Text Search

Marko Berezovský
Radek Mařík
PAL 2012

Nondeterministic Finite Automata

Transformation NFA to DFA and Simulation of NFA

Text Search Using Automata

Power of Nondeterministic Approach

Regular Expression Search

Dealing with ε–transitions

Languages, grammars, automata

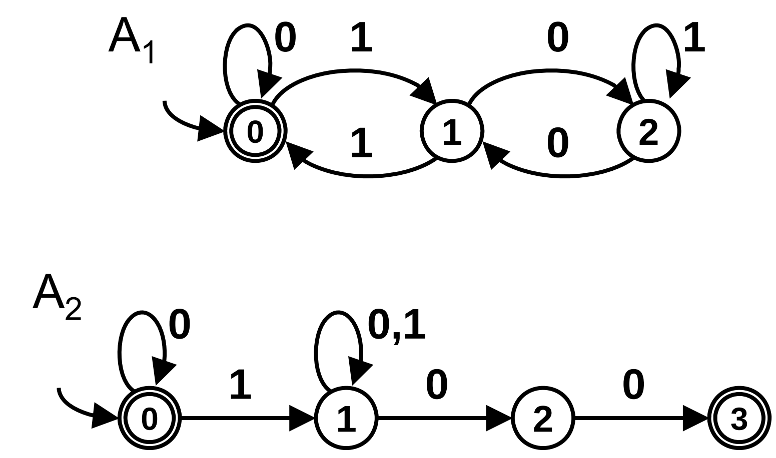
Czech instant sources:
[1] Prof. Marie Demlová: **A4B01JAG**
http://math.feld.cvut.cz/demlova/teaching/jag/predn_jag.html
Pages 1-27, in PAL, you may wish to skip: Proofs, chapters 2.4, 2.6, 2.8.

[2] I. Černá, M. Křetínský, A. Kučera: **Automaty a formální jazyky I**
http://is.muni.cz/do/1499/el/estud/fi/js06/ib005/Formalni_jazyky_a_automaty_I.pdf
Chapters 1 and 2, skip same parts as in [1].

English sources:
[3] B. Melichar, J. Holub, T. Polcar: **Text Search Algorithms**
http://cw.felk.cvut.cz/lib/exe/fetch.php/courses/a4m33pal/melichar-tsa-lectures-1.pdf
Chapters 1.4 and 1.5, it is probably too short, there is nothing to skip.

[4] J. E. Hopcroft, R. Motwani, J. D. Ullman: **Introduction to Automata Theory**
folow the link at http://cw.felk.cvut.cz/doku.php/courses/a4m33pal/literatura_odkazy
Chapters 1., 2., 3., there is a lot to skip, consult the teacher preferably.

For more references see PAL links page
http://cw.felk.cvut.cz/doku.php/courses/a4m33pal/literatura_odkazy



Deterministic Finite Automaton (DFA)
Nondeterministic Finite Automaton (NFA)

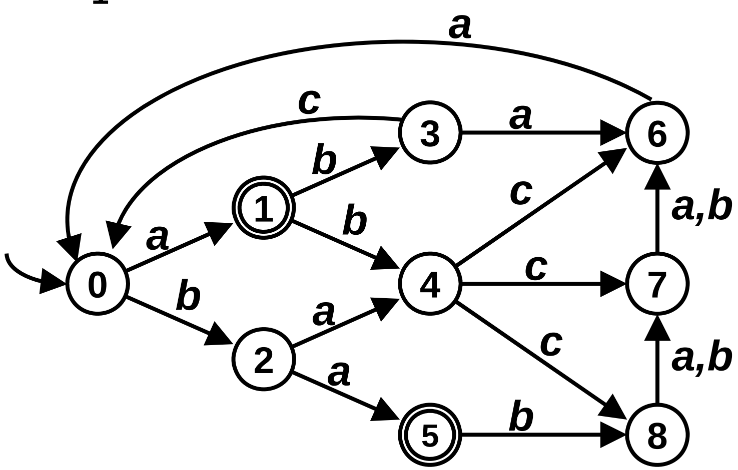
Both DFA nd NFA consist of:
- Finite input alphabet Σ.
- Finite set of internal states Q.
- One starting state $q_0 \in Q$.
- Nonempty set of accept states $F \subseteq Q$.
- Transition function δ.

DFA transition function is $\delta: Q \times \Sigma \rightarrow Q$.
DFA is always in one of its states.
DFA transits from current state to another state depending on the current input symbol.

NFA transition function is $\delta:\ \mathrm{Q} \times \Sigma \rightarrow P(Q)$ ($P(Q)$ is powerset of Q, set of all subsets of Q)
NFA is always (simultaneously) in a set of any number of its states.
NFA transits from a state to a set of states depending on the current input symbol.

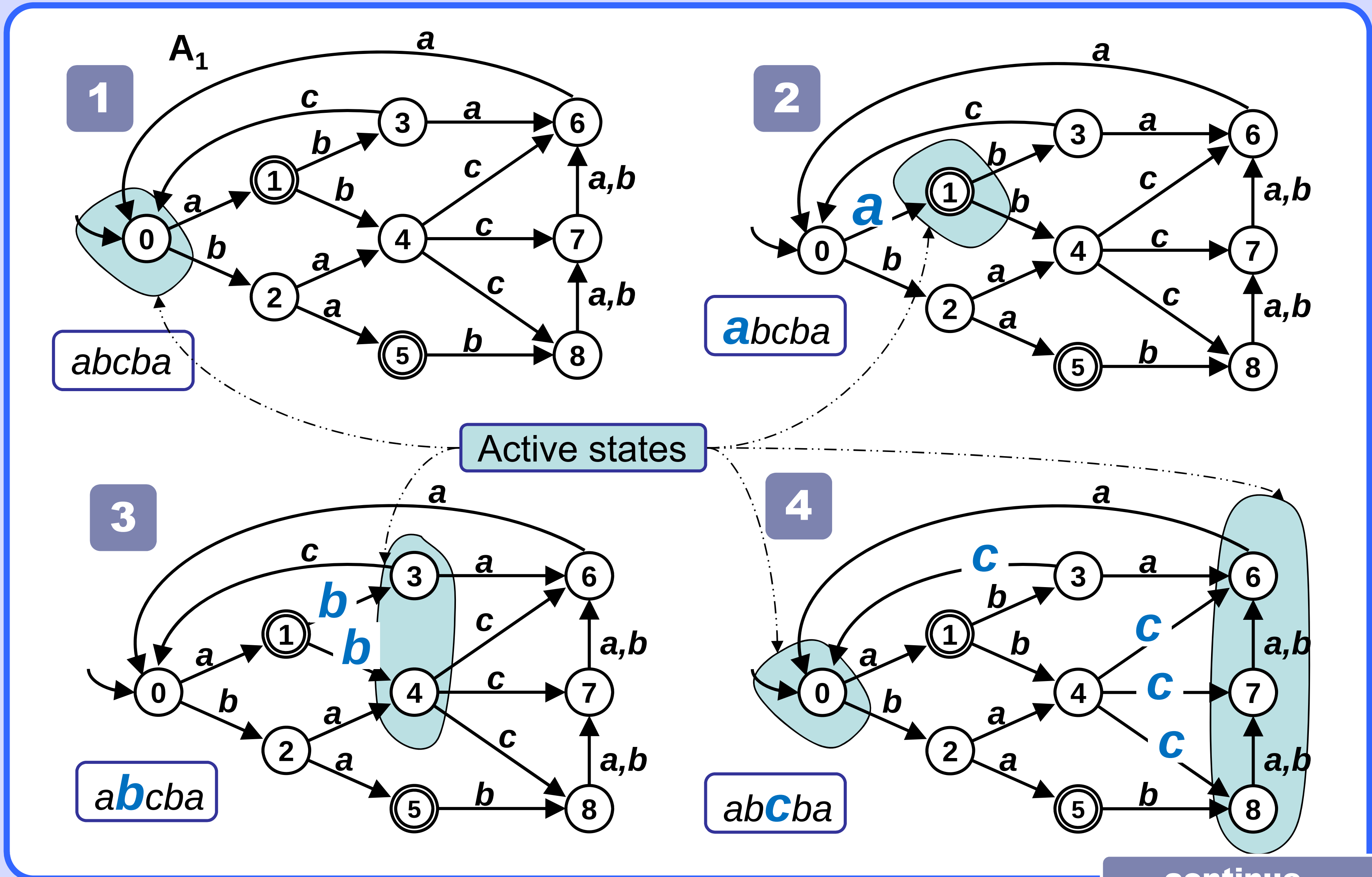
NFA $\mathbf{A_1}$, its transition diagram and its transition table

$\mathbf{A_1}$



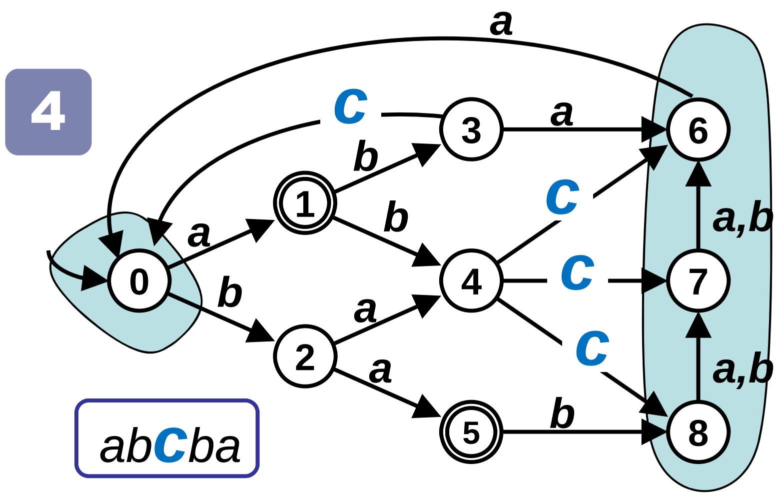
	a	b	c	
0	1	2		
1		3,4		F
2	4,5			
3	6		0	
4			6,7,8	
5		8		F
6	0			
7	6	6		
8	7	7		

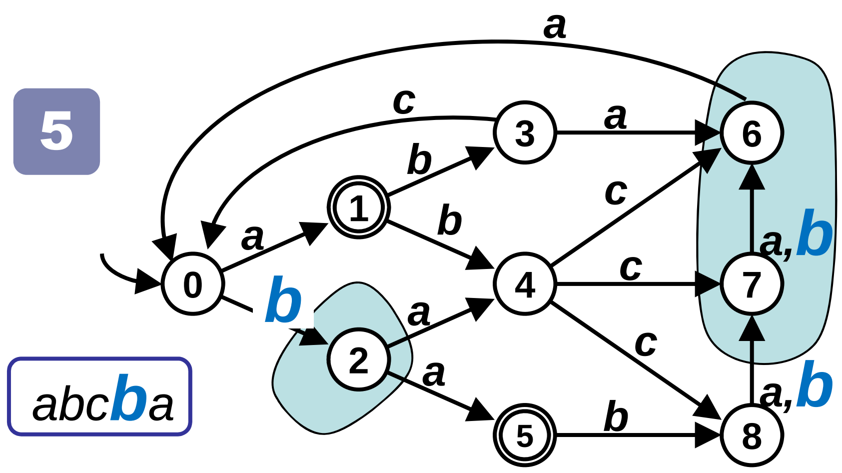
NFA **A_1** processing input word *abcba*

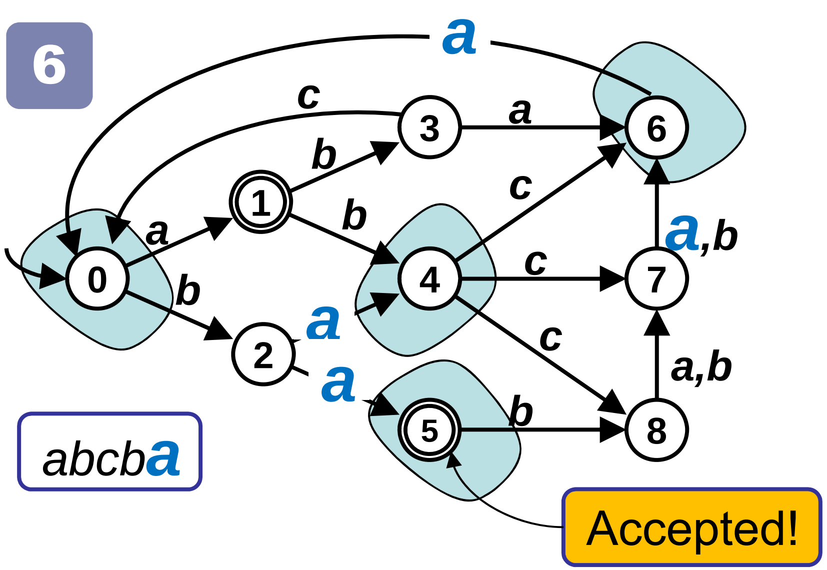


continue...

...continued







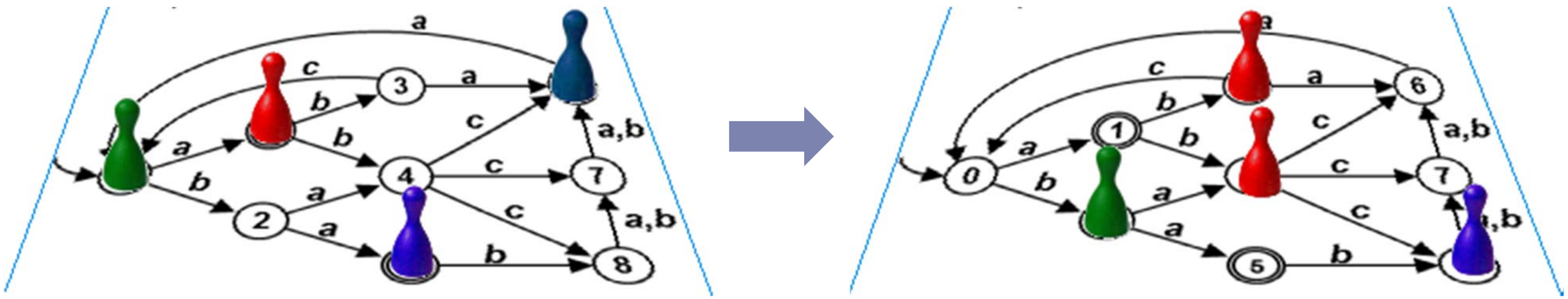
NFA $\mathbf{A_1}$ has processed word *abcba* and went through read symbols and respective sets(!) of states

$\{0\} \to a \to \{1\} \to b \to \{3, 4\} \to c \to$
$\to \{0, 6, 7, 8\} \to b \to \{2, 6, 7\} \to a \to$
$\to \{0, 4, 5, 6\}$.

NFA simulation without transform to DFA

Each of current states is occupied by one token.
Read an input symbol and move tokens accordingly.
If token has more possibilities it will split into two or more tokens,
if token has no possibility it will leave the board, uhm, the transition diagram.

Read **b** from input



NFA simulation without transform to DFA

Idea:
Register all states to which you have just arrived. In the next step read the input symbol x and move SIMULTANEOUSLY to ALL states to which you can get from ALL current states along transitions marked by x.

Input: NFA , text in array t

```
SetOfStates S = {q0}, S_tmp;

i = 1;
while ((i <= t.length) && (!S.isEmpty())) {
  S_tmp = Set.emptySet();
  for (q in S)      // for each state in S
    S_tmp.union(delta(q, t[i]));
  S = S_tmp;
  i++;
}
return S.containsFinalState();  // true or false
```

Generating DFA $\mathbf{A_2}$ equivalent to NFA $\mathbf{A_1}$ using transition tables

Data
Each state of DFA is a subset of states of NFA
Start state of DFA is a one element set containing just start state of NFA.
A state of DFA is accept state iff it contains at least one accept state of NFA.

Construction
Create start state of DFA and corresponding first line of its transition table (TT).

```
For each state Q of DFA not yet processed do {
   Decompose Q into its constituent states Q1, ..., Qk of NFA
    For each symbol x of alphabet do {
     S = union of all references in NFA table at positions [Q1] [x], ... [Qk][x]
     if (S is not among states of DFA yet)
           add S to states of DFA and add corresponding line to TT of DFA
   }
  Mark Q as processed
}
```

// Remember, empty set is also a set ot states, it can be easily a state of DFA

Generating DFA A_2 equivalent to NFA A_1

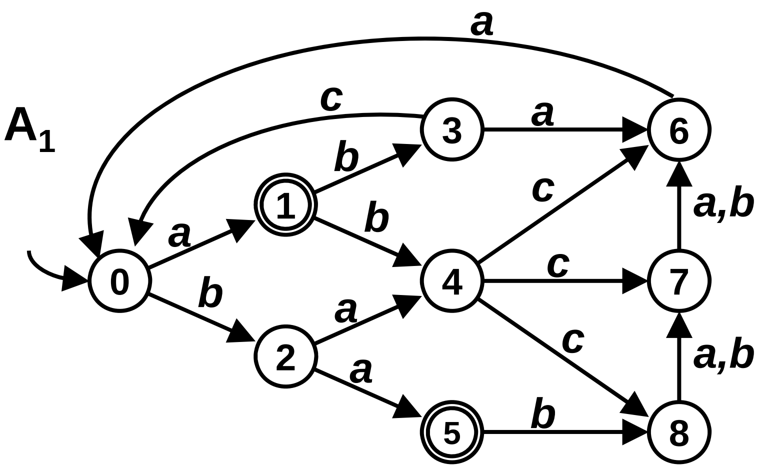
	a	*b*	*c*	
0	1	2		
1		3,4		F
2	4,5			
3	6		0	
4			6,7,8	
5		8		F
6	0			
7	6	6		
8	7	7		

Copy start state

A_2

	a	*b*	*c*
0	1	2	
...			

A_1

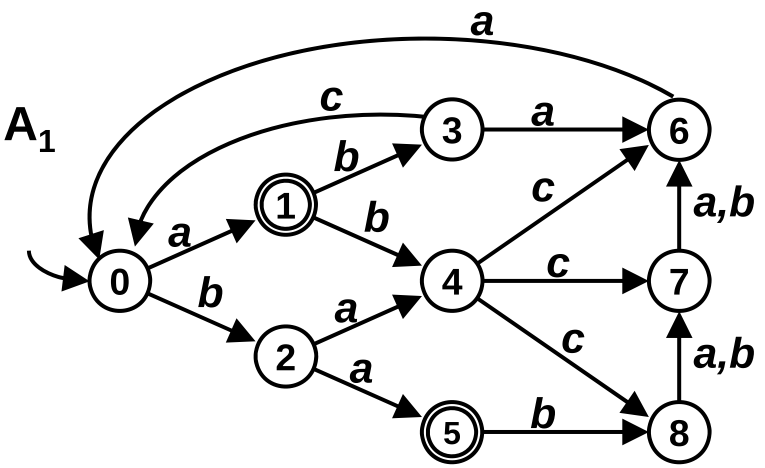


continue...

Generating DFA $\mathbf{A_2}$ equivalent to NFA $\mathbf{A_1}$

	a	*b*	*c*	
0	1	2		
1		3,4		F
2	4,5			
3	6		0	
4			6,7,8	
5		8		F
6	0			
7	6	6		
8	7	7		

A_1



Add new state(s)

A_2

	a	*b*	*c*	
0	1	2		
1		34		F
...				

continue...

Generating DFA $\mathbf{A_2}$ equivalent to NFA $\mathbf{A_1}$

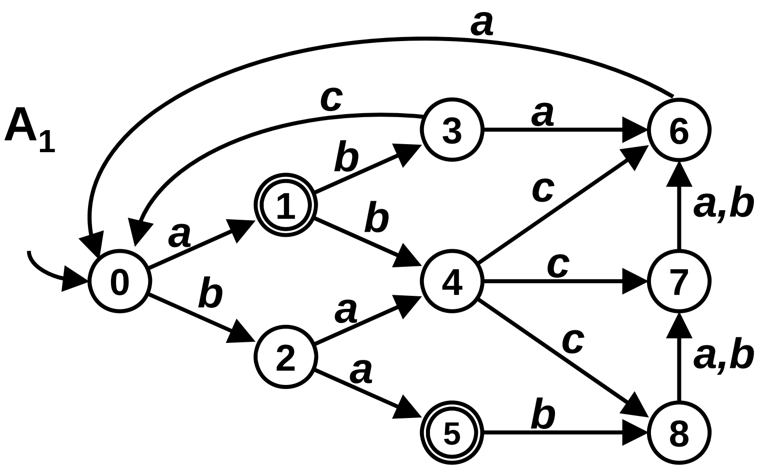
	a	*b*	*c*	
0	1	2		
1		3,4		F
2	4,5			
3	6		0	
4			6,7,8	
5		8		F
6	0			
7	6	6		
8	7	7		

Add new state(s)

A_2

	a	*b*	*c*	
0	1	$\boxed{2}$		
1		34		F
2	45			
...				

A_1



continue...

Generating DFA A_2 equivalent to NFA A_1

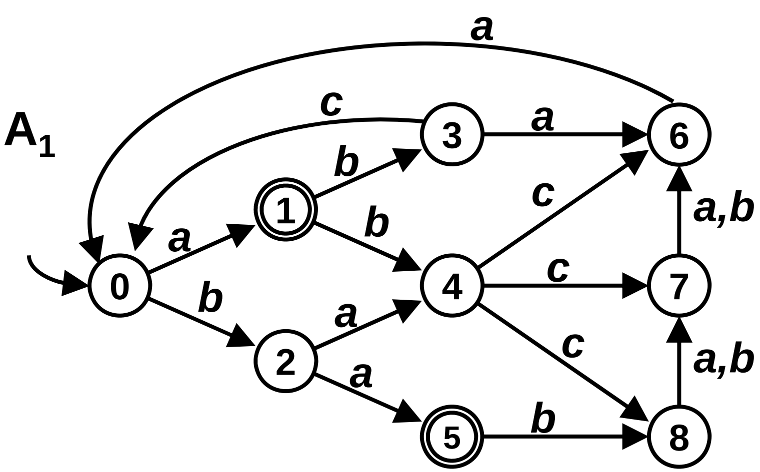
	a	b	c	
0	1	2		
1		3,4		F
2	4,5			
3	6		0	
4			6,7,8	
5		8		F
6	0			
7	6	6		
8	7	7		

Add new state(s)

A_2

	a	b	c	
0	1	2		
1		34		F
2	45			
34	6		0678	
...				

A_1

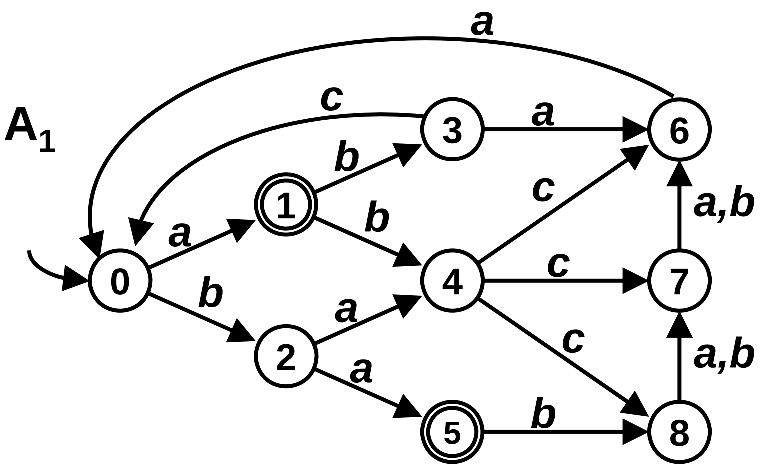


Note:
In the example we add
the empty set to the table
at the very end of the process
just to keep the table uncluttered....

continue...

Generating DFA $\mathbf{A_2}$ equivalent to NFA $\mathbf{A_1}$

	a	*b*	*c*	
0	1	2		
1		3,4		F
2	4,5			
3	6		0	
4			6,7,8	
5		8		F
6	0			
7	6	6		
8	7	7		



Add new state(s)

A_2

	a	*b*	*c*	
0	1	2		
1		34		F
2	$\boxed{45}$			
34	6		0678	
45		8	678	F
...				

continue...

Generating DFA A_2 equivalent to NFA A_1

... after few more iterations...

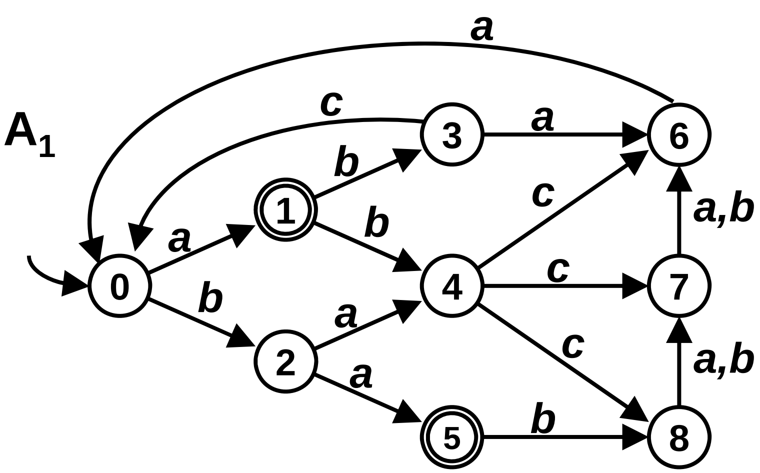
Add new state(s)

	a	b	c	
0	1	2		
1		3,4		F
2	4,5			
3	6		0	
4			6,7,8	
5		8		F
6	0			
7	6	6		
8	7	7		

A_2

	a	b	c	
0	1	2		
1		34		F
2	45			
34	6		0678	
45		8	678	F
6	0			
0678	0167	267		
8	7	7		
678	$\boxed{067}$	67		
0167	016	2346		F
267	0456	6		
7	6	6		
067				
...				

A_1



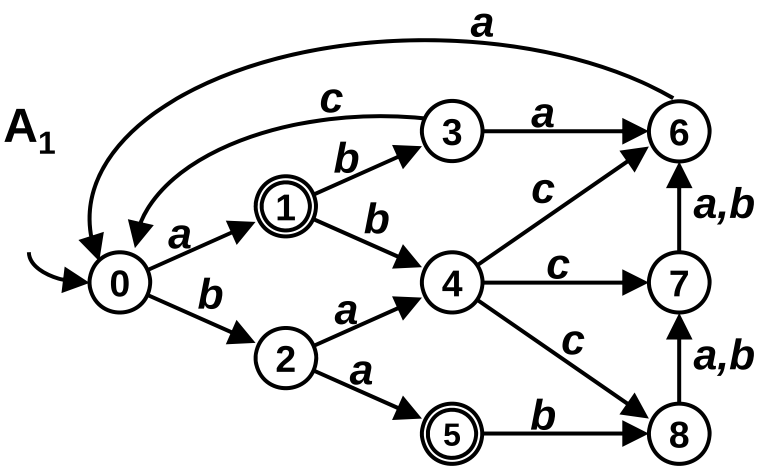
continue...

DFA A_2 equivalent to NFA A_1

...FINISHED!

	a	b	c	
0	1	2		
1		3,4		F
2	4,5			
3	6		0	
4			6,7,8	
5		8		F
6	0			
7	6	6		
8	7	7		

A_1



A_2

	a	b	c	
0	1	2	n	
1	n	34	n	F
2	45	n	n	
34	6	n	0678	
45	n	8	678	F
6	0	n	n	
0678	0167	267	n	
8	7	7	n	
678	067	67	n	
0167	016	2346	n	F
267	0456	6	n	
7	6	6	n	
067	016	2346	n	
67	06	6	n	
016	01	234	n	F
2346	0456	n	0678	
0456	01	28	678	F
06	01	2	n	
01	1	234	n	F
234	456	n	0678	
28	457	7	n	
456	0	8	678	F
457	6	68	678	F
68	07	7	n	
07	16	26	n	
16	0	34	n	
26	045	n	n	
654	1	28	678	F
n	n	n	n	

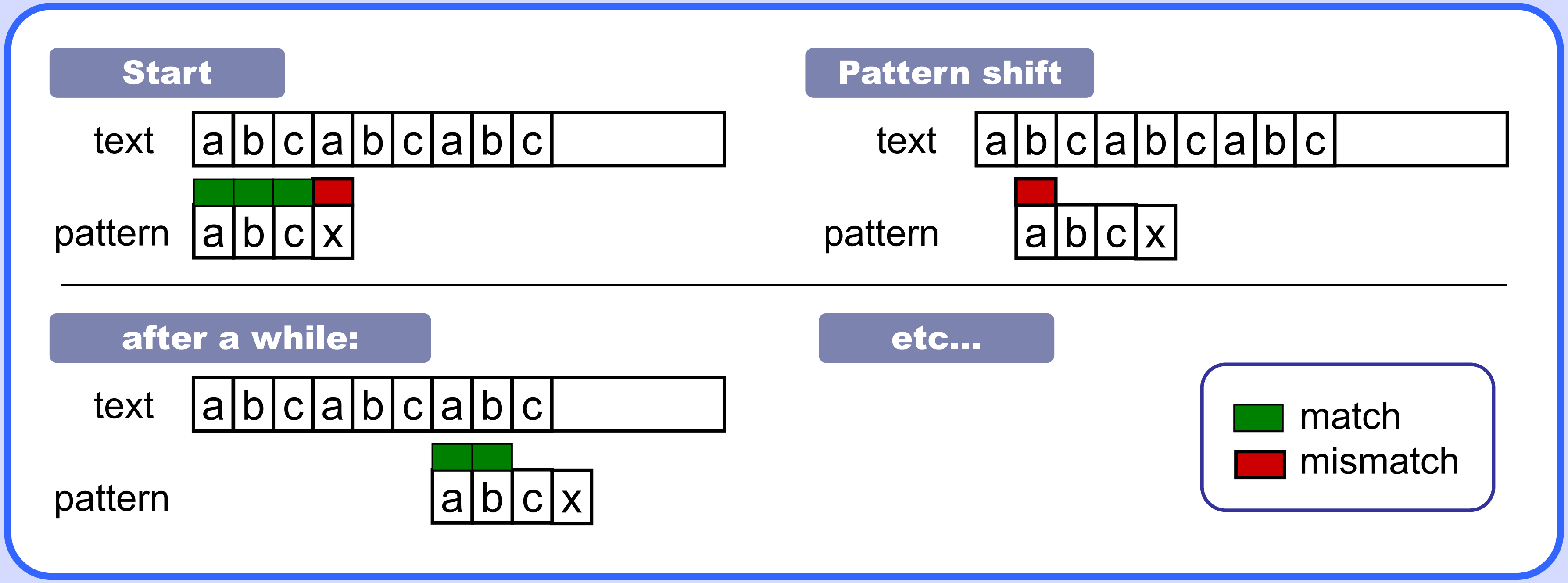
Text Search

To be used with great caution!

Naïve approach

1. Align pattern with the beginning of text.
2. While corresponding symbols of pattern and text match each other
   move forward by one symbol in pattern.
3. When symbol mismatch occurs shift pattern forward by one symbol, reset position in the pattern to the beginning of pattern and go to 2.
4. When the end of pattern is passed report success, reset position in the pattern to its beginning and go to 2.
5. When the end of text is reached stop.

Might be both efficient and not



Alphabet: Finite set of symbols.
Text: Sequence of symbols of the alphabet.
Pattern: Sequence of symbols of the same alphabet, pattern occurence is to be detected in the text

Text is often fixed or seldom changed, pattern typically varies (looking for different words in the same document), pattern is often significantly shorter than the text.

Notation

Alphabet: Σ

Symbols in the text: $t_1, t_2, \dots t_n$

Symbols in the pattern: $p_1, p_2, \dots p_m$

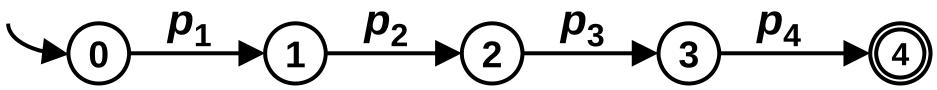
Holds $m \leq n$, usually $m << n$

Example

Text: `...task is very simple but it is used very freq...`
Pattern: `simple`

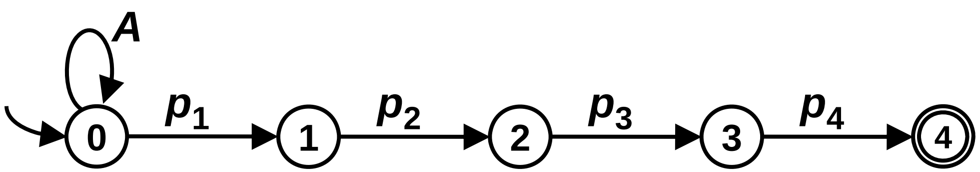
NFA A_3 which accepts just a single word $p_1p_2p_3p_4$.

A_3



NFA A_4 which accepts each word with suffix $p_1\ p_2\ p_3\ p_4$ with its transition table.

A_4



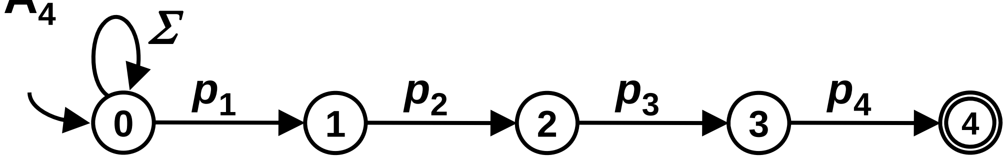
	p_1	p_2	p_3	p_4	z	
0	0,1	0	0	0	0	
1		2				
2			3			
3				4		
4						F

$z \in \Sigma - \{p1, p2, p3, p4\}$

repeated

NFA A_4 which accepts
each word with suffix $p_1\, p_2\, p_3\, p_4$
and its transition table.

A_4



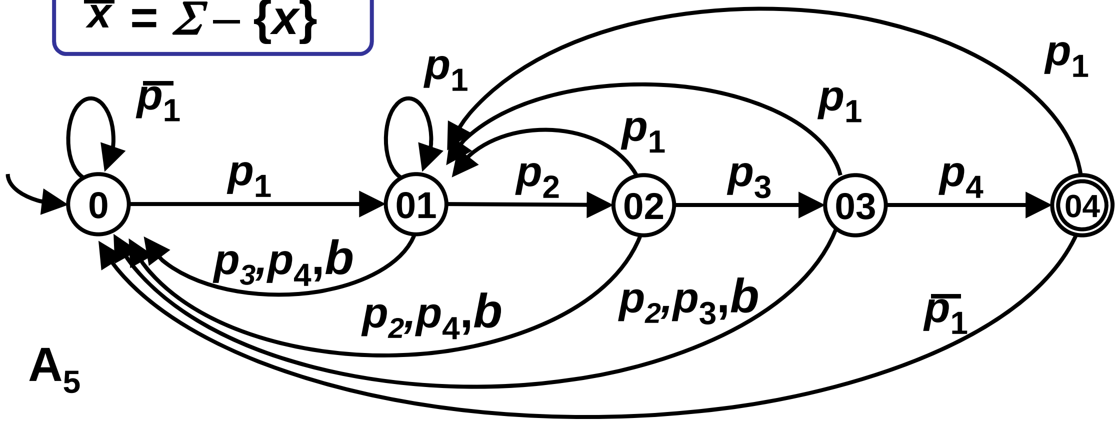
	p_1	p_2	p_3	p_4	z	
0	0,1	0	0	0	0	
1		2				
2			3			
3				4		
4						F

$z \in \Sigma - \{p1, p2, p3, p4\}$

equivalently

DFA A_5 is a deterministic equivalent of NFA A_4.

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

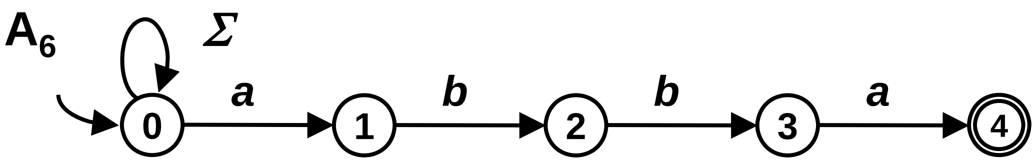


A_5

	p_1	p_2	p_3	p_4	z	
0	01	0	0	0	0	
01	01	02	0	0	0	
02	01	0	03	0	0	
03	01	0	0	04	0	
04	01	0	0	0	0	F

example

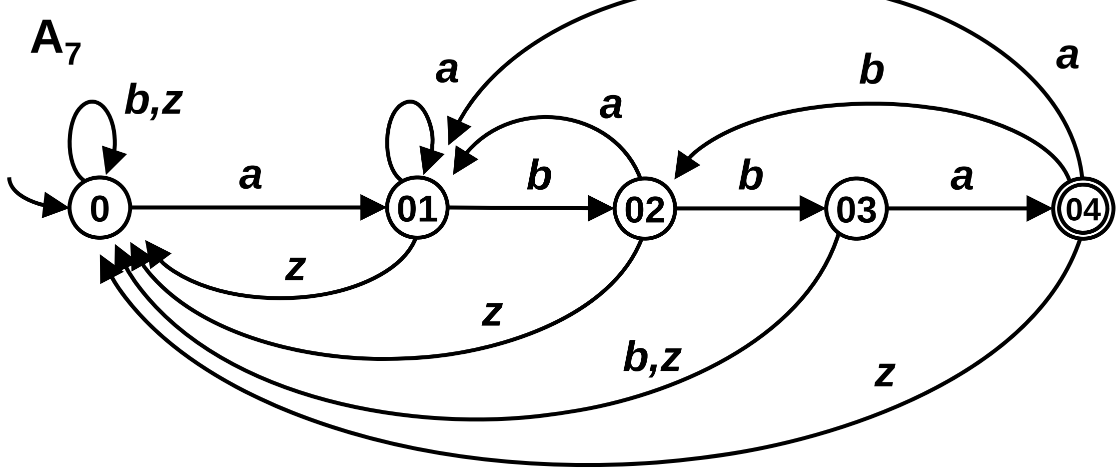
NFA A_6 which accepts each word with suffix *abba* with its transition table



	a	b	z	
0	0,1	0	0	
1		2		
2		3		
3	4			
4				F

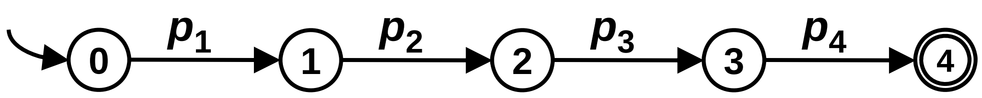
$z \in \Sigma - \{a, b\}$

DFA A_7 is a deterministic equivalent of NFA A_6.
It also accepts each word with suffix *abba*.

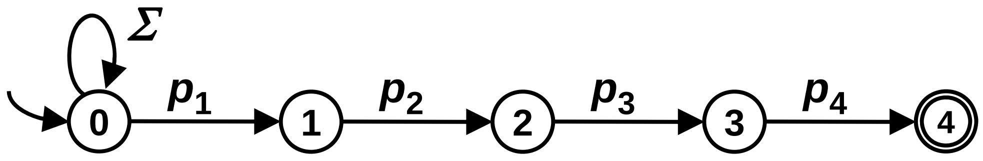


	a	b	z	
0	01	0	0	
01	01	02	0	
02	01	03	0	
03	014	0	0	
04	01	02	0	F

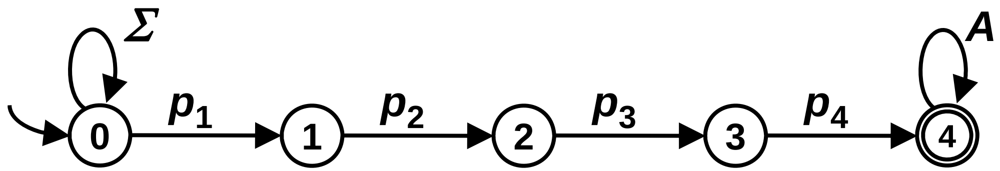
NFA accepting exactly one word $p_1p_2p_3p_4$.



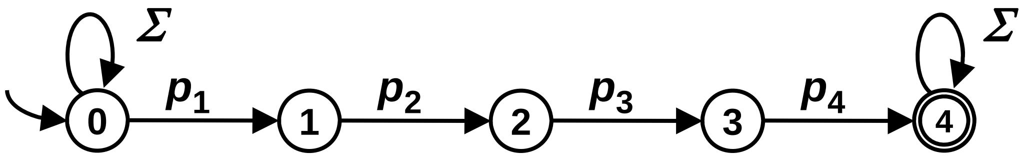
NFA accepting any word with suffix $p_1p_2p_3p_4$.



NFA accepting any word with substring (factor) $p_1p_2p_3p_4$ anywhere in it.

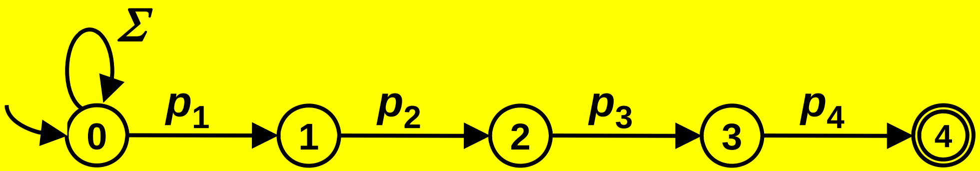


NFA accepting any word with substring (factor) $p_1p_2p_3p_4$ anywhere in it.



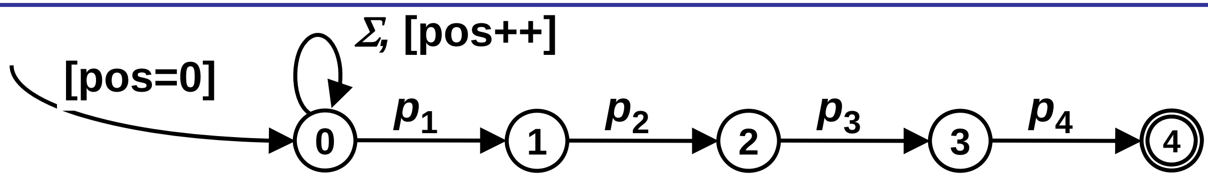
Can be used for search, but the following reduction is usual.

Text search NFA for finding pattern $P = p_1p_2p_3p_4$ in the text.



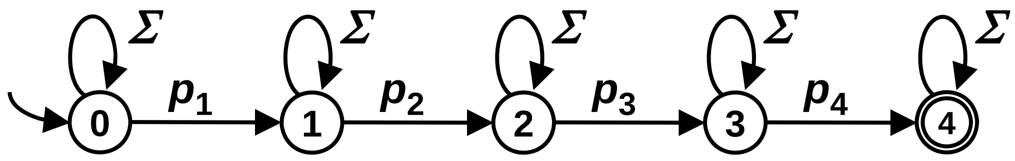
NFA stops when pattern is found.

Want to know the position of the pattern in the text?
Equip the transitions with a counter.



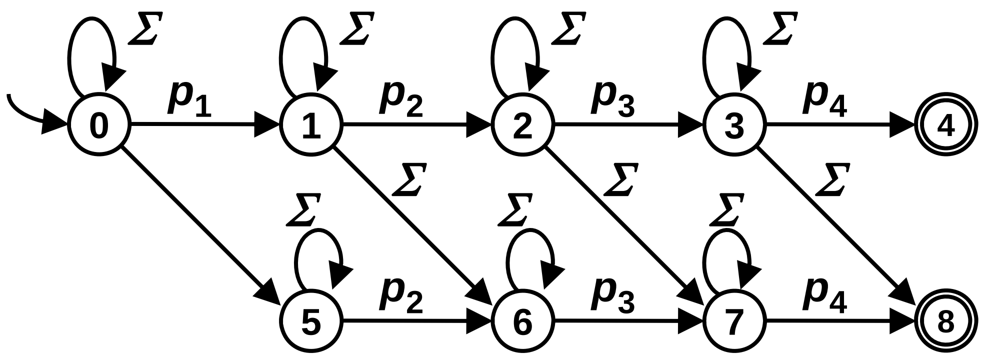
Example

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.

Languages Hierarchy

Search NFA can search for more than one pattern simultaneously.
The number of patterns can be
finite -- this leads to dictionary automaton (we will meet them later)
or infinite -- this leads to regular language.

Chomsky language hierarchy remainder

Grammar	Language	Automaton
Type-0	Recursively enumerable	Turing machine
Type-1	Context-sensitive	Linear-bounded non-deterministic Turing machine
Type-2	Context-free	Non-deterministic pushdown automaton
Type-3	Regular	Finite state automaton (NFA or DFA)

Only regular languages can be processed by NFA/DFA. More complex languages cannot. For example any language containing *well-formed parentheses* is context-free and not regular and cannot be recognized by NFA/DFA.

Operations on regular languages

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.

$L_1.L_2$ 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 of language L_1. It is 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 $L_1 \cup L_2$, $L_1.L_2$, L_1^* are regular languages too.

Example

L_1 = {001, 0001, 00001, ...}, L_2 = {110, 1110, 11110, ...}.

$L_1 \cup L_2$ = {001, 110, 0001, 1110, 0001, 1110, ...}

$L_1.L_2$ = {001110, 0011110, 00111110, ..., 0001110, 00011110, 000111110, ... }

L_1^* = {ε, 001, 001001, 001001001, ... 0010001, 00100010001, ...
..., 00100001, 001000001, ... } // this one is not easy to list nicely ... or is it?

Regular expressions defined recursively
Symbol ε is regular expression.
Each symbol of alphabet Σ is regular expression.
Whenever e_1 and e_2 are regular expressions also strings (e_1), e_1+e_2, e_1e_2, $(e_1)^*$ are regular expressions.

Languages represented by regular expressions (RE) defined recursively
RE ε represents language containing only empty string
RE x, where $x \in \Sigma$, represents language $\{x\}$.
Let RE's e_1 and e_2 represent languages L_1 and L_2. Then
RE (e_1) represents L_1, RE e_1+e_2 represents $L_1 \cup L_2$, RE e_1e_2 represents $L_1.L_2$,
RE $(e_1)^*$ represents L_1^* .

Examples
$0+1(0+1)^*$ all integers in binary without leading 0's
$0.(0+1)^*1$ all finite binary fractions $\in (0, 1)$ without trailing 0's
((0+1)(0+1+2+3+4+5+6+7+8+9) + 2(0+1+2+3)):(0+1+2+3+4+5)(0+1+2+3+4+5+6+7+8+9)
all 1440 day's times in format hh:mm from 00:00 to 23:59
(mon+(wedne+t(ue+hur))s+fri+s(atur+un))day
English names of days in the week
$(1+2+3+4+5+6+7+8+9)(0+1+2+3+4+5+6+7+8+9)^*((2+7)5+(5+0)0)$
all decimal integers ≥ 100 divisible by 25

Convert regular expression to NFA

Input: Regular expression R containing *n* characters of the given alphabet.
Output: NFA recognizing language L(R) described by R.

Create start state S

```
for each k (1 ≤ k ≤ n) {
   assign index k  to the k-th character in R
    // this makes all characters in R unique: c[1], c[2], ..., c[n].
   create state S[k]          // S[k] corresponds directly to c[k]
}

for each k (1 ≤ k ≤ n) {
    if c[k] can be the first character in some string described by R
       then create transition S → S[k]  labeled by c[k] with index stripped off
    if c[k] can be the last character in some string described by R
       then mark S[k] as final state
    for  each p (1 ≤ p ≤ n)
       if (c[k] can follow immediately after c[p] in some string described by R)
         then create transition S[p] → S[k]  labeled by c[k] with index stripped off
}
```

Regular Expression to NFA

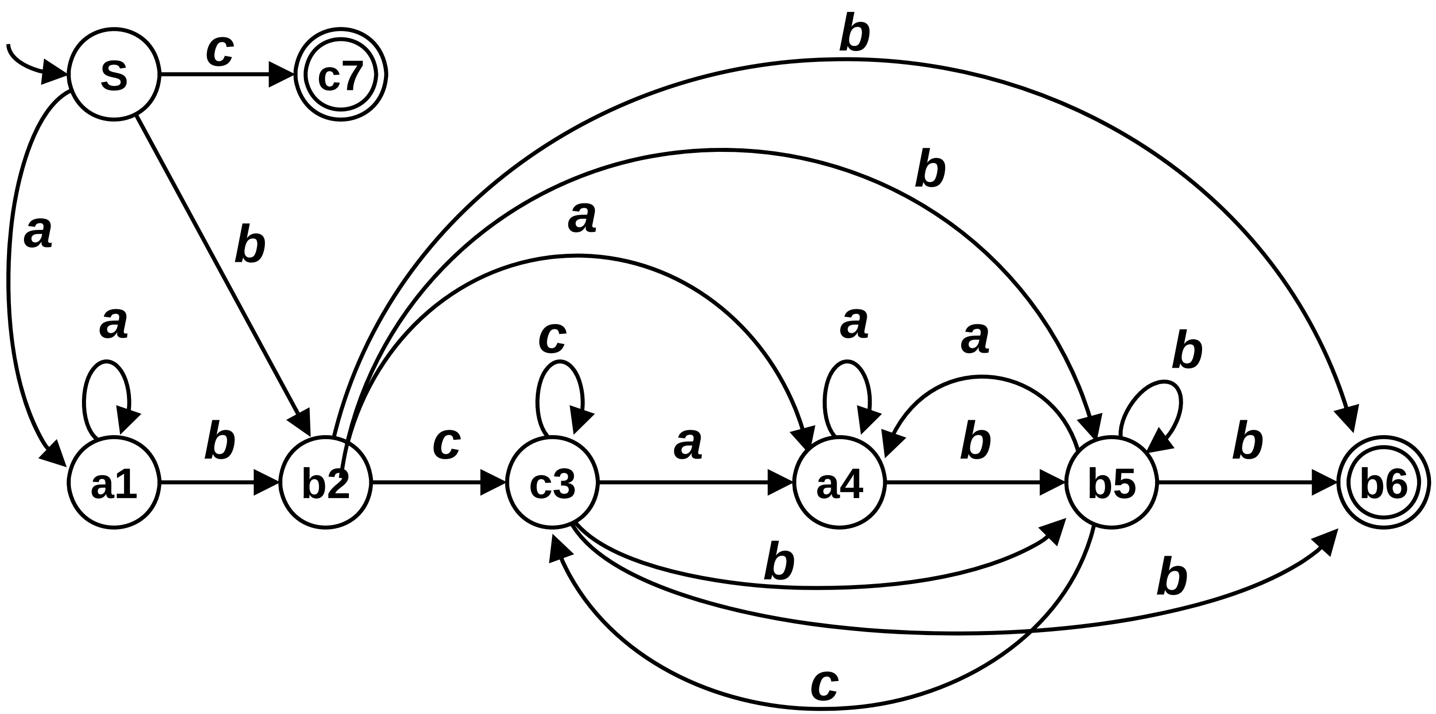
Regular expression

$R = a^*b(c + a^*b)^* b + c$

Add indices:

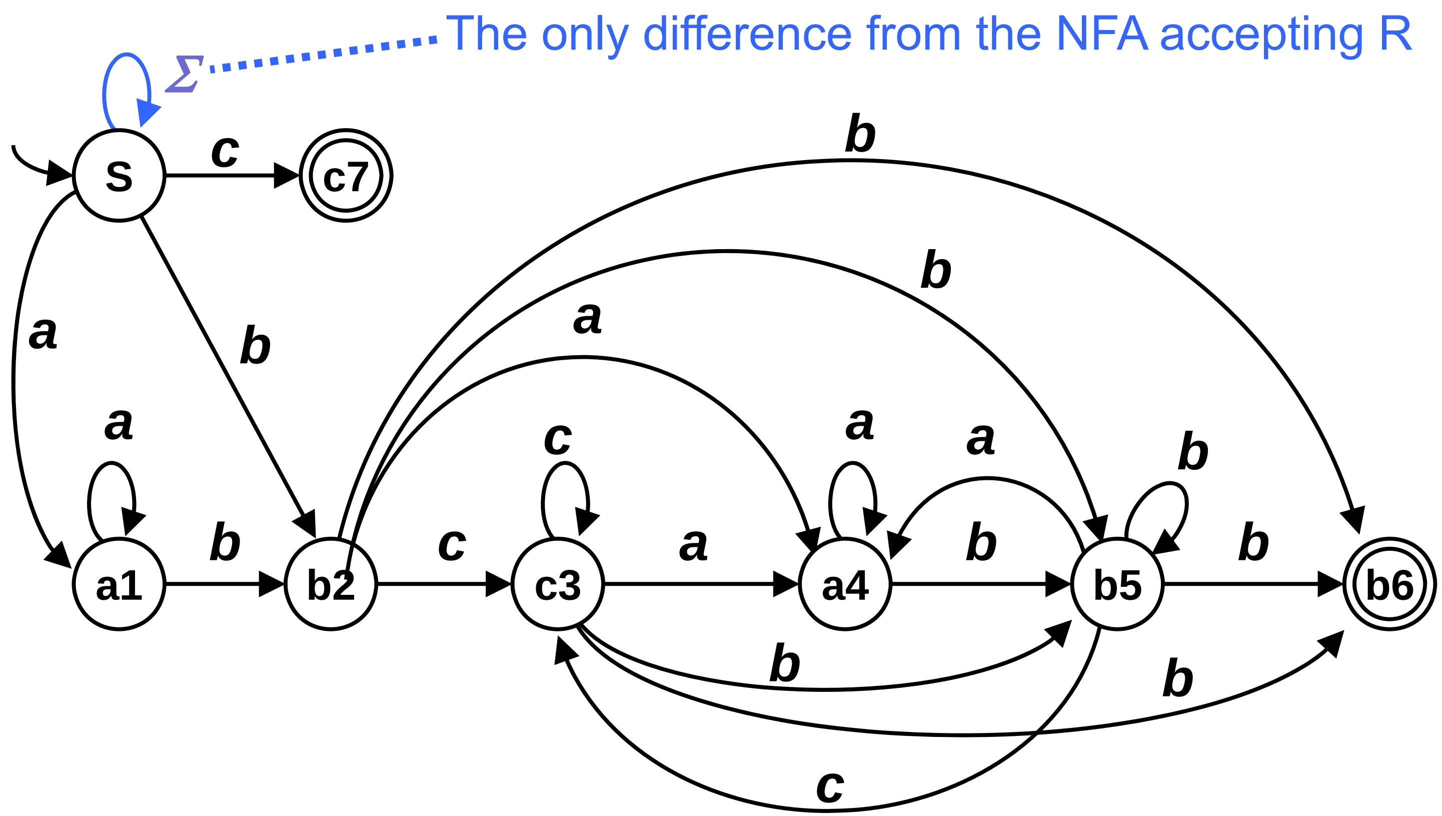
$R = a_1^* b_2(c_3 + a_4^*b_5)^* b_6 + c_7$

NFA accepts L(R)



NFA searches the text for any occurence of any word of L(R)

R = a*b (c + a*b)* b + c



Bonus

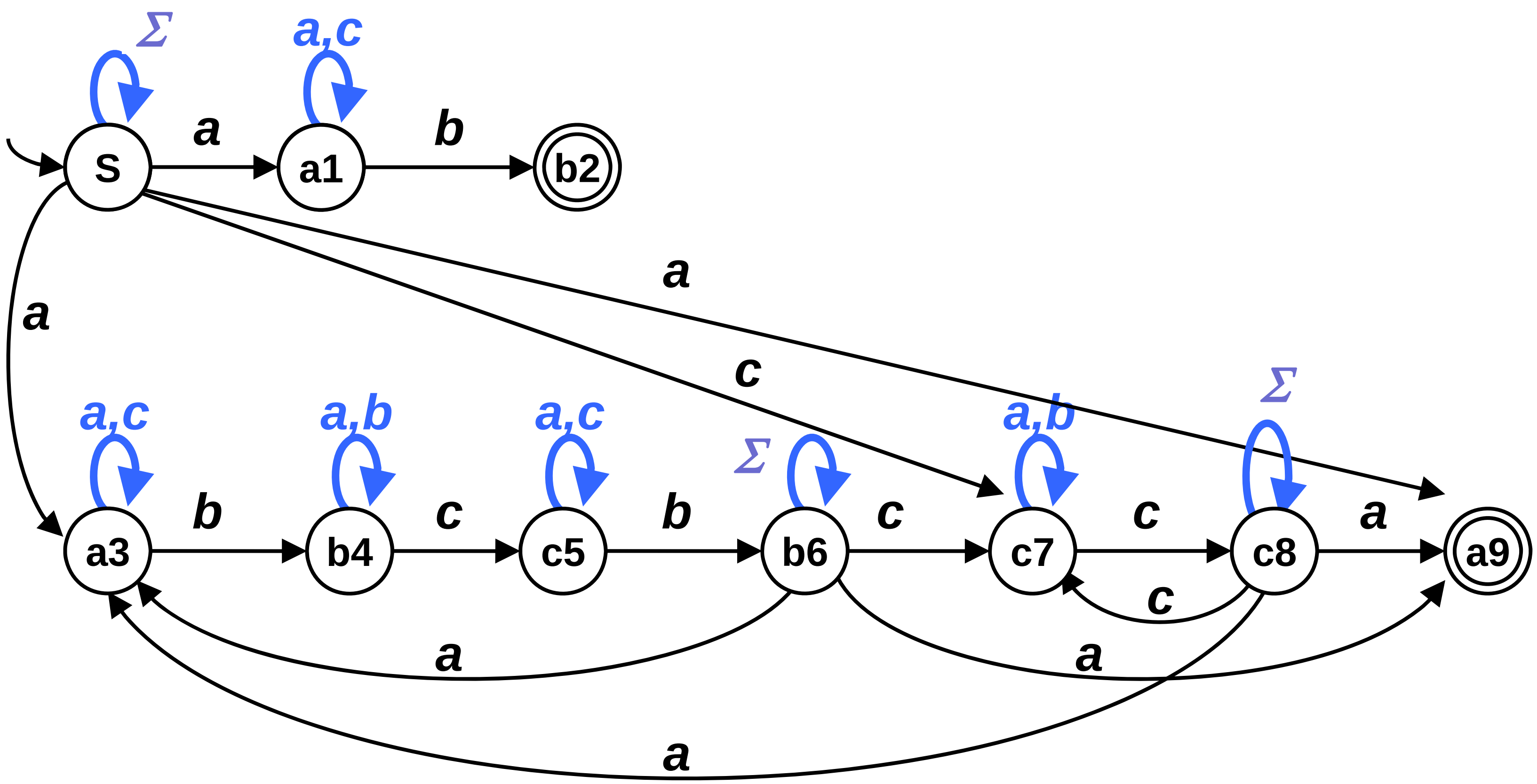
To find a subsequence representing a word $\in$ L(R), where R is a regular expression, do the following:

Create NFA acepting L(R)
Add self loops to the states of NFA:
1. Self loop labeled by Σ (whole alphabet) at the start state.
2. Self loop labeled $\Sigma - \{x\}$ at each state which outgoing transition(s) are labeled by single $x \in \Sigma$. // serves as an "optimized" wait loop
3. Self loop labeled by Σ at each state which outgoing transition(s) are labeled by more than single symbol from Σ. // serves as an "usual" wait loop
4. No self loop to all other states. // which have no outgoing loop, final ones

Bonus

NFA searches the text for any occurence of any subsequence representing a word word of L(R)

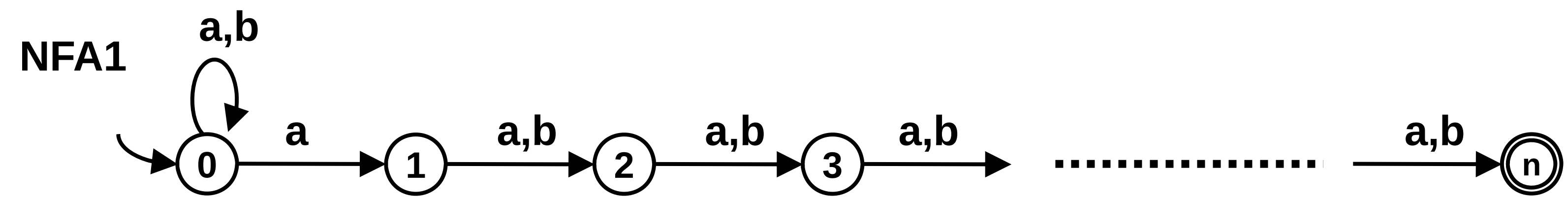
R = ab + (abcb + cc)* a



Transforming NFA which searches text for an occurence of a word of a given regular language into the equivalent DFA might take exponential space and thus also exponential time. Not always, but sometimes yes:

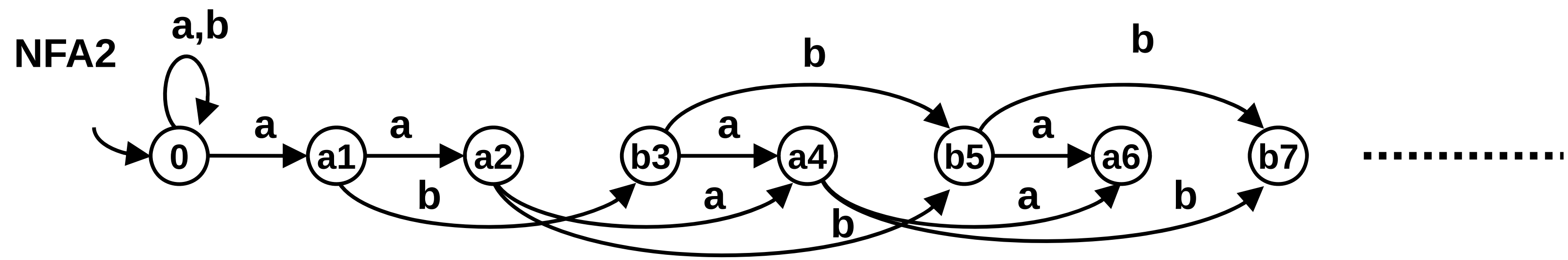
Consider regular expression **R** = **a(a+b)(a+b)...(a+b)** over alphabet {a, b}.

Text search NFA1 for R



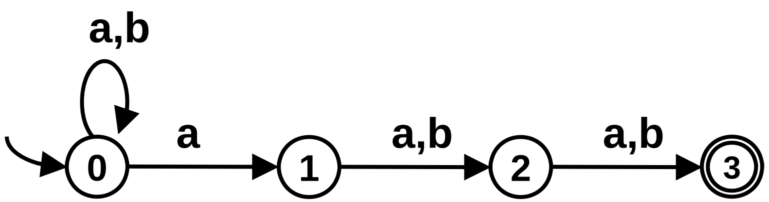
Mystery

Text search NFA2 for R, why not this one?



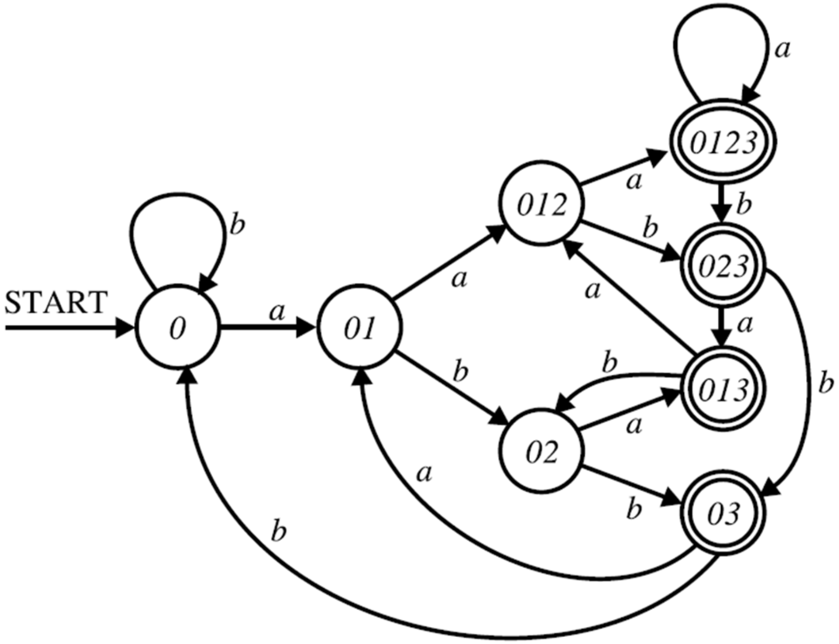
R = a(a+b)(a+b)

Text search NFA for R



NFA table

	a	b
0	0,1	0
1	2	2
2	3	3
3	-	-



DFA table

	a	b
0	01	0
01	012	02
012	0123	023
0123	0123	023
02	013	03
023	013	03
013	012	02
03	01	0

Search the text for more than just exact match

NFA with ε–transitions

The transition from one state to another can be performed **without** reading any input symbol. Such transition is labeled by symbol ε.

ε**–closure**

Symbol ε–CLOSURE(p) denotes the set of all states q, which can be reached from p using only ε–transitions.

By definition let ε–CLOSURE(p) = {p}, when there is no ε–transition out from p.

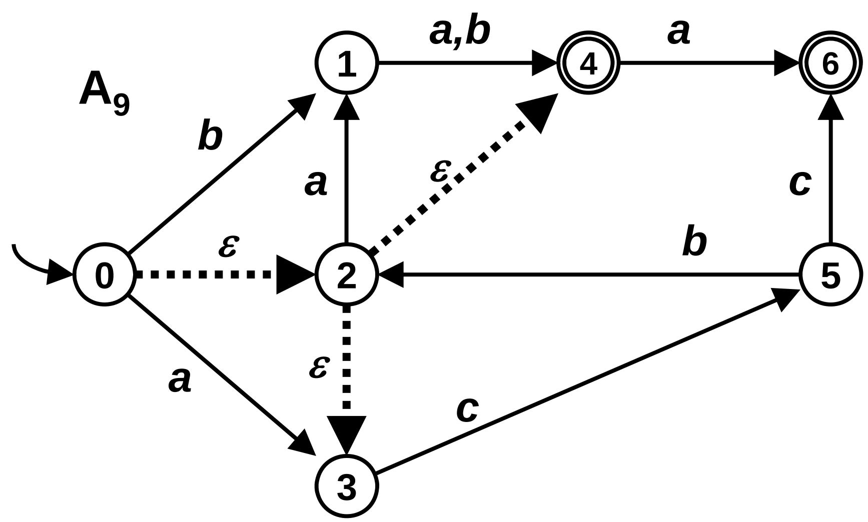
ε–CLOSURE(0) = {2, 3, 4}

ε–CLOSURE(1) = {1}

ε–CLOSURE(2) = {3, 4}

ε–CLOSURE(3) = {3}

...

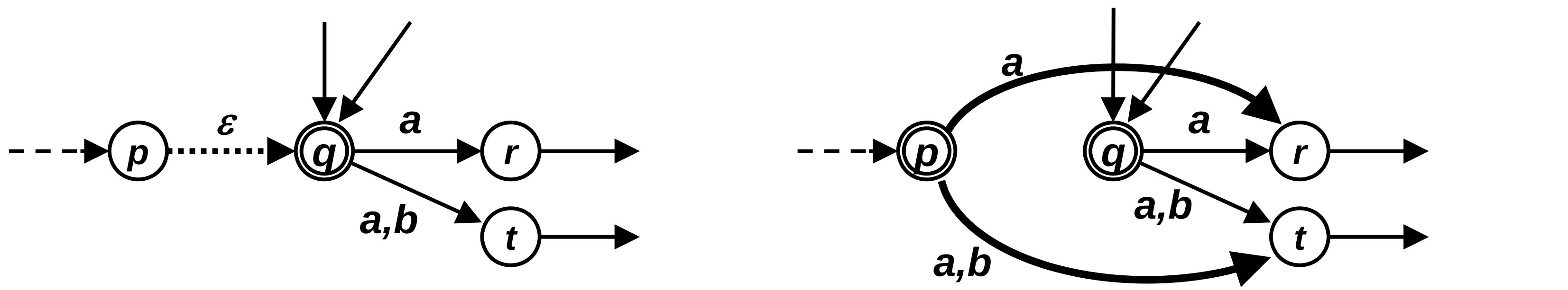


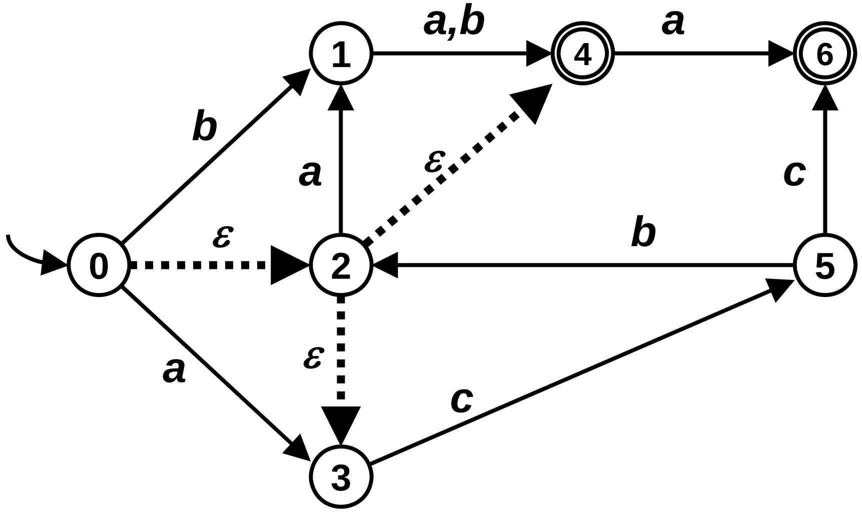
Construction of equivalent NFA without ε–transitions

Input: NFA A with some ε–transitions.
Output: NFA A' without ε–transitions.

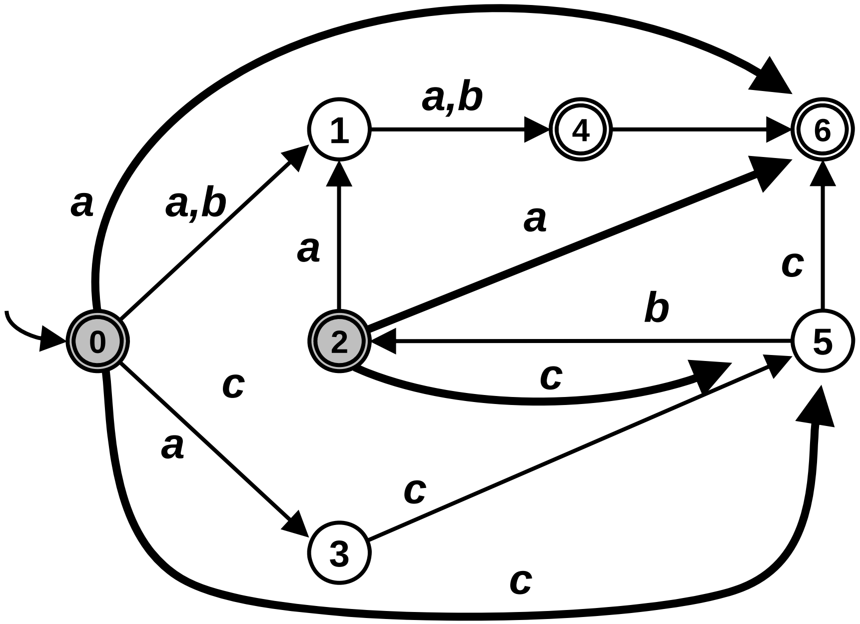
1. A' = exact copy of A.
2. Remove all ε–transitions from A'.
3. In A' for each (q, a) do: add to the set $\delta(p,a)$ all such states r for which holds $q \in \varepsilon$–CLOSURE(p) and $\delta(q,a) = r$.
4. Add to the set of final states F in A' all states p for which holds ε–CLOSURE$(p) \cap F \neq \emptyset$.

easy construction





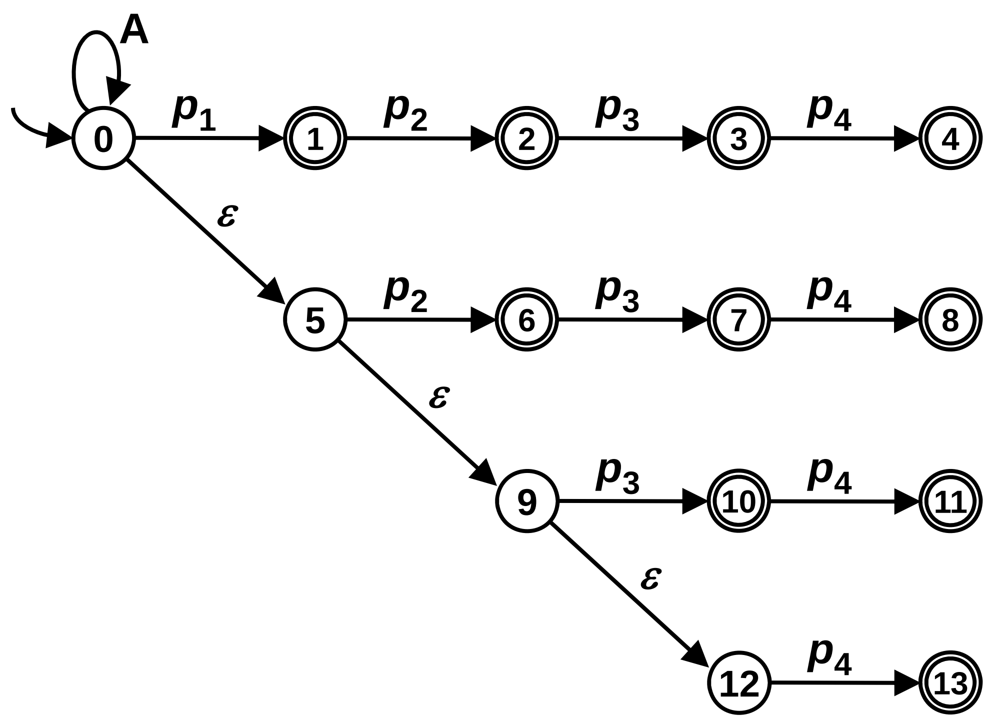
NFA with s ε–transitions



Equivalent NFA
without ε–transitions

New transitions and accept states are highlighted

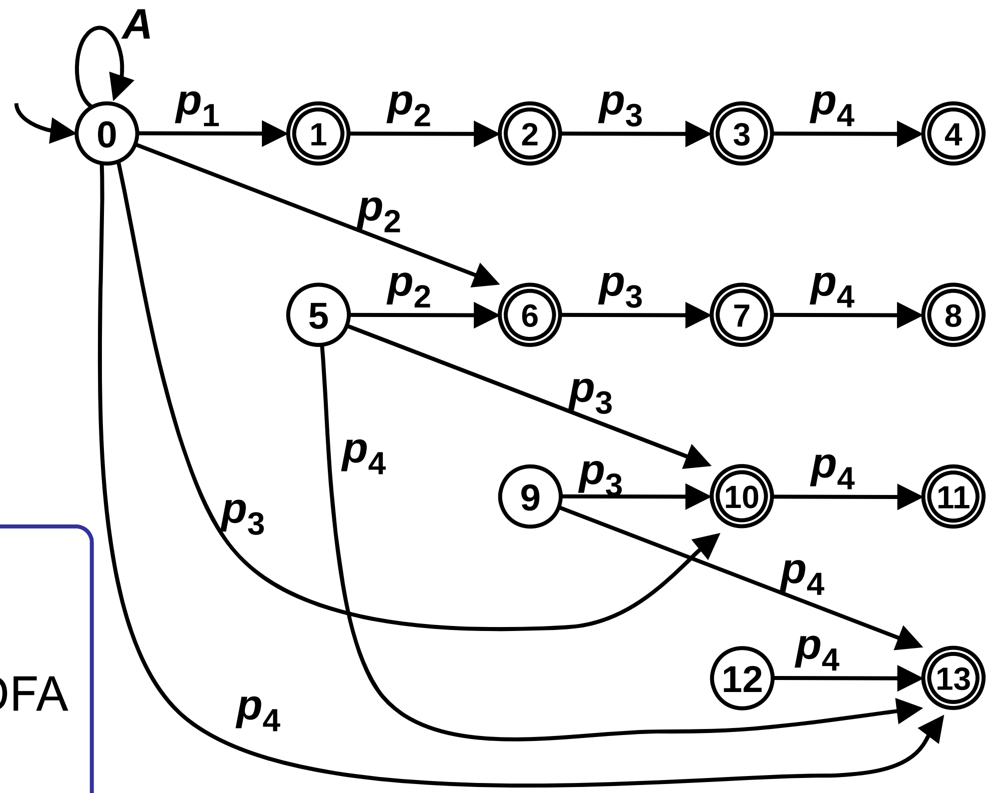
NFA for search for any unempty substring of pattern $p_1p_2p_3p_4$ over alphabet Σ.
Note the ε–transitions.



Powerful trick!

Union of two or more NFA:
Create additional start state S and add ε–transitions from S to start states of all involved NFA's. Draw an example yourself!

Equivalent NFA for search for any unempty substring of pattern $p_1p_2p_3p_4$ with ε–transitions removed.



States 5, 9, 12 are unreachable.
Transformation algorithm NFA -> DFA if applied, will neglect them.

	p_1	p_2	p_3	p_4	z	
0	0,1	0,6	0,10	0,13	0	
1		2			0	F
2			3		0	F
3				4	0	F
4					0	F
5		6	10	13	0	
6			7		0	F
7				8	0	F
8					0	F
9			10	13	0	
10				11	0	F
11					0	F
12				13	0	
13					0	F

	p_1	p_2	p_3	p_4	z	
0	0.1	0.6	0.10	0.13	0	
0.1	0.1	0.2.6	0.10	0.13	0	F
0.6	0.1	0.6	0.7.10	0.13	0	F
0.10	0.1	0.6	0.10	0.11.13	0	F
0.13	0.1	0.6	0.10	0.13	0	F
0.2.6	0.1	0.6	0.3.7.10	0.13	0	F
0.7.10	0.1	0.6	0.10	0.8.11.13	0	F
0.11.13	0.1	0.6	0.10	0.13	0	F
0.3.7.10	0.1	0.6	0.10	0.4.8.11.13	0	F
0.8.11.13	0.1	0.6	0.10	0.13	0	F
0.4.8.11.13	0.1	0.6	0.10	0.13	0	F

Transition table of NFA above without ε–transitions.

Transition table of DFA which is equivalent to previous NFA.

DFA in this case has less states than the equivalent NFA.
Q: Does it hold for any automaton of this type? Proof?

Text search using NFA simulation without transform to DFA

Input: NFA , text in array t,

```
SetOfStates S = eps_CLOSURE(q0), S_tmp;
int i = 1;
while ((i <= t.length) && (!S.empty())) {
  for (q in S)     // for each state in S
    if (q.isFinal)
      print(q.final_state_info); // pattern found
  S_tmp = Set.empty();
  for (q in S)
    S_tmp.union(eps_CLOSURE(delta(q, t[i]);));
  S = S_tmp;
  i++;
}

return S.containsFinalState();  // true or false
```