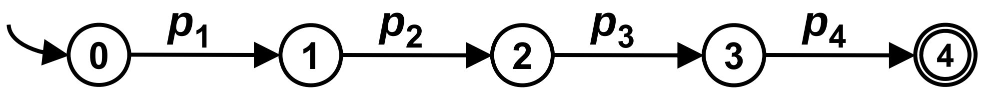
Text Search

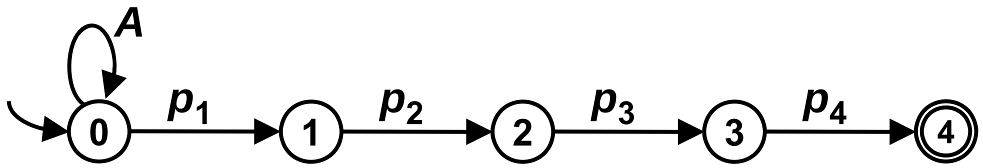
- Power of nondeterministic approach
- Levenshtein distance and Dynamic Programming
- Epsilon-transitions and their removal
- Levenshtein distance search NFA
- Regular expression search
- Dictionary search

Literature: Borivoj Melichar, Jan Holub, Tomas Polcar
TEXT SEARCHING ALGORITHMS VOLUME I.
.CTU, FEE, Nov 2005

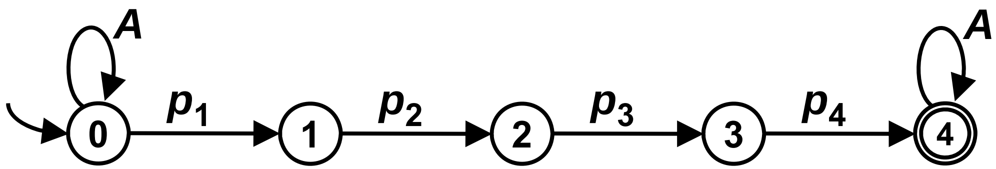
NFA accepting exactly one word $p_1p_2p_3p_4$.



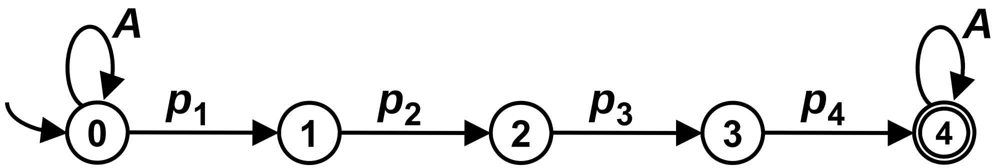
NFA accepting any word with suffix $p_1p_2p_3p_4$.



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

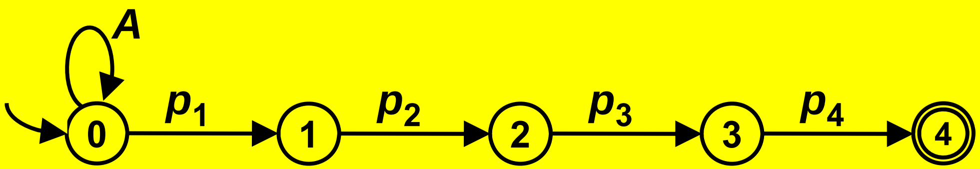


NFA accepting any word with substring (factor) $p_1p_2p_3p_4$ anywhere in it.



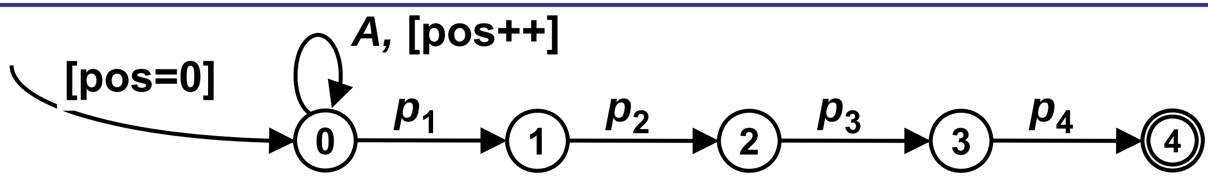
Could be used for search, but the following reduction is usual.

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



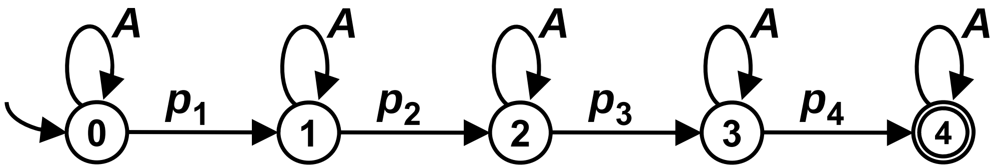
NFA stops when pattern is found.

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



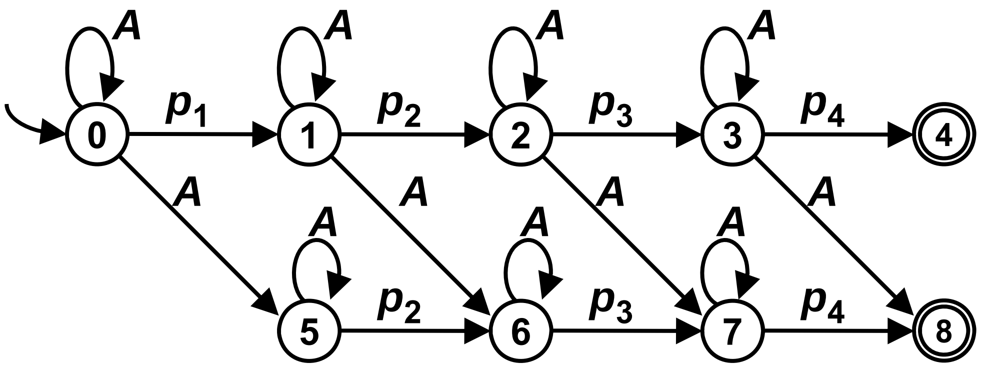
Example

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

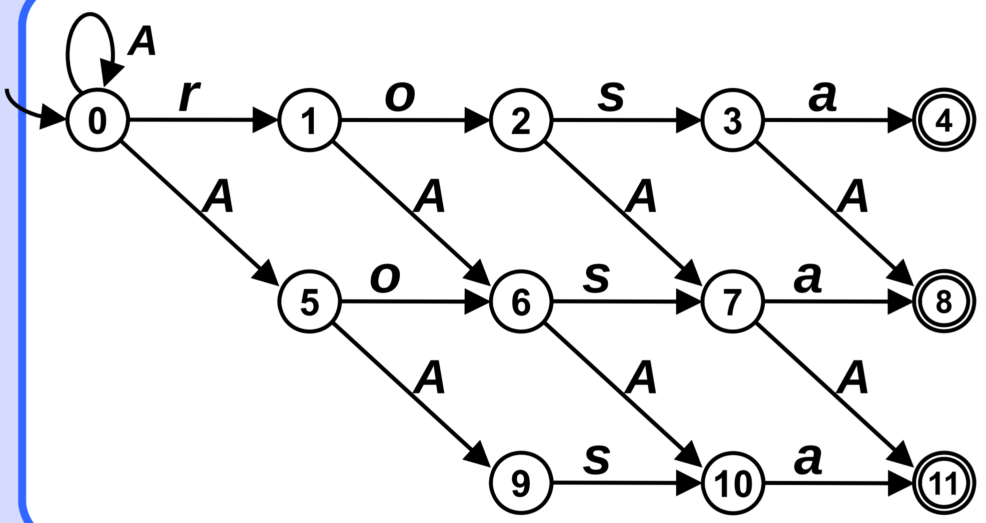


Example

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

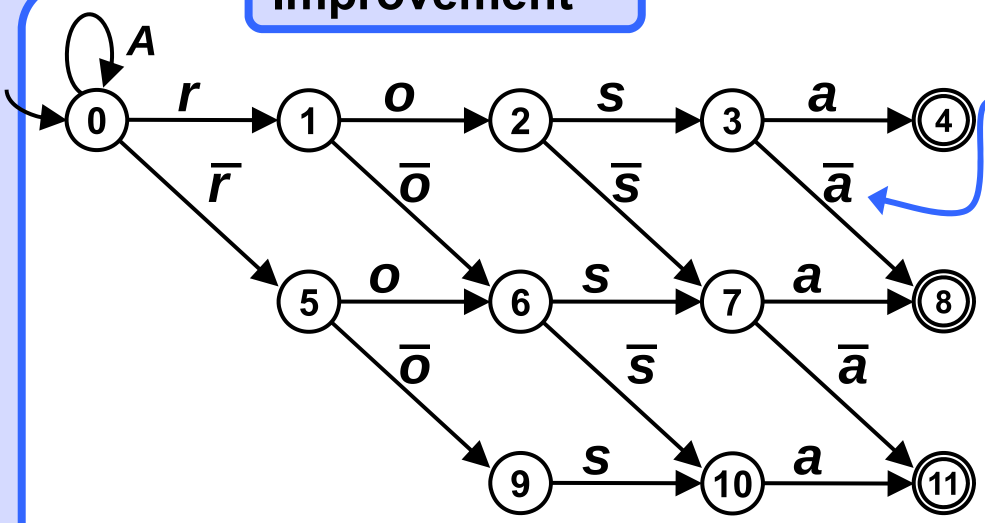


Alternatively: NFA accepting any word containing a subsequence Q which Hamming distance from $p_1p_2p_3p_4$ is at most 1.



Hamming distance of the found pattern Q from pattern P = rosa cannot be deduced from the particular end state.
E.g.: "ropa":
r - 1 - o - 2 - p - 7 - a - 8.
r - 5 - o - 6 - p - 10 - a - 11.

Improvement



Notation: $\overline{x}$ = A — {x}
means: Complement of x in A.

Hamming distance from the pattern P = rosa 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 ($k \geq 0$), that we can change A to o B or B to A
by applying exactly k edit operations on one of them.
The edit operation is Remove, Insert or Rewrite any symbol of the alphabet anywhere in the string. (Rewrite is also called Substitution.)

Levenshtein distance is thus 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]) **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.

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·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.

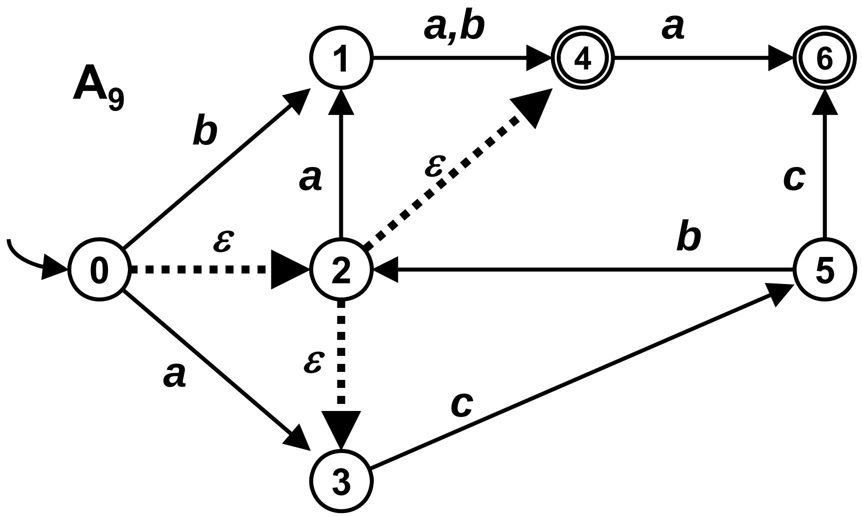
Why bother to do calculations with DP?

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}
...

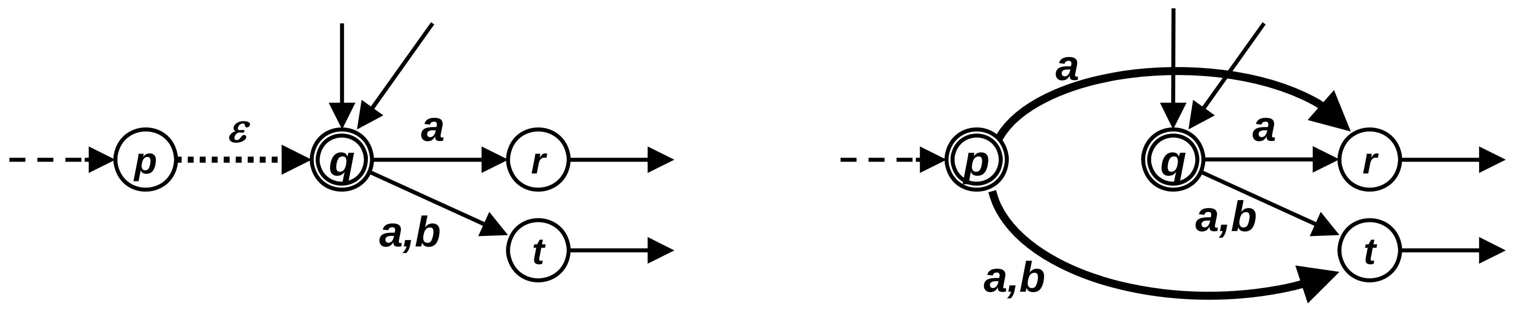


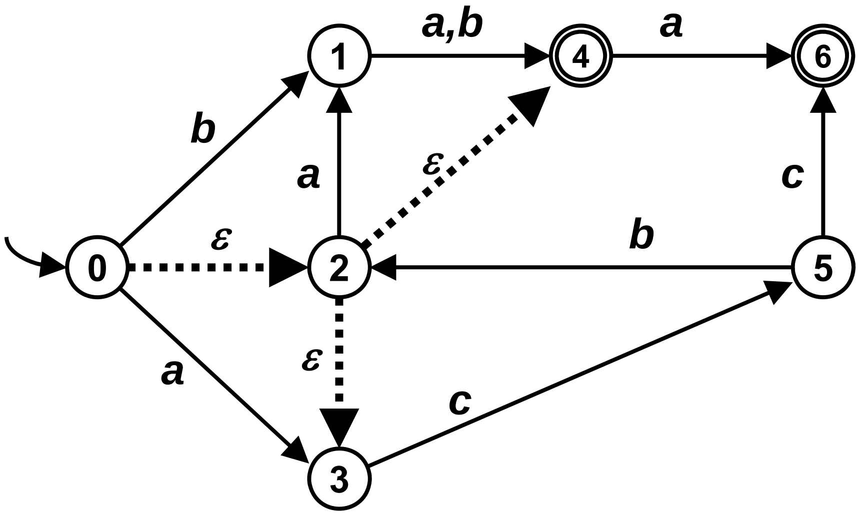
Construction of equivalent NFA without ε–transitions

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

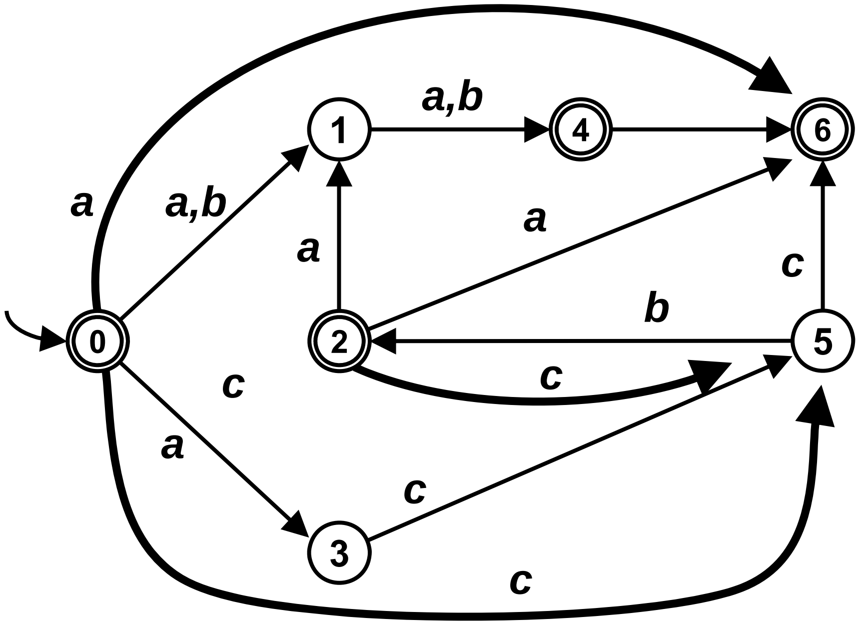
1. A' = exact copy of A.
2. Remove all ε–transitions from A'.
3. In A' for each (q, a) do: add to the set $\delta(p,a)$ all such states r for which holds $q \in \varepsilon$–CLOSURE(p) and $\delta(q,a) = r$.
4. Add to the set of final states F in A' all states p for which holds ε–CLOSURE(p) $\cap F \neq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

easy construction



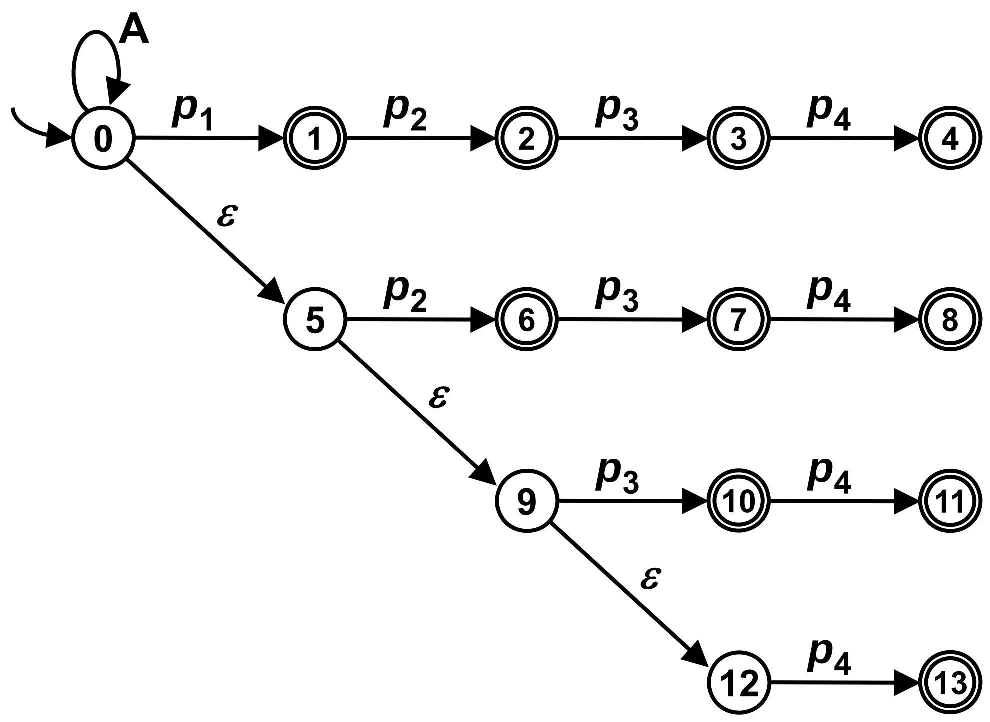


NFA with s ε–transitions



Equivalent NFA
without ε–transitions

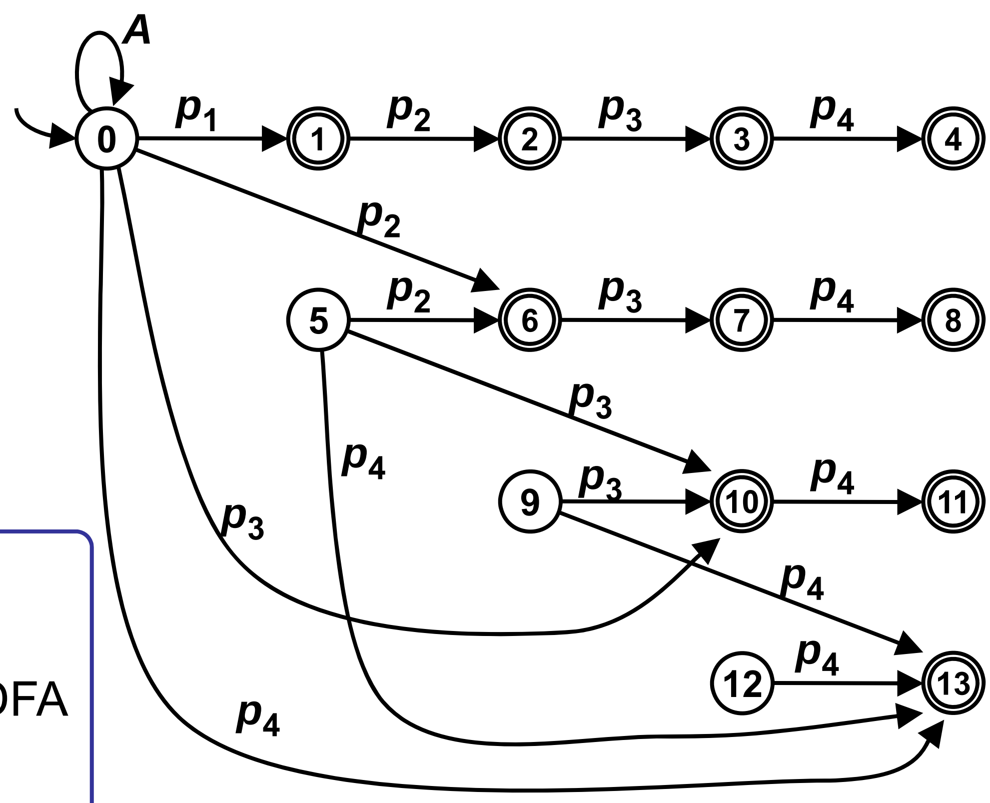
NFA for search for any unempty substring of pattern $p_1p_2p_3p_4$ over alphabet A.
Note the ε–transitions.



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

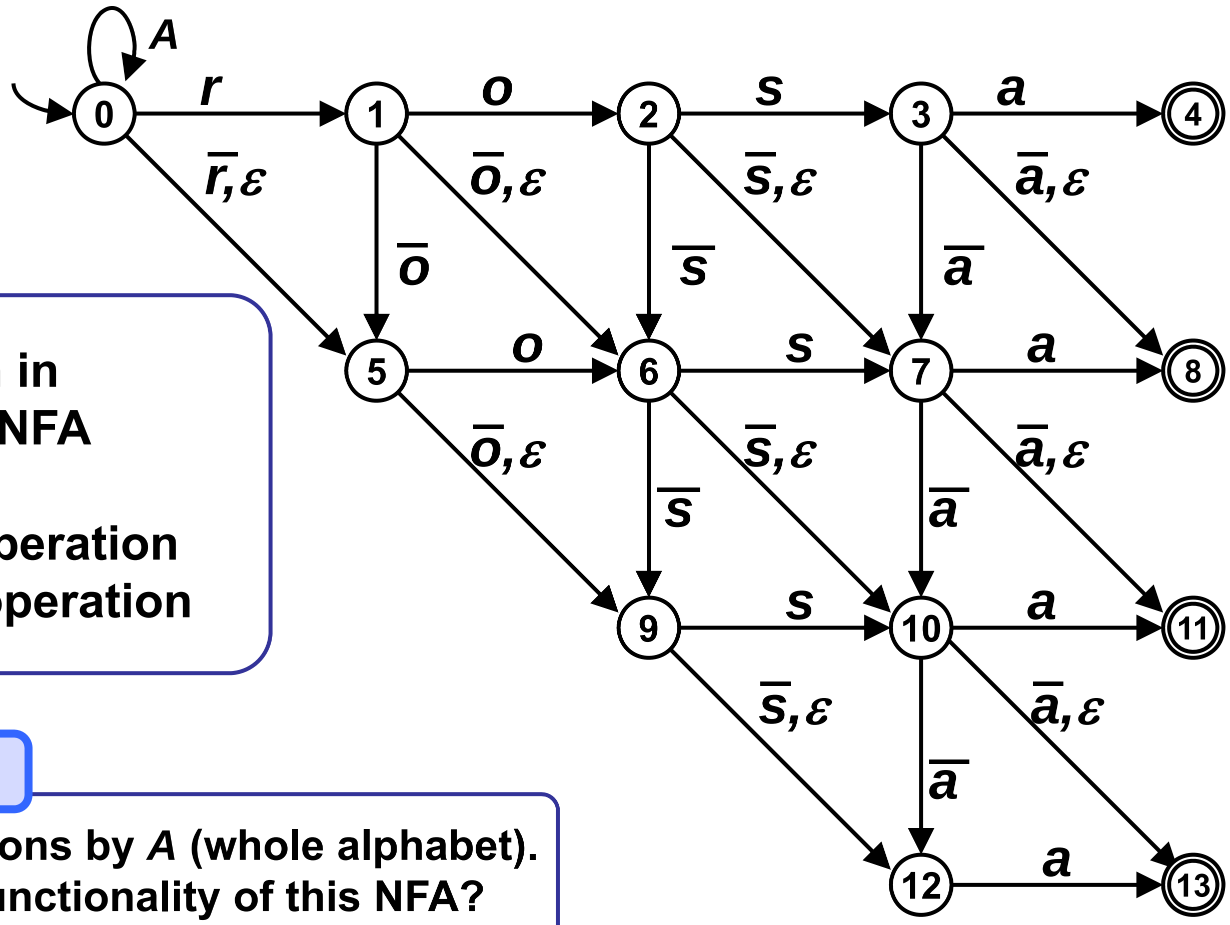
Transition table of NFA above without ε–transitions.

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?

NFA searches in text for a pattern within the given Levenshtein distance from the pattern "rosa".

Note the ε–transitions.



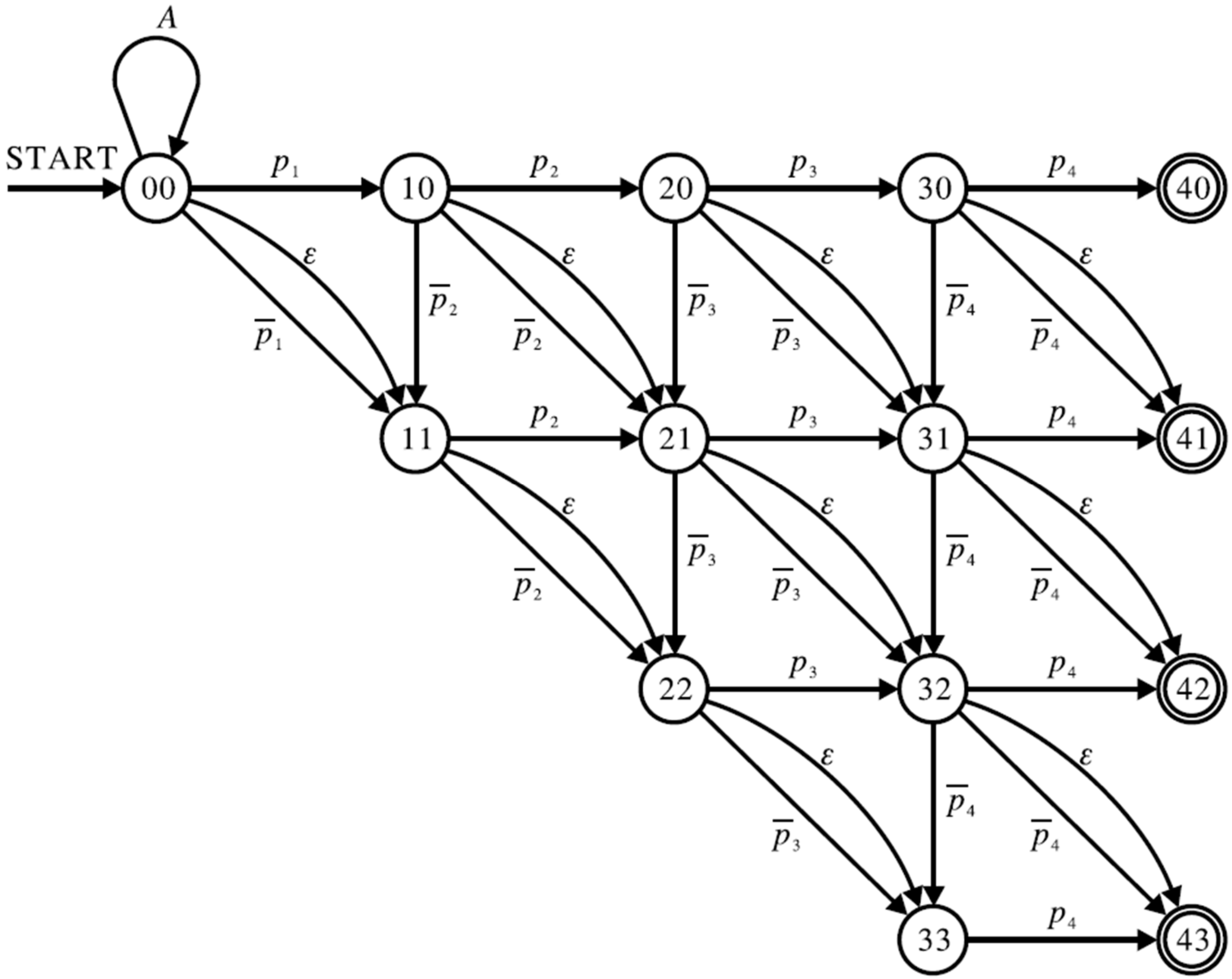
More transitions than in a Hamming distance NFA

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

Self-check question

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

Another example



Search NFA can search for more than one pattern simultaneously.
The number of patterns can be **finite** -- dictionary automaton
or infinite -- 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.

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
}
```

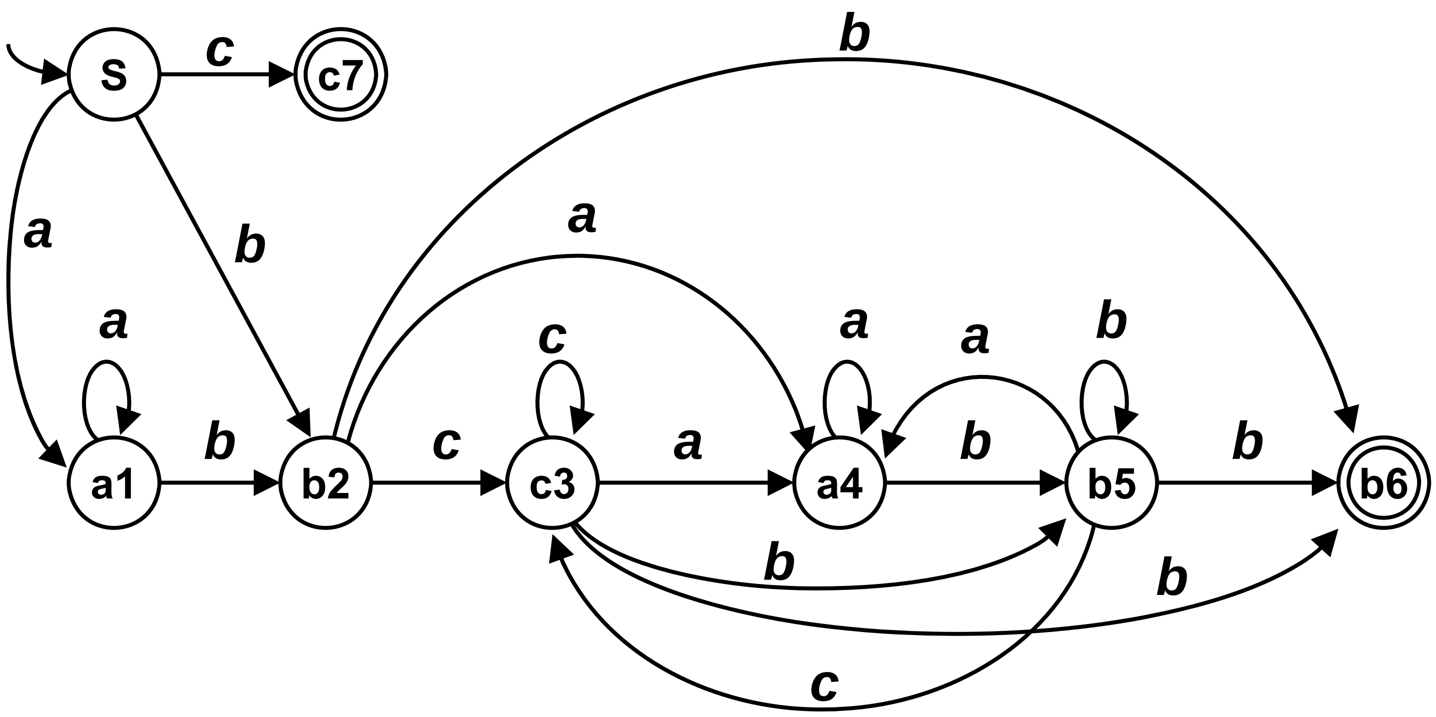
Regular expression

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

Indices:

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

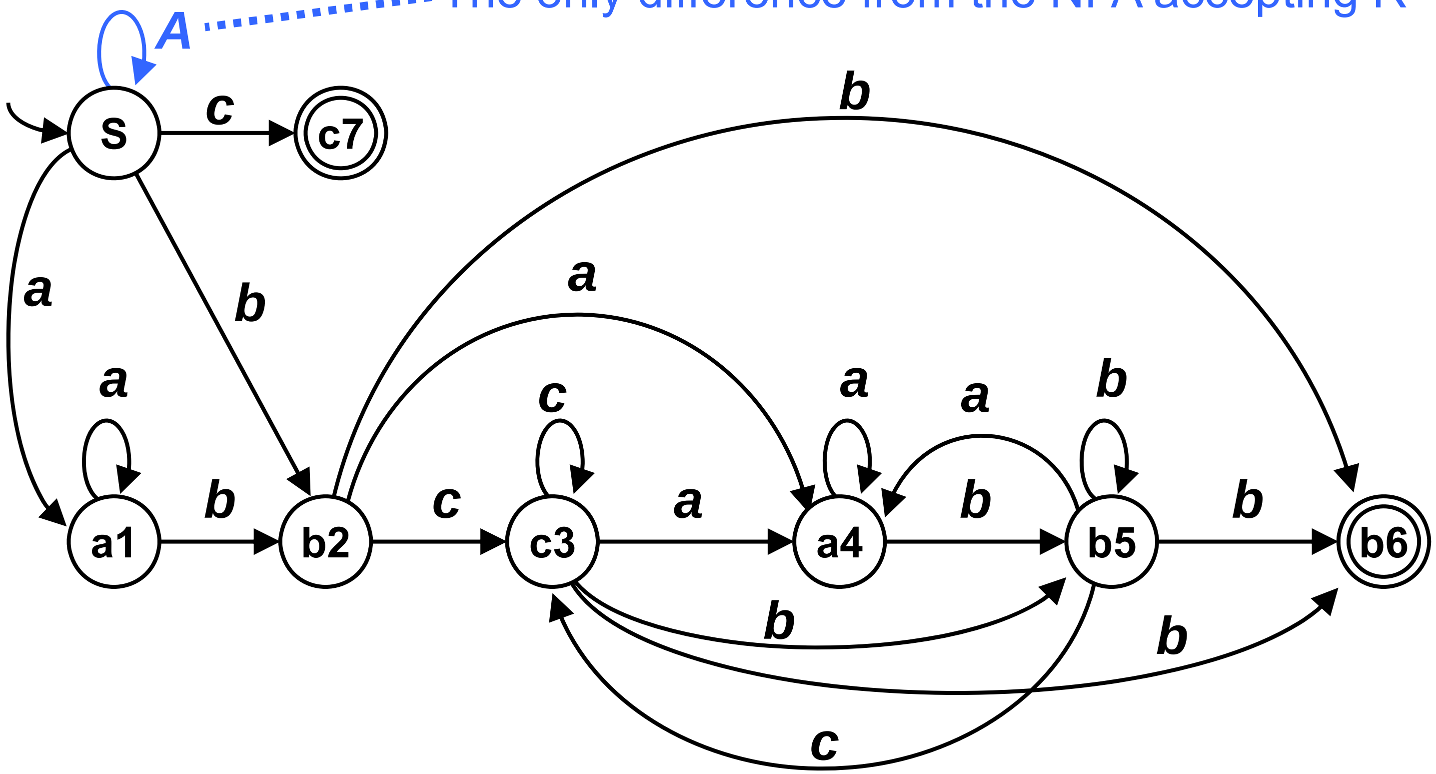
NFA accepts L(R)



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

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

The only difference from the NFA accepting R



Bonus

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

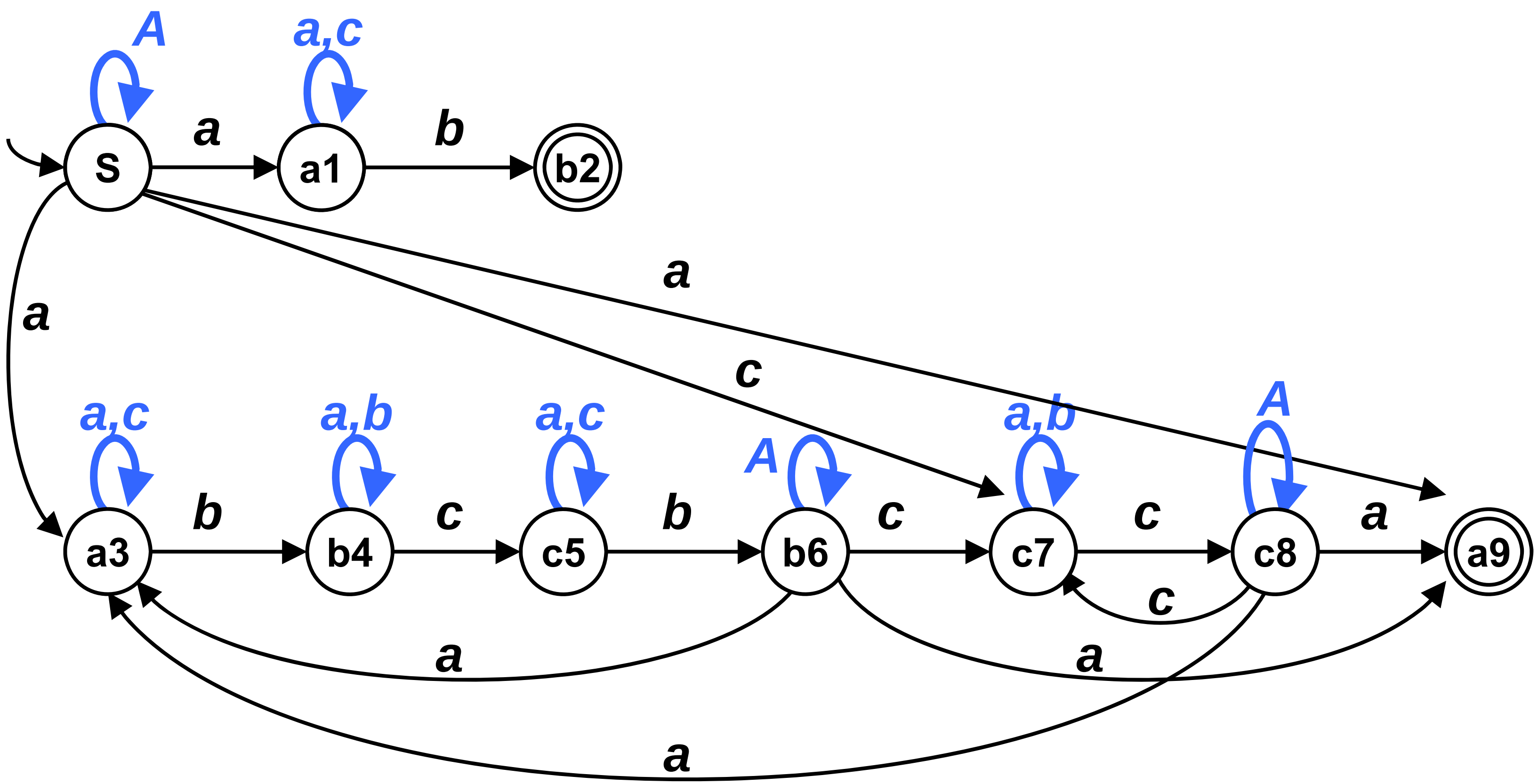
Create NFA acepting L(R)
Add self loops to the states of NFA:

1. Self loop labeled by A (alphabet) to the start state.
2. Self loop labeled A – {x} to each state which outgoing transition(s) are labeled by single x ∈ A. // serves as an "optimized" wait loop
3. Self loop labeled by A to each state which outgoing transition(s) are labeled by more than single symbol from A. // serves as an "usual" wait loop
4. No self loop to all other states. // which have no outgoing loop, final ones

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

R = ab + (abcb + cc)* a

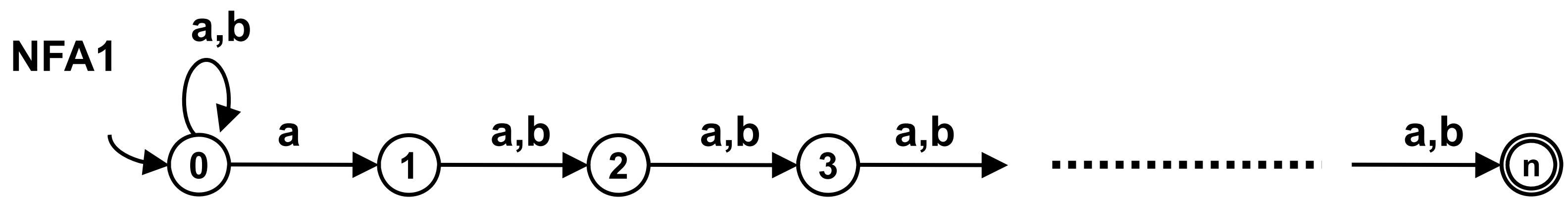
Bonus



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:

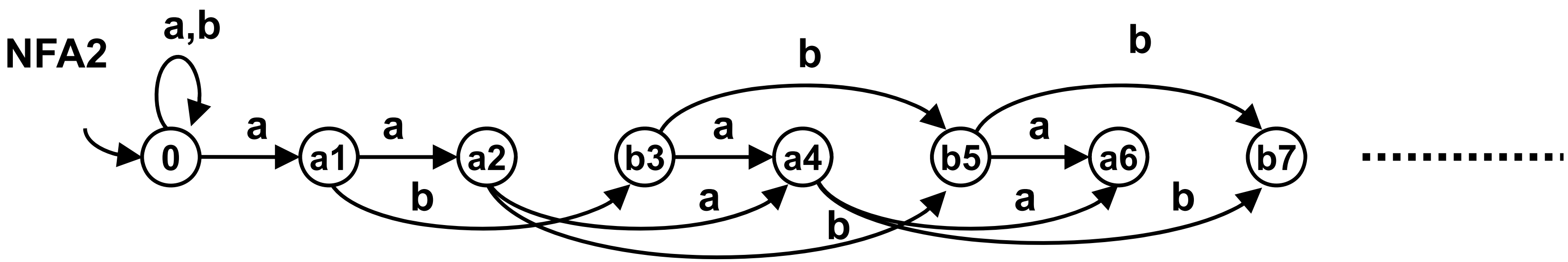
Consider regular expression **R = a(a+b)(a+b)...(a+b)** over alphabet {a, b}.

Text search NFA1 for R



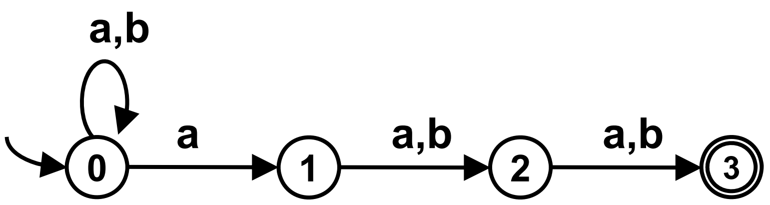
Mystery

Text search NFA2 for R, why not this one?



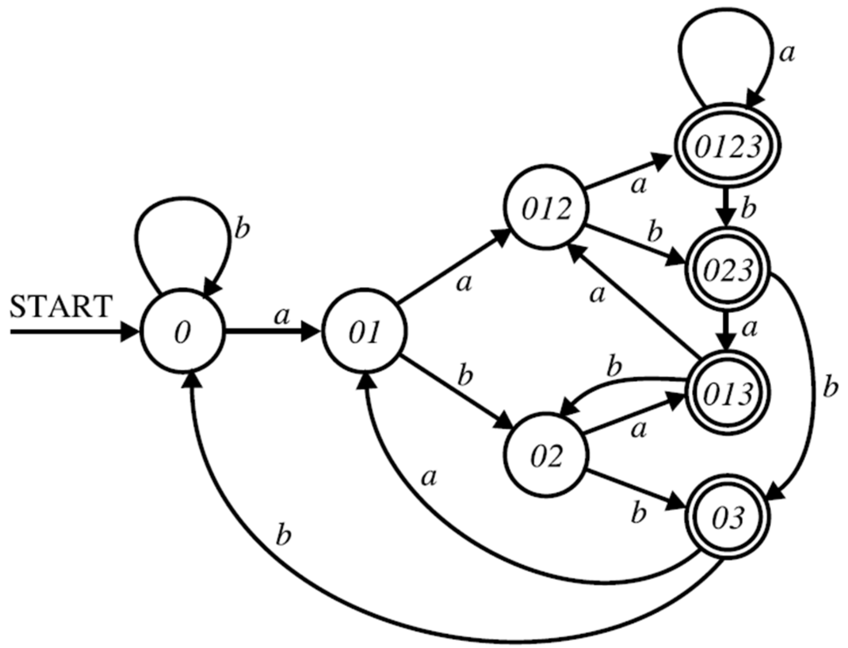
R = a(a+b)(a+b)

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

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.

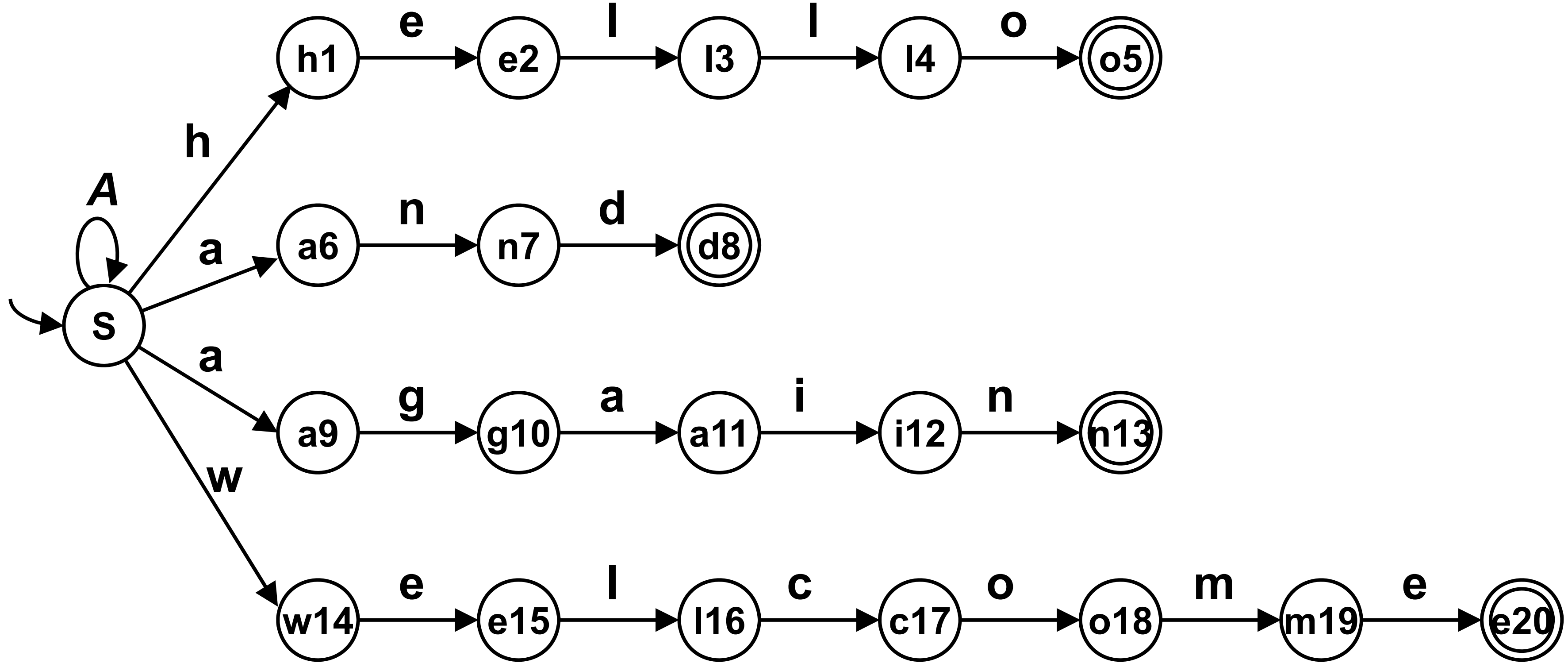
Recycle older knowledge

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.

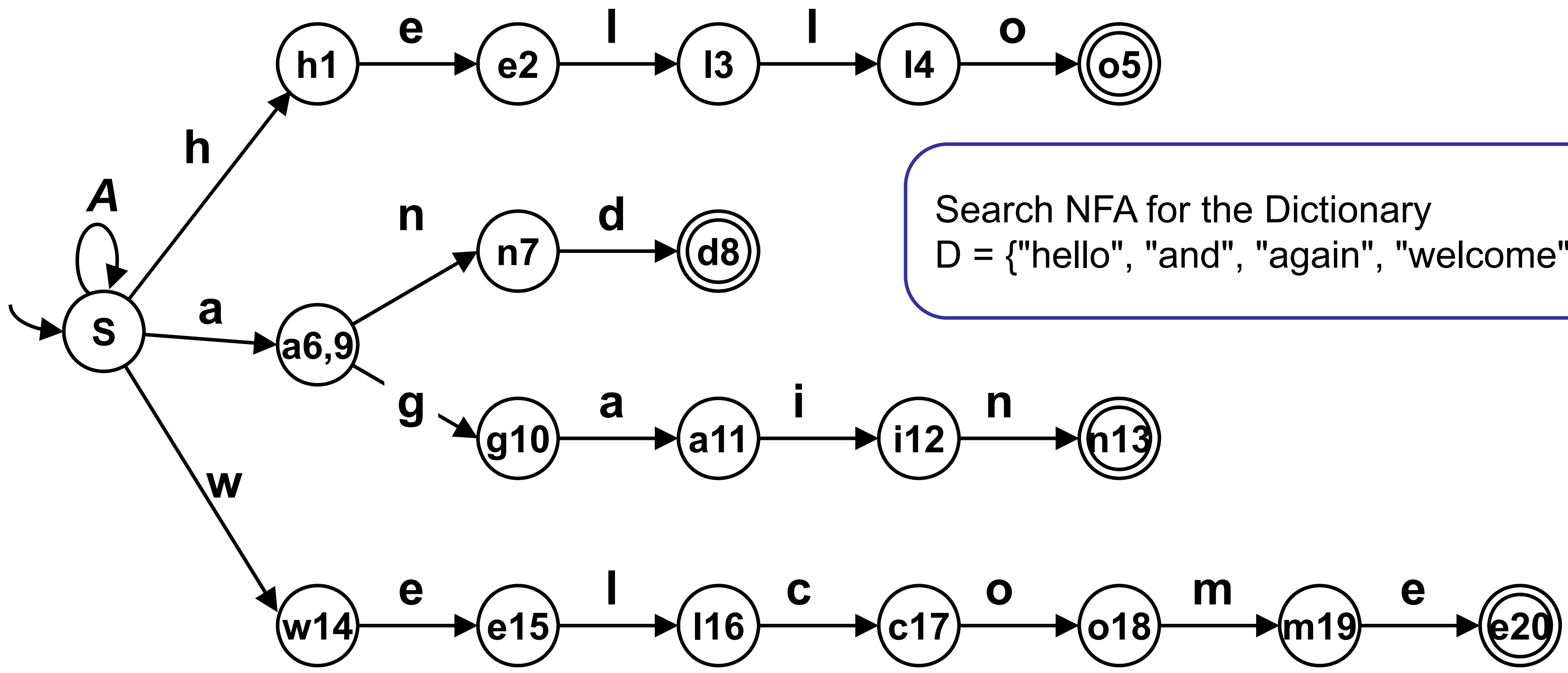
Example

Alphabet A = {a, c, d, e, g, h, i, l, m, n, o , w}
Dictionary D = {"hello", "and", "again", "welcome"}
Regular expression for D: hello+and+again+welcome
NFA for D:

Search NFA for the Dictionary D = {"hello", "and", "again", "welcome"}



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. BFS might be useful in it.



Search NFA for the Dictionary
D = {"hello", "and", "again", "welcome"}

Effectivity

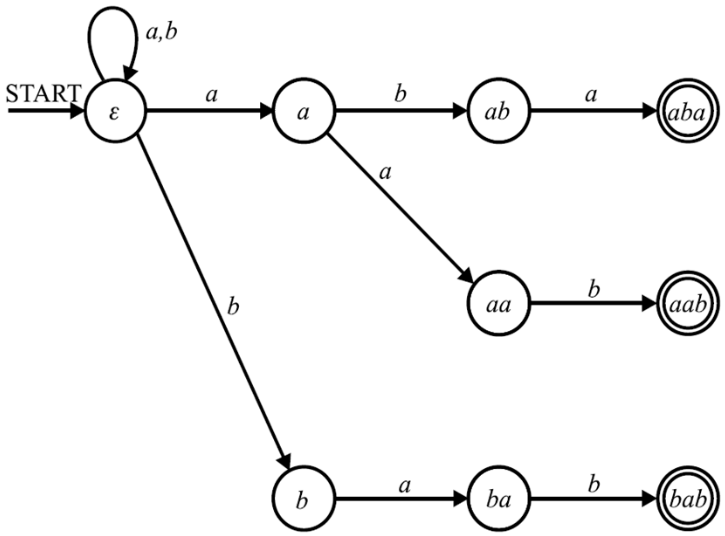
Dictionary NFA constructed as above has useful property:

Transforming this NFA to DFA does not increase number of states.

Another example

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

Before



After