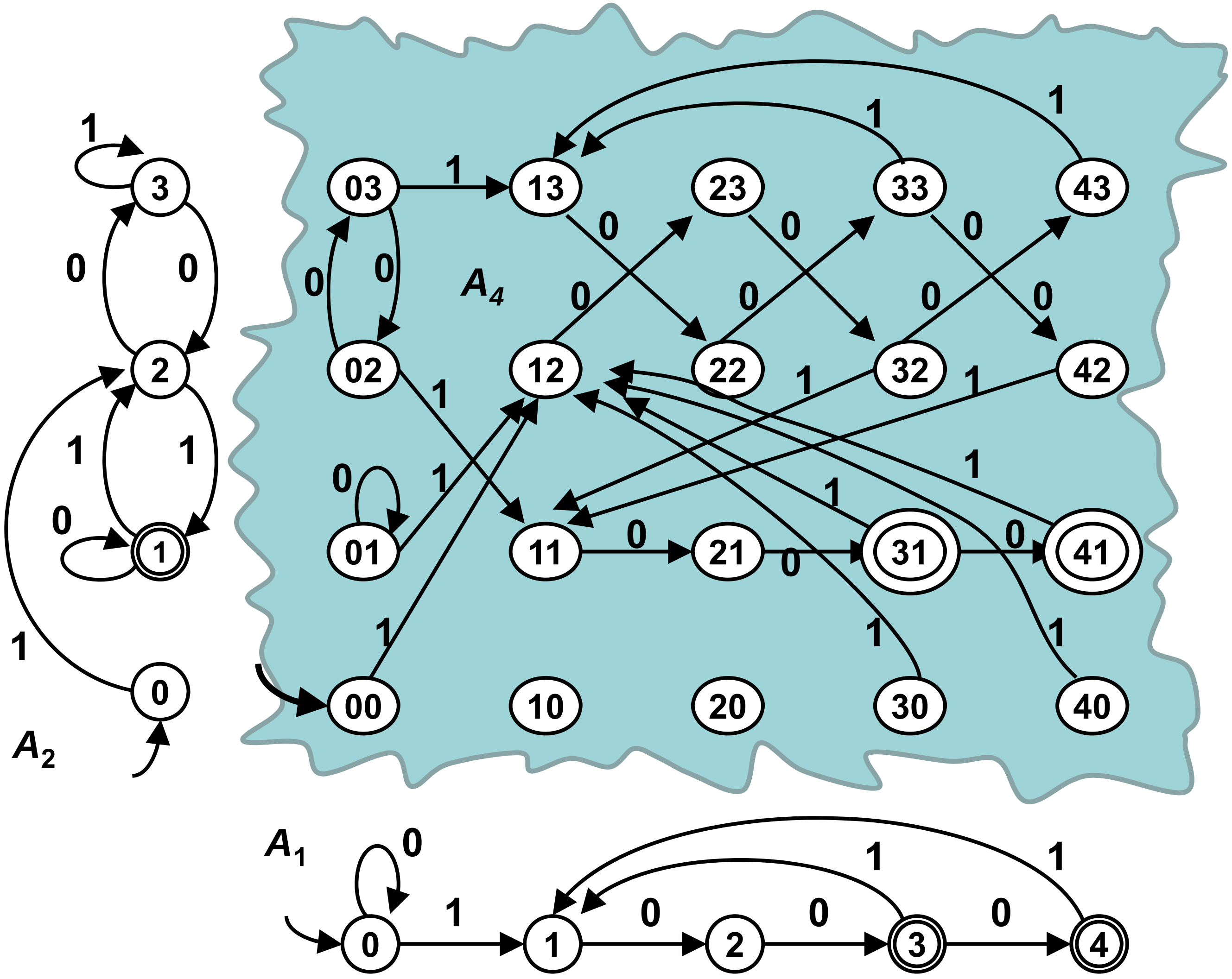
Create transition from state (p_1, p_2) to (q_1, q_2) in A_4 labeled by symbol x
if and only if
there is a transition $p_1 \to q_1$ labeled by x in A_1 and also
there is a transition $p_2 \to q_2$ labeled by x in A_2.

Scheme of an automaton A_4 accepting the intersection of two regular languages L_1, L_2 accepted by automata A_1, A_2 respectively.



Automaton A_4
accepting
binary integers
divisible by 3 (C_4)
in which
each symbol 1
is followed by
exactly two
or three
symbols 0 (C_2).

It's quite a nice
one, isn't it?



Hamming distance

Hamming distance of two strings is equal to *k* ($k \leq 0$),
whenever *k* is the minimal number of rewrite operations which when applied on one of the strings produce the other string.
Rewrite operation rewrites one symbol of the alphabet
by some other symbol of the alphabet.
Symbols cannot be deleted or inserted.
Hamming distance is defined only for pairs of strings of equal length.

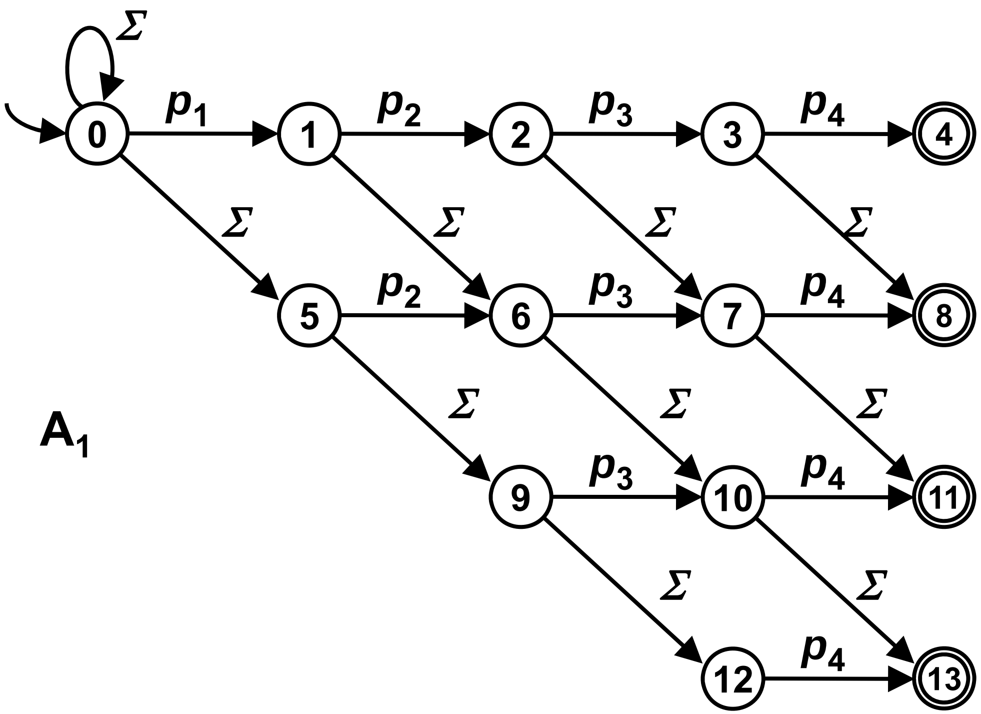
Informally: Align the strings and count the number of mismatches of corresponding symbols.

Learn some Czech

```
l o k o m o t i v a
v y k o l e j i l a    distance = 6

m a l é _ p i v o
v e l k ý _ v ů z      distance = 8
```

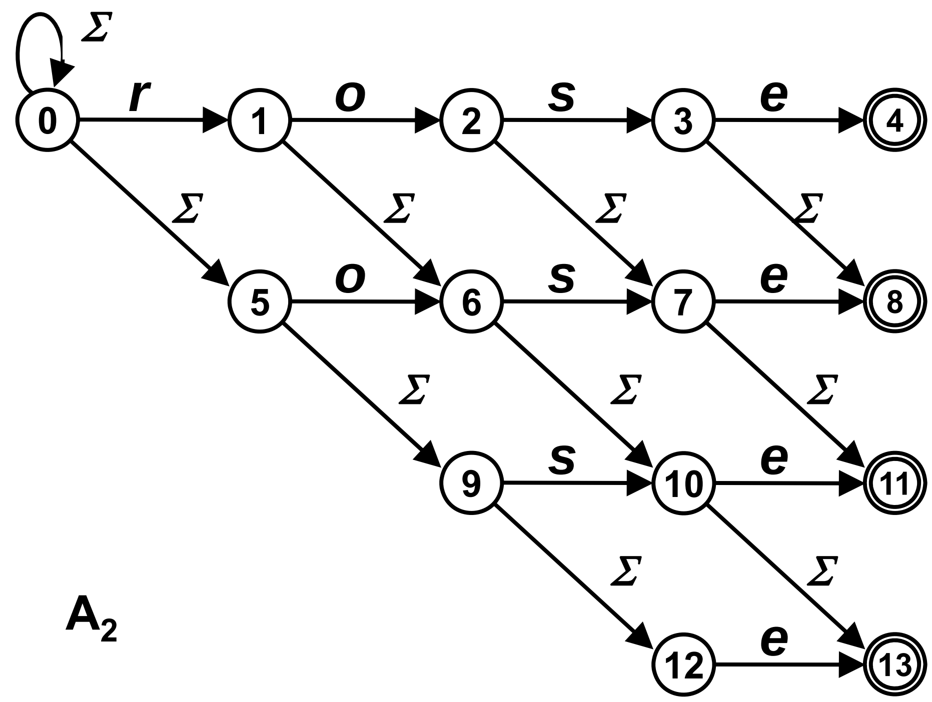
Automaton A_1 for aproximate pattern matching. It detects all occurences of substrings which Hamming distance form the pattern $p_1p_2p_3p_4$ is less or equal to 3.



Automaton A_2 for aproximate pattern matching. It detects all occurences of substrings which Hamming distance form the pattern 'rose' is less or equal to 3.

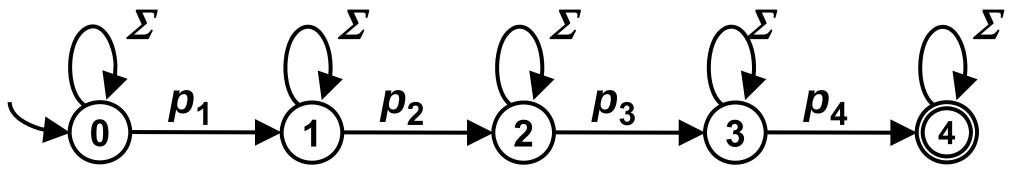
Automaton A_2 detects among others also the words:

rose (distance = 0)
dose (distance = 1)
rest (distance = 2)
list (distance = 3)
and more...



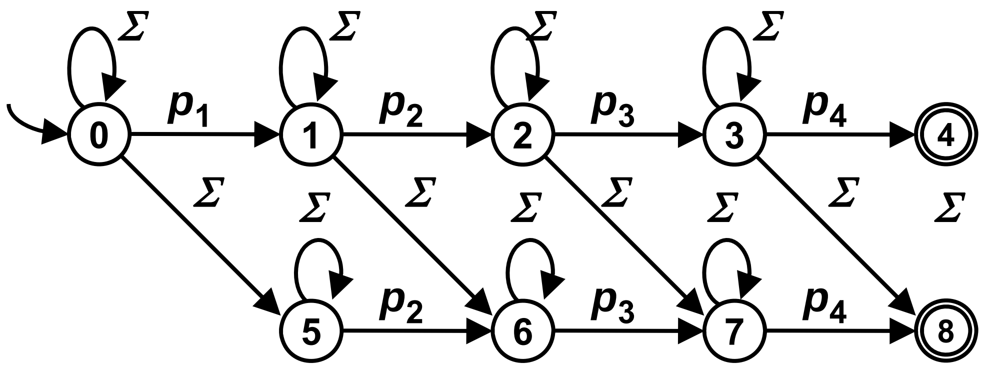
Example

NFA accepting any word with subsequence $p_1p_2p_3p_4$ anywhere in it.

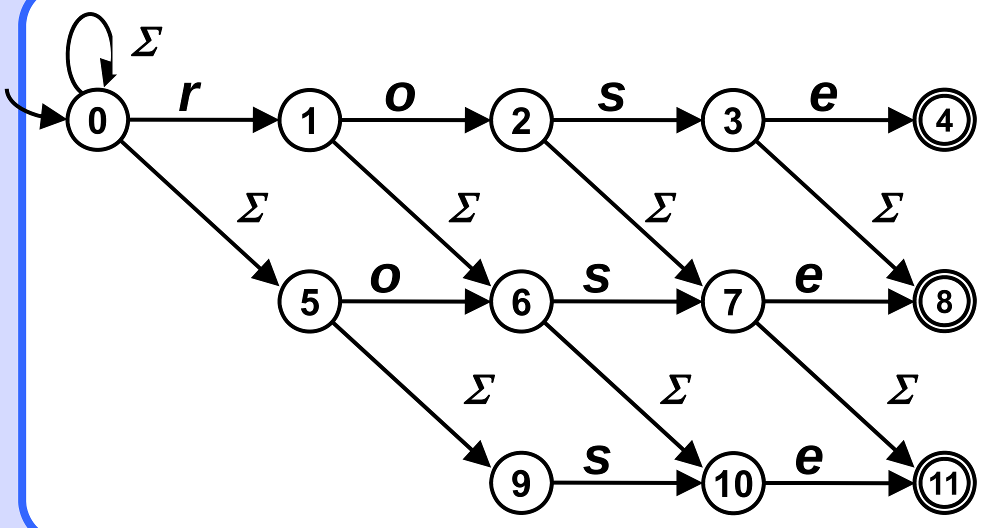


Example

NFA accepting any word with subsequence $p_1p_2p_3p_4$ anywhere in it, one symbol in the sequence may be altered.



Alternatively: NFA accepting any word containing a subsequence Q which Hamming distance from $p_1p_2p_3p_4$ is at most 1.



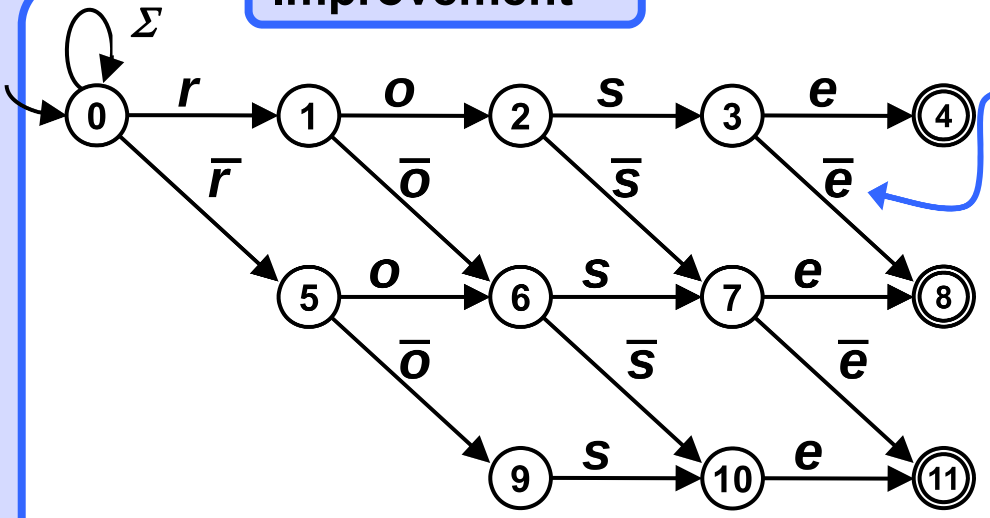
Hamming distance of the found pattern Q from pattern P = rose cannot be deduced from the particular end state.

E.g.: "rope":

r - 1 - **o** - 2 - **p** - 7 - **e** - 8.

r - 5 - **o** - 6 - **p** - 10 - **e** - 11.

Improvement



Notation: $\overline{x} = \Sigma - \{x\}$

means: Complement of x in Σ.

Hamming distance from the pattern P = rose to the found pattern Q corresponds exactly to the end state.

Levenshtein distance

Levenshtein distance of two strings A and B
is such minimal *k* (*k* ≥ 0), that we can change A to o B or B to A
by applying exactly *k* edit operations on one of them.
The edit operations are Remove, Insert or Rewrite any symbol of the alphabet anywhere in the string. (Rewrite is also called Substitution.)

Levenshtein distance is defined for any two strings over a given alphabet.

```
B R U X E L L E S          Delete X.
B E T E L G E U S E        Rewrite R->E, U->T, L->G.
                           Insert U, E.
Distance = 6
```

Note

Although the distance is defined unambiguously (prove!), the particular edit operations transforming one string to another may vary (find an example).

Calculating Levenshtein distance

Apply a simple Dynamic Programming approach.

Let A = a[1].a[2]. ... .a[n] = A[1..n], B = b[1].b[2]. ... .b[m] = b[1..m], n, m ≥ 0.

Dist(A, B) = |m − n| **if n = 0 or m = 0**

Dist(A, B) = 1+ min (Dist(A[1..n − 1], B[1..m]),
Dist(A[1..n], B[1..m −1]),
Dist(A[1..n −1], B[1..m −1])) **if n > 0 and m > 0 and A[n] ≠ B[m]**

Dist(A, B) = Dist(A[1..n − 1], B[1..m−1]]) **if n > 0 and m > 0 and A[n] = B[m]**

Calculation corresponds to ... Operation

Dist(A[1..n −1], B[1..m]), ... **Insert**(A, n −1, B[m]) or **Delete**(B, m)
Dist(A[1..n], B[1..m −1]), ... **Insert**(B, m −1, A[n]) or **Delete**(A, n)
Dist(A[1..n −1], B[1..m −1]) ... **Rewrite**(A, n, B[m]) or **Rewrite**(B, m, A[n])

Dist("BETELGEUSE","BRUXELLES") = 6

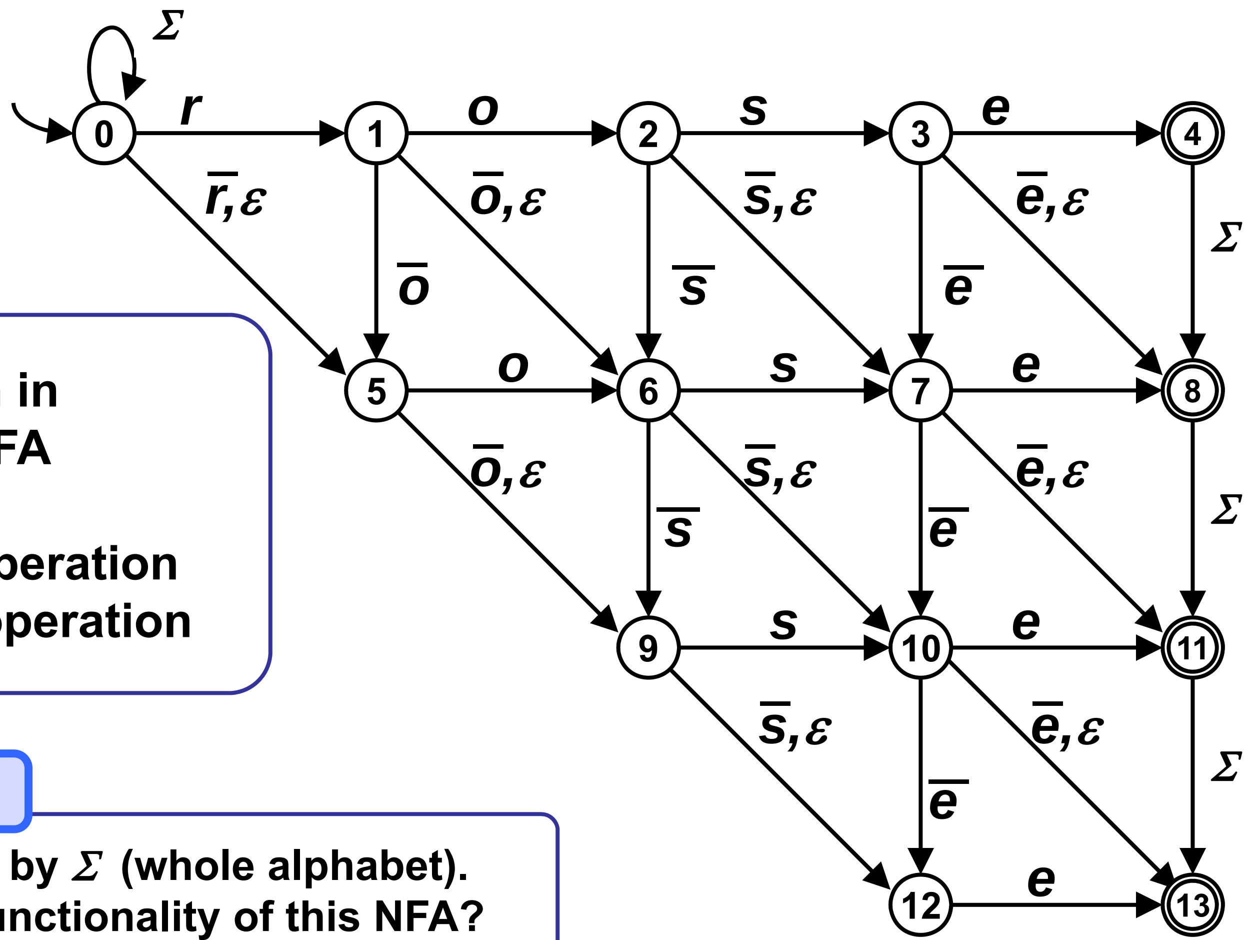
		B	E	T	E	L	G	E	U	S	E
	0	1	2	3	4	5	6	7	8	9	10
B	1	0	1	2	3	4	5	6	7	8	9
R	2	1	1	2	3	4	5	6	7	8	9
U	3	2	2	2	3	4	5	6	6	7	8
X	4	3	3	3	3	4	5	6	7	7	8
E	5	4	3	4	3	4	5	5	6	7	7
L	6	5	4	4	4	3	4	5	6	7	8
L	7	6	5	5	5	4	4	5	6	7	8
E	8	7	6	6	5	5	5	4	5	6	7
S	9	8	7	7	6	6	6	5	5	5	6

Warning

Some top of Google search links to "compute Levenshtein distance" are wrong, typically they mistakenly init 0-th row/column with 0's. Wikipedia code is correct.

NFA searches in text for a string within Levenshtein distance 3 from the pattern "rose".

Note the ε–transitions.



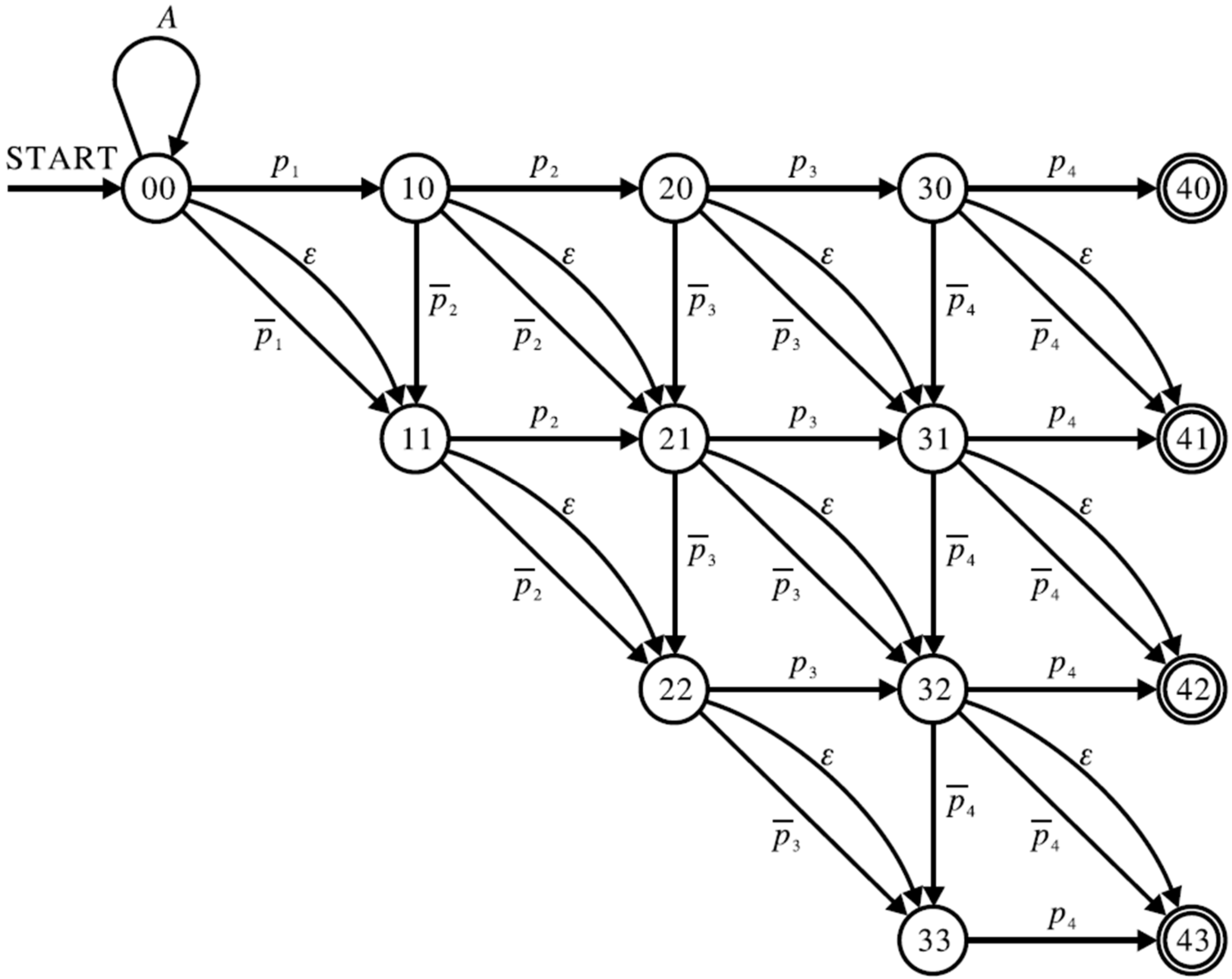
More transitions than in Hamming distance NFA

vertical ... Insert operation
epsilon ... Delete operation

Self-check question

Label verical transitions by Σ (whole alphabet).
How will it change the functionality of this NFA?

Another example



Challenge?

There is a kind of discrepancy, seemingly:

1. Levenshtein distance of strings A and B can be calculated using the DP approach in $O(m \cdot n)$ time.
2. Determining the Levenshtein distance between A and B can be done also by treating A as text and B as a pattern (or vice versa) and applying the appropriate NFA on the text, which would run in just $O(\min(m, n))$ time.

Why bother to do calculations with DP?

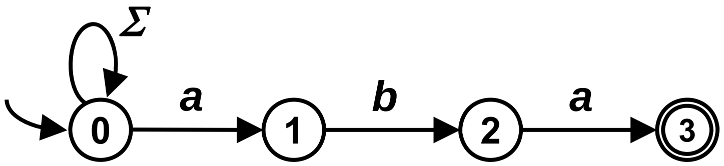
Bit representation of NFA

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

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

For bit vector **F** of final states holds **F[j] == 1** $\Leftrightarrow$ $\mathbf{q_j} \in \mathbf{F}_A$

Example



A

	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
		0	0	0
		1	0	0
i=3		0	0	0
		0	0	0
		0	0	0
		0	0	0

Bit representation of automaton A.

F

0	0	0	1

`T[1,2][4] == 0`

starting configuration

time →

A

sets of states represented by bit arrays during computation

text symbols:	-	*a*	*c*	*c*	*a*	*b*	*c*	*a*	*a*	*b*	*a*
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	1
		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
		0	0	0
		1	0	0
i=3		0	0	0
		0	0	0
		0	0	0
		0	0	0

example

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

1		**0**		**1**
0	**OR**	**0**	**=**	**0**
0		**1**		**1**
0		**0**		**0**

F

0	0	0	1

Bit Representation

Simulation of work of NFA without ε–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] or T[j][t[i]];
  i++;
}
```