Regular Languages and Finite State Automata

Data structures and algorithms for Computational Linguistics III

Çağrı Çöltekin
ccoltekin@sfs.uni-tuebingen.de

University of Tübingen
Seminar für Sprachwissenschaft

Winter Semester 2018–2019

Why study finite-state automata?

- Unlike some of the abstract machines we discussed, finite-state automata are efficient models of computation
- There are many applications
  - Electronic circuit design
  - Workflow management
  - Games
  - Pattern matching
  - …

  But More importantly ;)
  - Tokenization, stemming
  - Morphological analysis
  - Shallow parsing/chunking
  - …

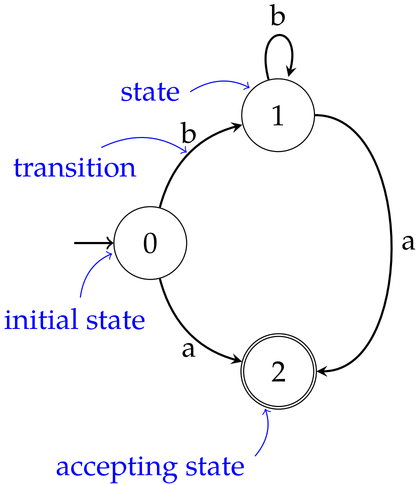
Finite-state automata (FSA)

- A finite-state machine is in one of a finite-number of states in a given time
- The machine changes its sated based on its input
- Every regular language is generated/recognized by an FSA
- Every FSA generates/recognizes a regular language
- Two flavors:
  - *Deterministic finite automata* (DFA)
  - *Non-deterministic finite automata* (NFA)

  Note: the NFA is a superset of DFA.

DFA as a graph

- States are represented as nodes
- Transitions are shown by the edges, labeled with symbols from an alphabet
- One of the states is marked as the *initial state*
- Some states are accepting states



DFA: formal definition

Formally, a finite state automaton, M, is a tuple $(\Sigma, Q, q_0, F, \Delta)$ with

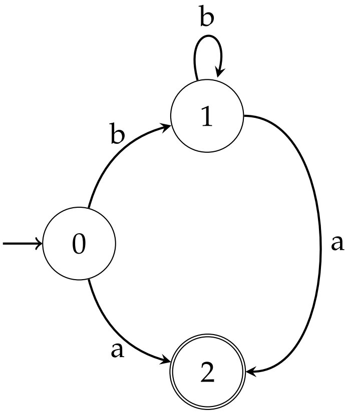
- Σ is the alphabet, a finite set of symbols
- Q a finite set of states
- q_0 is the start state, $q_0 \in Q$
- F is the set of final states, $F \subseteq Q$
- Δ is a function that takes a state and a symbol in the alphabet, and returns another state ($\Delta : Q \times \Sigma \rightarrow Q$)

At any given time, for any input,
a DFA has a single well-defined action to take.

DFA: formal definition

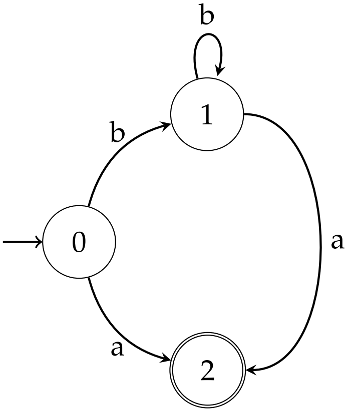
an example

$$
\begin{aligned}
\Sigma &= \{a, b\} \\
Q &= \{q_0, q_1, q_2\} \\
q_0 &= q_0 \\
F &= \{q_2\} \\
\Delta &= \{(q_0, a) \to q_2, \\
&\quad (q_0, b) \to q_1, \\
&\quad (q_1, a) \to q_2, \\
&\quad (q_1, b) \to q_1\}
\end{aligned}
$$



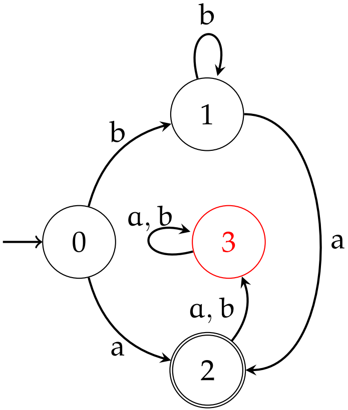
Another note on DFA

- Is this FSA deterministic?



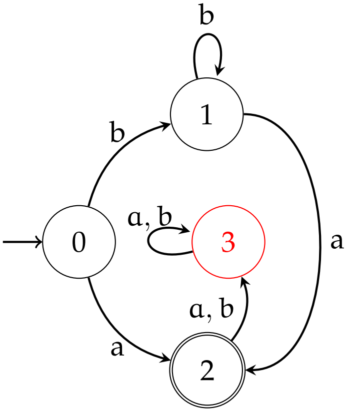
Another note on DFA

- Is this FSA deterministic?
- To make all transitions well-defined, we can add a sink (or error) state



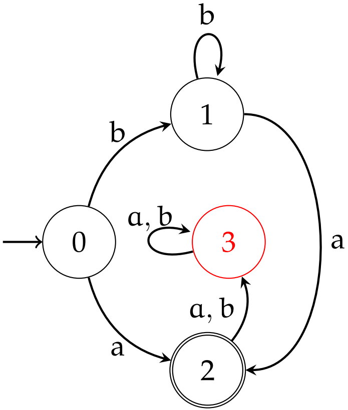
Another note on DFA

- Is this FSA deterministic?
- To make all transitions well-defined, we can add a sink (or error) state
- For brevity, we skip the explicit error state



Another note on DFA

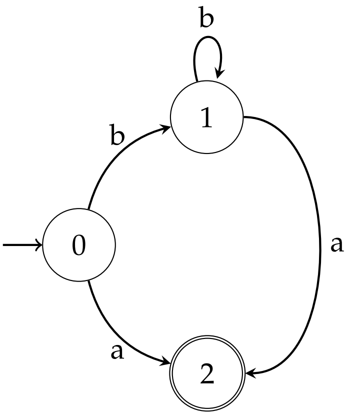
- Is this FSA deterministic?
- To make all transitions well-defined, we can add a sink (or error) state
- For brevity, we skip the explicit error state
  - In that case, when we reach a dead end, recognition fails



DFA: the transition table

transition table

		symbol	
		a	**b**
state	$\rightarrow$**0**	2	1
	1	2	1
	***2**	$\emptyset$	$\emptyset$

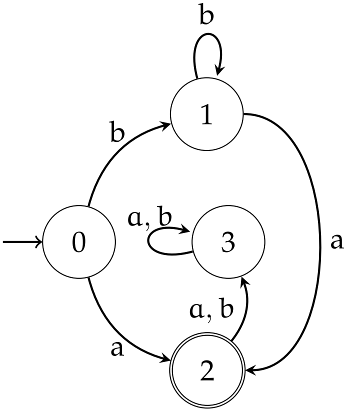


- $\rightarrow$ marks the start state
- * marks the accepting state(s)

DFA: the transition table

transition table

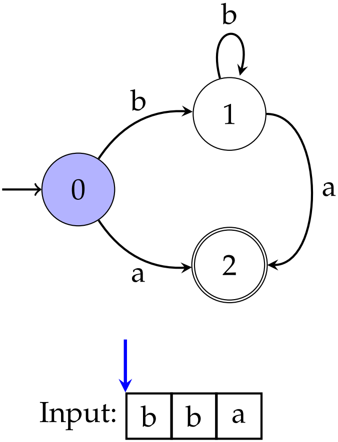
		symbol	
		a	**b**
state	→**0**	2	1
	1	2	1
	***2**	3	3
	3	3	3



- → marks the start state
- * marks the accepting state(s)

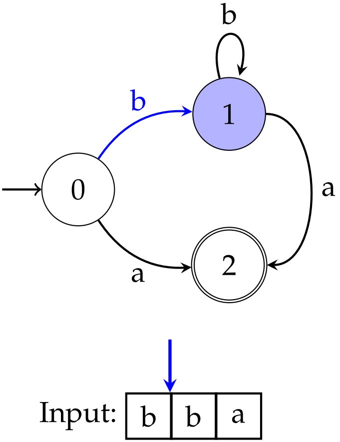
DFA recognition

1. Start at q_0
2. Process an input symbol, move accordingly
3. Accept if in a final state at the end of the input



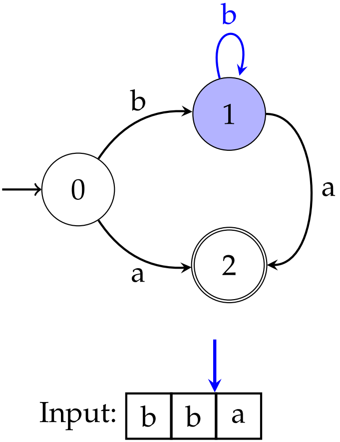
DFA recognition

1. Start at q_0
2. Process an input symbol, move accordingly
3. Accept if in a final state at the end of the input



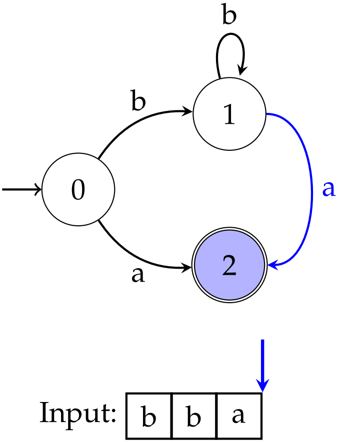
DFA recognition

1. Start at q_0
2. Process an input symbol, move accordingly
3. Accept if in a final state at the end of the input



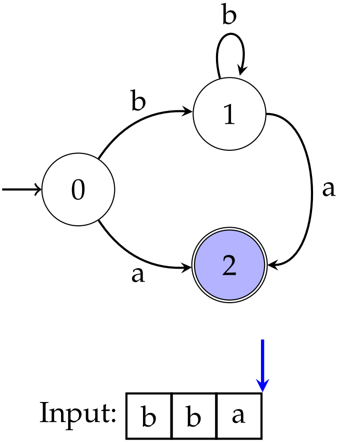
DFA recognition

1. Start at q_0
2. Process an input symbol, move accordingly
3. Accept if in a final state at the end of the input



DFA recognition

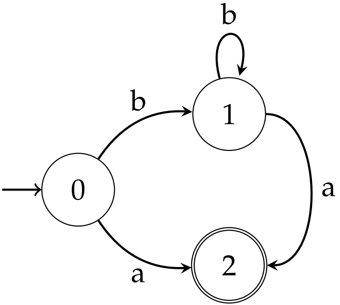
1. Start at q_0
2. Process an input symbol, move accordingly
3. Accept if in a final state at the end of the input



DFA recognition

1. Start at q_0
2. Process an input symbol, move accordingly
3. Accept if in a final state at the end of the input

- What is the complexity of the algorithm?
- How about inputs:
  - bbbb
  - aa

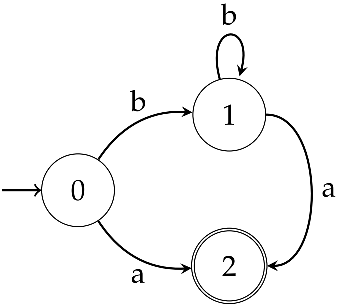


Input:

b	b	a

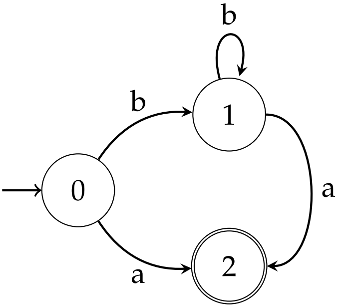
A few questions

- What is the language recognized by this FSA?



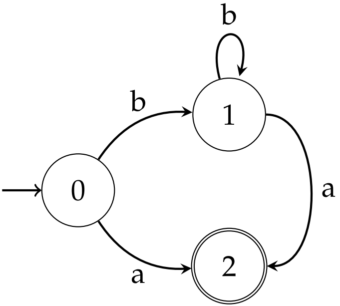
A few questions

- What is the language recognized by this FSA?
- Can you draw a simpler DFA for the same language?



A few questions

- What is the language recognized by this FSA?
- Can you draw a simpler DFA for the same language?
- Draw a DFA recognizing strings with even number of ‘a’s over $\Sigma = \{a, b\}$



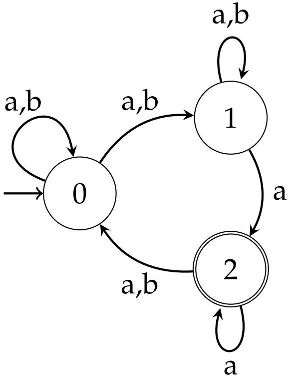
Non-deterministic finite automata

Formal definition

A non-deterministic finite state automaton, M, is a tuple $(\Sigma, Q, q_0, F, \Delta)$ with

- Σ is the alphabet, a finite set of symbols
- Q a finite set of states
- q_0 is the start state, $q_0 \in Q$
- F is the set of final states, $F \subseteq Q$
- Δ is a function from (Q, Σ) to $P(Q)$, power set of Q $(\Delta : Q \times \Sigma \to P(Q))$

An example NFA



transition table

state	*symbol* **a**	**b**
→**0**	0,1	0,1
1	1,2	1
***2**	0,2	0

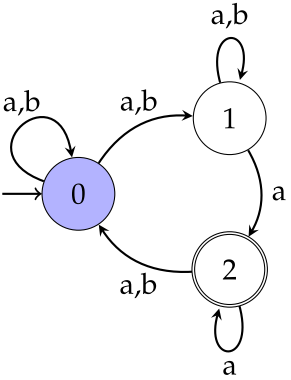
- We have nondeterminism, e.g., if the first input is a, we need choose between states 0 or 1
- Transition table cells have *sets* of states

Dealing with non-determinism

- Follow one of the links, store alternatives, and *backtrack* on failure
- Follow all options in parallel
- Use dynamic programming (e.g., as in chart parsing)

NFA recognition

as search (with backtracking)



Agenda

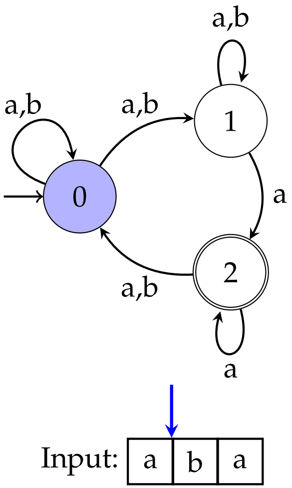
Input:	a	b	a

1. Start at q_0
2. Take the next input, place all possible actions to an *agenda*
3. Get the next action from the agenda, act
4. At the end of input

Accept	if in an accepting state
Reject	not in accepting state & agenda empty
Backtrack	otherwise

NFA recognition

as search (with backtracking)



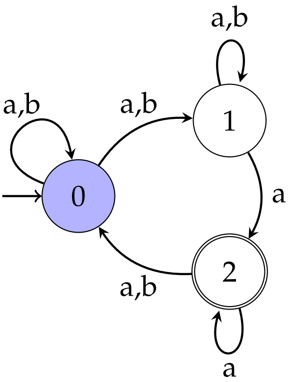
Agenda
$(q_0, 1)$
$(q_1, 1)$

1. Start at q_0
2. Take the next input, place all possible actions to an *agenda*
3. Get the next action from the agenda, act
4. At the end of input

Accept	if in an accepting state
Reject	not in accepting state & agenda empty
Backtrack	otherwise

NFA recognition

as search (with backtracking)



Agenda
$(q_0, 1)$
$(q_1, 1)$

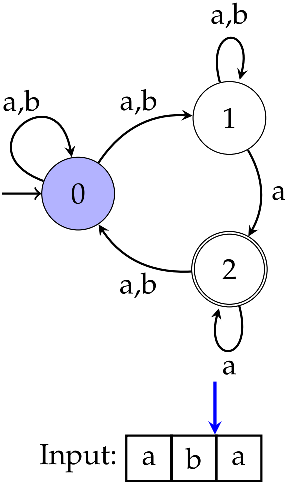
Input: | a | b | a |

1. Start at q_0
2. Take the next input, place all possible actions to an *agenda*
3. Get the next action from the agenda, act
4. At the end of input

Accept if in an accepting state
Reject not in accepting state & agenda empty
Backtrack otherwise

NFA recognition

as search (with backtracking)



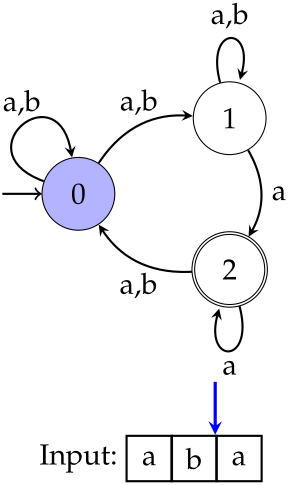
Agenda
$(q_0, 2)$
$(q_1, 2)$
$(q_1, 1)$

1. Start at q_0
2. Take the next input, place all possible actions to an *agenda*
3. Get the next action from the agenda, act
4. At the end of input

Accept	if in an accepting state
Reject	not in accepting state & agenda empty
Backtrack	otherwise

NFA recognition

as search (with backtracking)



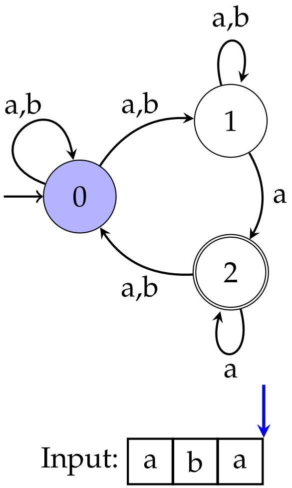
Agenda
$(q_0, 2)$
$(q_1, 2)$
$(q_1, 1)$

1. Start at q_0
2. Take the next input, place all possible actions to an *agenda*
3. Get the next action from the agenda, act
4. At the end of input

Accept if in an accepting state
Reject not in accepting state & agenda empty
Backtrack otherwise

NFA recognition

as search (with backtracking)



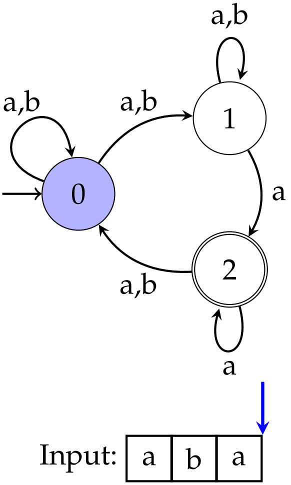
Agenda
$(q_0, 3)$
$(q_1, 3)$
$(q_1, 2)$
$(q_1, 1)$

1. Start at q_0
2. Take the next input, place all possible actions to an *agenda*
3. Get the next action from the agenda, act
4. At the end of input

Accept if in an accepting state
Reject not in accepting state & agenda empty
Backtrack otherwise

NFA recognition

as search (with backtracking)



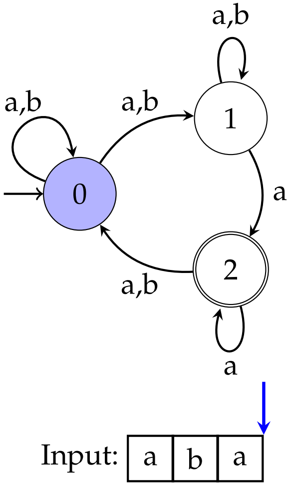
Agenda
$(q_0, 3)$
$(q_1, 3)$
$(q_1, 2)$
$(q_1, 1)$

1. Start at q_0
2. Take the next input, place all possible actions to an *agenda*
3. Get the next action from the agenda, act
4. At the end of input

Accept if in an accepting state
Reject not in accepting state & agenda empty
Backtrack otherwise

NFA recognition

as search (with backtracking)



Agenda
$(q_1, 3)$
$(q_1, 2)$
$(q_1, 1)$

1. Start at q_0
2. Take the next input, place all possible actions to an *agenda*
3. Get the next action from the agenda, act
4. At the end of input

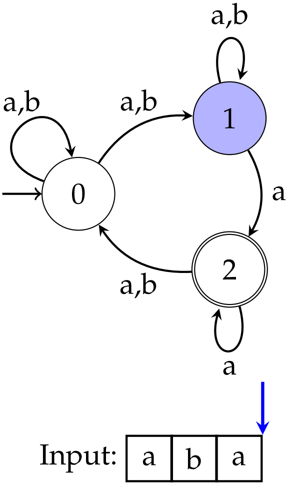
Accept: if in an accepting state

Reject: not in accepting state & agenda empty

Backtrack: otherwise

NFA recognition

as search (with backtracking)



Agenda
$(q_1, 3)$
$(q_1, 2)$
$(q_1, 1)$

1. Start at q_0
2. Take the next input, place all possible actions to an *agenda*
3. Get the next action from the agenda, act
4. At the end of input

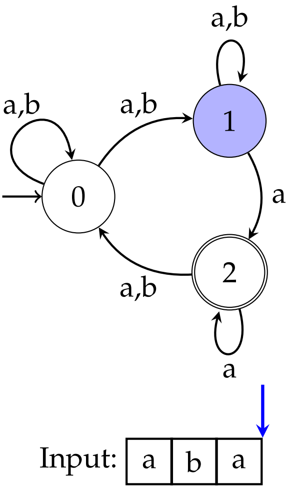
Accept — if in an accepting state

Reject — not in accepting state & agenda empty

Backtrack — otherwise

NFA recognition

as search (with backtracking)



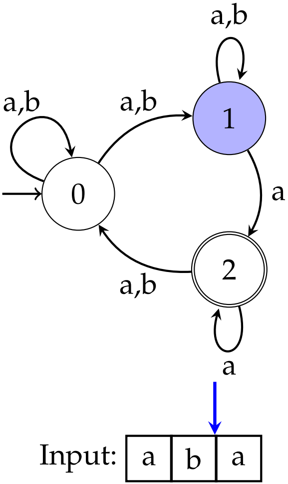
Agenda
$(q_1, 2)$
$(q_1, 1)$

1. Start at q_0
2. Take the next input, place all possible actions to an *agenda*
3. Get the next action from the agenda, act
4. At the end of input

Accept	if in an accepting state
Reject	not in accepting state & agenda empty
Backtrack	otherwise

NFA recognition

as search (with backtracking)



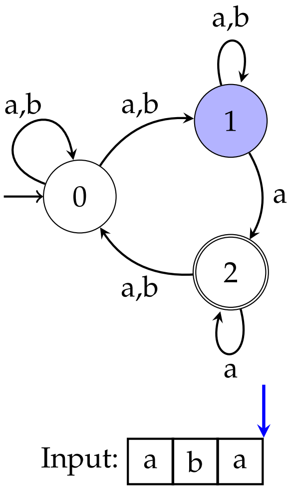
Agenda
$(q_1, 2)$
$(q_1, 1)$

1. Start at q_0
2. Take the next input, place all possible actions to an *agenda*
3. Get the next action from the agenda, act
4. At the end of input

Accept if in an accepting state
Reject not in accepting state & agenda empty
Backtrack otherwise

NFA recognition

as search (with backtracking)



Agenda
$(q_2, 3)$
$(q_1, 3)$
$(q_1, 1)$

1. Start at q_0
2. Take the next input, place all possible actions to an *agenda*
3. Get the next action from the agenda, act
4. At the end of input

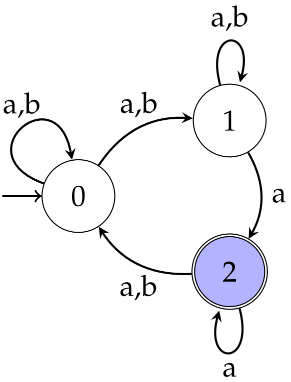
Accept: if in an accepting state

Reject: not in accepting state & agenda empty

Backtrack: otherwise

NFA recognition

as search (with backtracking)



Agenda
$(q_2, 3)$
$(q_1, 3)$
$(q_1, 1)$

Input:

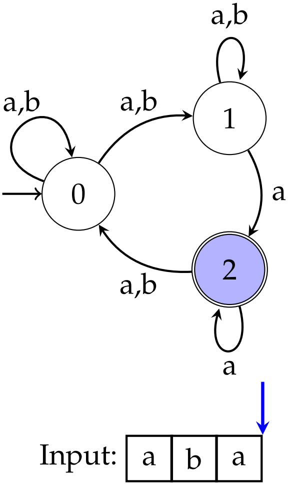
a	b	a

1. Start at q_0
2. Take the next input, place all possible actions to an *agenda*
3. Get the next action from the agenda, act
4. At the end of input

Accept	if in an accepting state
Reject	not in accepting state & agenda empty
Backtrack	otherwise

NFA recognition

as search (with backtracking)



Agenda
$(q_1, 3)$
$(q_1, 1)$

1. Start at q_0
2. Take the next input, place all possible actions to an *agenda*
3. Get the next action from the agenda, act
4. At the end of input

Accept	if in an accepting state
Reject	not in accepting state & agenda empty
Backtrack	otherwise

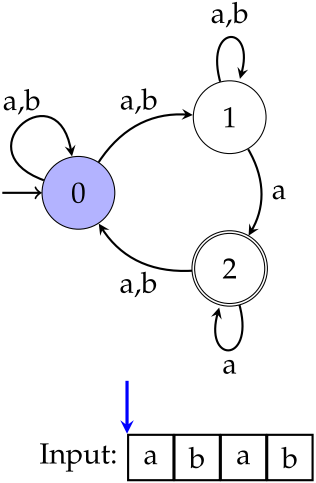
NFA recognition as search

summary

- Worst time complexity is exponential
  - Complexity is worse if we want to enumerate all derivations
- We used a stack as *agenda*, performing a depth-first search
- A queue would result in breadth-first search
- If we have a reasonable heuristic A* search may be an option
- Machine learning methods may also guide finding a fast or the best solution

NFA recognition

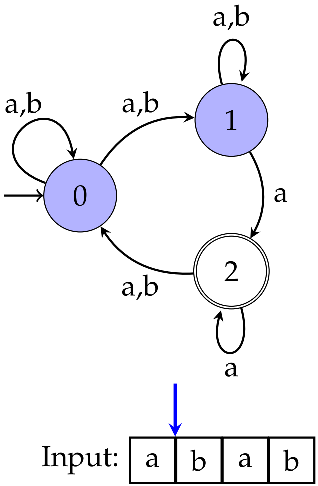
parallel version



1. Start at q_0
2. Take the next input, mark all possible next states
3. If an accepting state is marked at the end of the input, accept

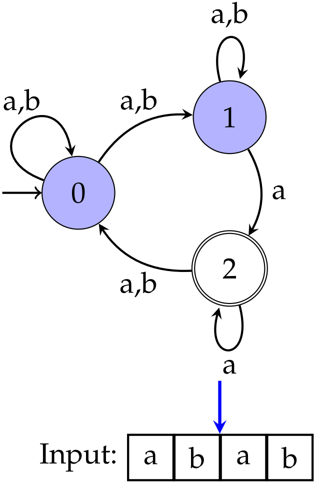
NFA recognition

parallel version



1. Start at q_0
2. Take the next input, mark all possible next states
3. If an accepting state is marked at the end of the input, accept

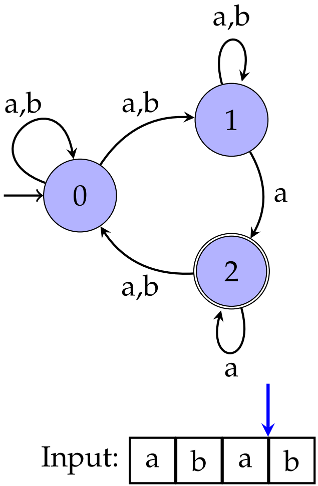
NFA recognition
parallel version



1. Start at q_0
2. Take the next input, mark all possible next states
3. If an accepting state is marked at the end of the input, accept

NFA recognition

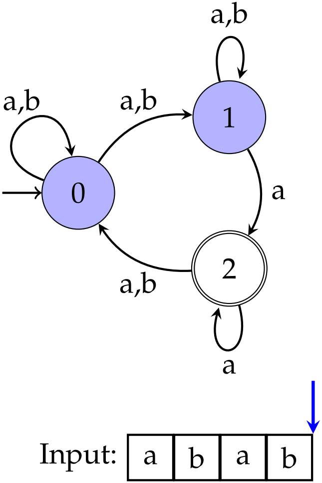
parallel version



1. Start at q_0
2. Take the next input, mark all possible next states
3. If an accepting state is marked at the end of the input, accept

NFA recognition

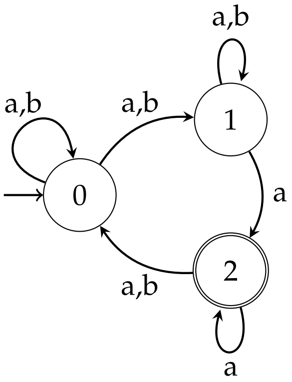
parallel version



1. Start at q_0
2. Take the next input, mark all possible next states
3. If an accepting state is marked at the end of the input, accept

NFA recognition

parallel version



Input: a b a b

1. Start at q_0
2. Take the next input, mark all possible next states
3. If an accepting state is marked at the end of the input, accept

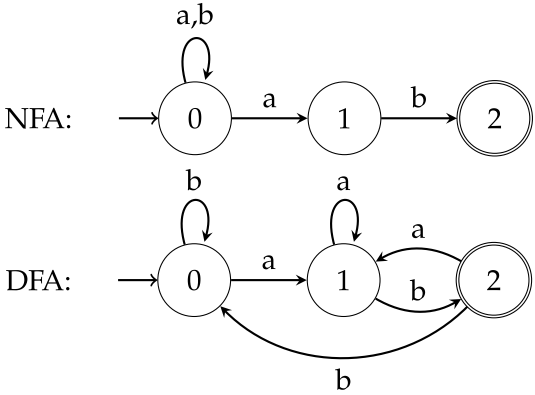
Note: the process is *deterministic*, and *finite-state*.

An exercise

Construct an NFA and a DFA for the language over $\Sigma = \{a, b\}$ where all string end with ab.

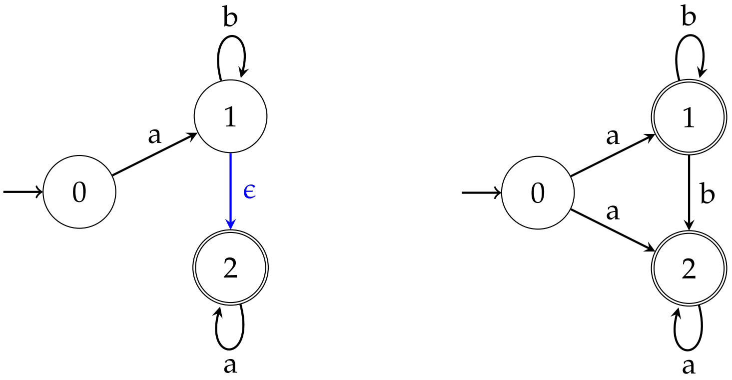
An exercise

Construct an NFA and a DFA for the language over $\Sigma = \{a, b\}$ where all string end with ab.

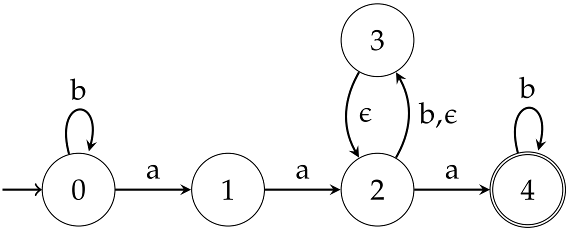


One more complication: ϵ transitions

- An extension of NFA, ϵ-NFA, allows moving without consuming an input symbol, indicated by an ϵ-transition
- Any ϵ-NFA can be converted to an NFA



ϵ-transitions need attention



- How does the (depth-first) NFA recognition algorithm we described earlier on this automaton?

NFA–DFA equivalence

- The language recognized by every NFA is recognized by some DFA
- The set of DFA is a subset of the set of NFA (a DFA is also an NFA)
- The same is true for ϵ-NFA
- All recognize/generate regular languages
- NFA can automatically be converted to the equivalent DFA

Why do we use an NFA then?

- NFA (or ϵ-NFA are often more easy to construct
  - Intuitive for humans
  - Some representations are Easy to convert to NFA rather than DFA, e.g., regular expressions
- NFA may require less memory (fewer states)

Why do we use an NFA then?

- NFA (or ϵ-NFA are often more easy to construct
  - Intuitive for humans
  - Some representations are Easy to convert to NFA rather than DFA, e.g., regular expressions
- NFA may require less memory (fewer states)

A quick exercise

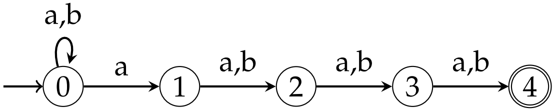
1. Construct (draw) an NFA for the language over $\Sigma = \{a, b\}$, such that 4th symbol from the end is an a

Why do we use an NFA then?

- NFA (or ϵ-NFA are often more easy to construct
  - Intuitive for humans
  - Some representations are Easy to convert to NFA rather than DFA, e.g., regular expressions
- NFA may require less memory (fewer states)

A quick exercise

1. Construct (draw) an NFA for the language over $\Sigma = \{a, b\}$, such that 4th symbol from the end is an a

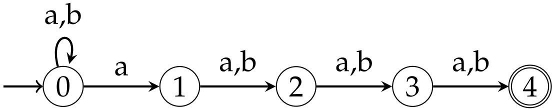


Why do we use an NFA then?

- NFA (or ϵ-NFA are often more easy to construct
  - Intuitive for humans
  - Some representations are Easy to convert to NFA rather than DFA, e.g., regular expressions
- NFA may require less memory (fewer states)

A quick exercise – and a not-so-quick one

1. Construct (draw) an NFA for the language over $\Sigma = \{a, b\}$, such that 4th symbol from the end is an a



2. Construct a DFA for the same language

Determinization

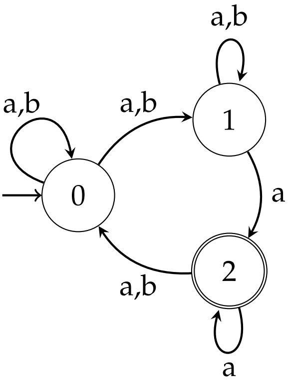
the subset construction

Intuition: remember the parallel NFA recognition. We can consider an NFA being a deterministic machine which is at a set of states at any given time.

- *Subset construction* (sometimes called powerset construction) uses this intuition to convert an NFA to a DFA
- The algorithm can be modified to handle ϵ-transitions (or we can eliminate ϵ's as a pre-processing step)

The subset construction

by example

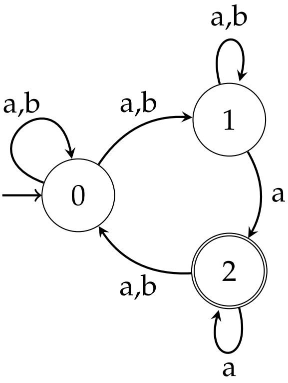


transition table with subsets

	symbol a	b
$\emptyset$	$\emptyset$	$\emptyset$
$\rightarrow\{0\}$	$\{0,1\}$	$\{0,1\}$
$\{1\}$	$\{1,2\}$	$\{1\}$
$*\ \{2\}$	$\{0,2\}$	$\{0\}$
$\{0,1\}$	$\{0,1,2\}$	$\{0,1\}$
$*\ \{0,2\}$	$\{0,1,2\}$	$\{0,1\}$
$*\ \{1,2\}$	$\{0,1,2\}$	$\{0,1\}$
$*\ \{0,1,2\}$	$\{0,1,2\}$	$\{0,1\}$

The subset construction

by example



transition table with subsets

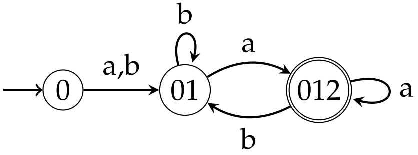
	symbol a	b
~~$\emptyset$~~	~~$\emptyset$~~	~~$\emptyset$~~
$\rightarrow\{0\}$	$\{0,1\}$	$\{0,1\}$
~~$\{1\}$~~	~~$\{1,2\}$~~	~~$\{1\}$~~
~~$*\{2\}$~~	~~$\{0,2\}$~~	~~$\{0\}$~~
$\{0,1\}$	$\{0,1,2\}$	$\{0,1\}$
~~$*\{0,2\}$~~	~~$\{0,1,2\}$~~	~~$\{0,1\}$~~
~~$*\{1,2\}$~~	~~$\{0,1,2\}$~~	~~$\{0,1\}$~~
$*\{0,1,2\}$	$\{0,1,2\}$	$\{0,1\}$

The subset construction

by example: the resulting DFA

transition table without useless/inaccessible states

	symbol a	b
$\rightarrow\{0\}$	$\{0,1\}$	$\{0,1\}$
$\{0,1\}$	$\{0,1,2\}$	$\{0,1\}$
$*\ \{0,1,2\}$	$\{0,1,2\}$	$\{0,1\}$

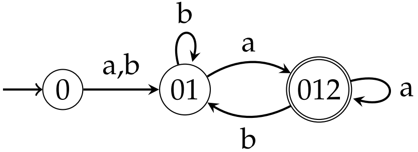


The subset construction

by example: the resulting DFA

transition table without useless/inaccessible states

	symbol **a**	**b**
$\rightarrow\{0\}$	$\{0, 1\}$	$\{0, 1\}$
$\{0, 1\}$	$\{0, 1, 2\}$	$\{0, 1\}$
$*\ \{0, 1, 2\}$	$\{0, 1, 2\}$	$\{0, 1\}$

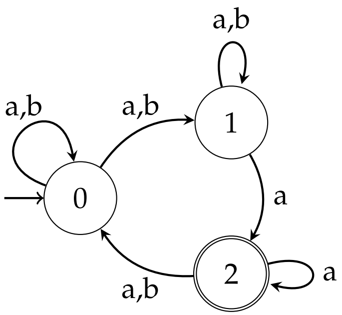


Do you remember the set of states marked during parallel NFA recognition?

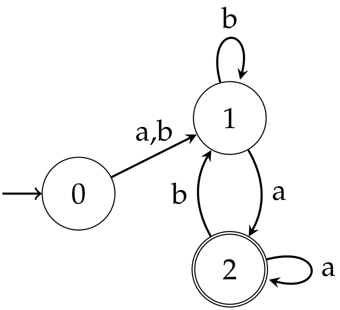
The subset construction

by example: side by side

NFA



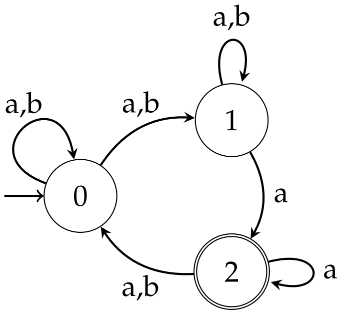
DFA



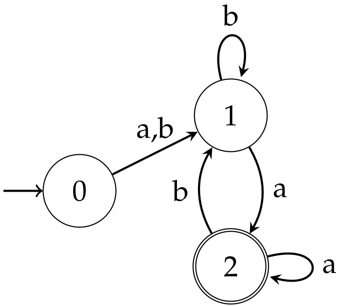
The subset construction

by example: side by side

NFA



DFA

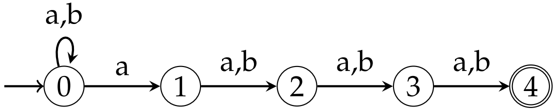


- What language do they recognize?

The subset construction

wrapping up

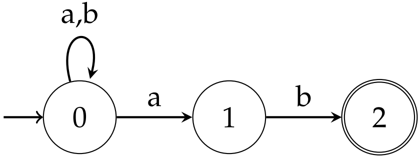
- In worst case, resulting DFA has 2^n nodes
- Worst case is rather rare, in practice number of nodes in an NFA and the converted DFA are often similar
- In practice, we do not need to enumerate all 2^n subsets
- We've already seen a typical problematic case:



- We can also skip the unreachable states during subset construction

Yet another exercise

Determinize the following automaton

NFA: 

Regular languages: definition

A regular grammar is a tuple $G = (\Sigma, N, S, R)$ where

- Σ is an alphabet of terminal symbols
- N are a set of non-terminal symbols
- S is a special 'start' symbol $\in N$
- R is a set of rewrite rules following one of the following patterns ($A, B \in N$, $a \in \Sigma$, ϵ is the empty string)

Left regular

1. $A \rightarrow a$
2. $A \rightarrow Ba$
3. $A \rightarrow \epsilon$

Right regular

1. $A \rightarrow a$
2. $A \rightarrow aB$
3. $A \rightarrow \epsilon$

Regular languages: another definition

A language is regular if there is an FSA that recognizes it

- We denote the language recognized by a finite state automaton M, as $\mathcal{L}(M)$
- The above definition reformulated: if a language L is regular, there is a DFA M, such that $\mathcal{L}(M) = L$
- Remember: any NFA (with or without ϵ transitions) can be converted to a DFA

Some operations on regular languages (and FSA)

$L_1 L_2$ Concatenation of two languages L_1 and L_2: any sentence of L_1 followed by any sentence of L_2

L^* Kleene star of L: L concatenated by itself 0 or more times

L^R Reverse of L: reverse of any string in L

$\overline{L}$ Complement of L: all strings in Σ_L^* except the ones in L ($\Sigma_L^* - L$)

$L_1 \cup L_2$ Union of languages L_1 and L_2: strings that are in any of the languages

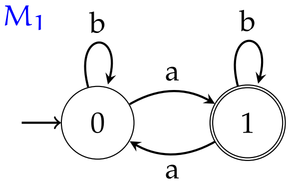
$L_1 \cap L_2$ Intersection of languages L_1 and L_2: strings that are in both languages

Regular languages are closed under all of these operations.

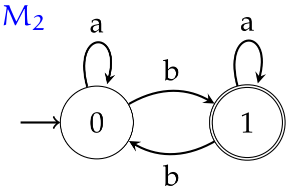
Two example FSA

what languages do they accept?

$L_1 = \mathcal{L}(M_1)$



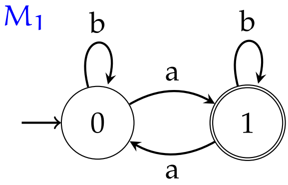
$L_2 = \mathcal{L}(M_2)$



Two example FSA

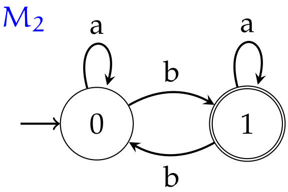
what languages do they accept?

$L_1 = \mathcal{L}(M_1)$



Odd number of a's over $\{a, b\}$.

$L_2 = \mathcal{L}(M_2)$

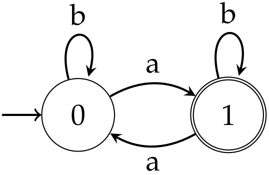


Odd number of b's over $\{a, b\}$.

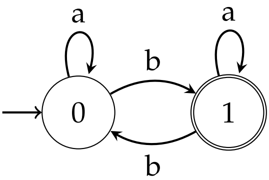
We will use these languages and automata for demonstration.

Concatenation

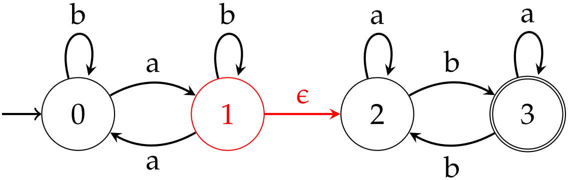
L_1



L_2

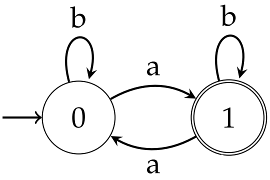


$L_1 L_2$

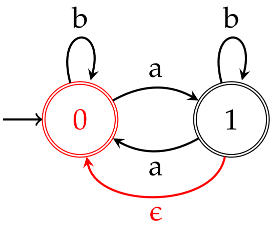


Kleene star

L_1

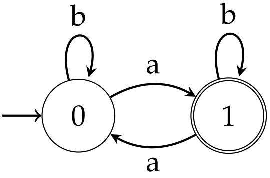


L_1^*

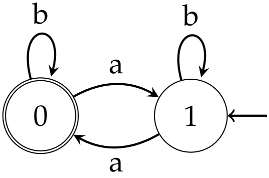


Reversal

L_1

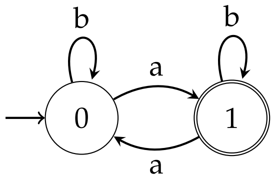


L_1^R

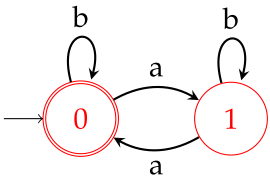


Complement

L_1

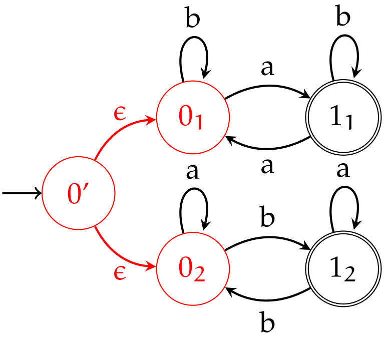


$\overline{L_1}$

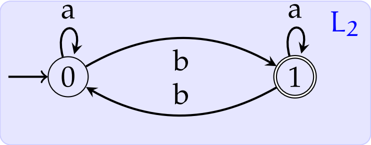


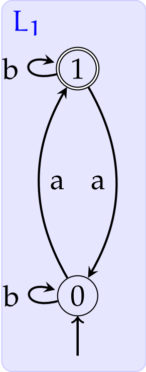
Union

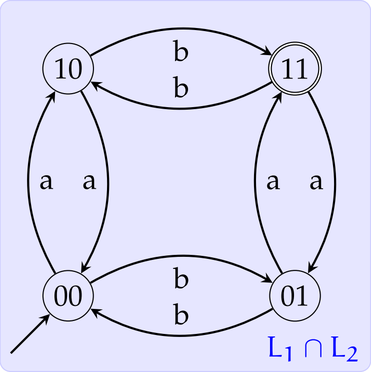
$L_1 \cup L_2$



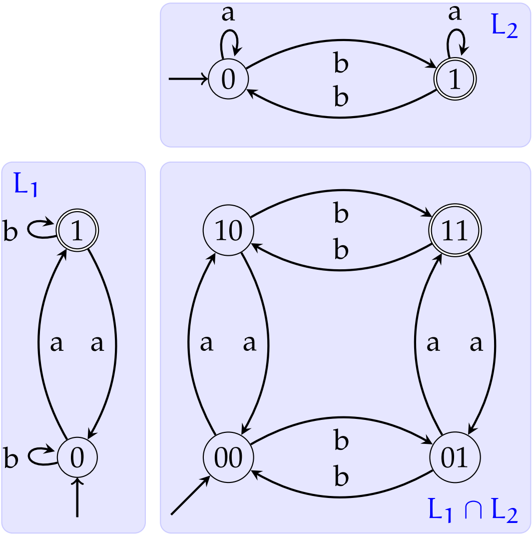
Intersection







Intersection



…or

$$L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$$

Closure properties of regular languages

- Since results of all the operations we studies are FSA:
  Regular languages are closed under
  - Concatenation
  - Kleene star
  - Reversal
  - Complement
  - Union
  - Intersection

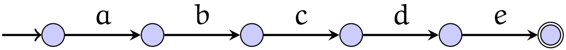
Is a language regular?

— or not

- To show that a language is regular, it is sufficient to find an FSA that recognizes it.
- Showing that a language is *not* regular is more involved
- We will study a method based on *pumping lemma*

Pumping lemma

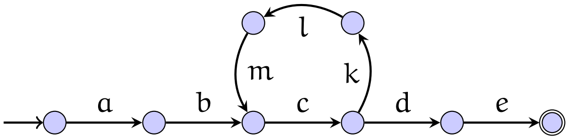
intuition



- What is the length of longest string generated by this FSA?

Pumping lemma

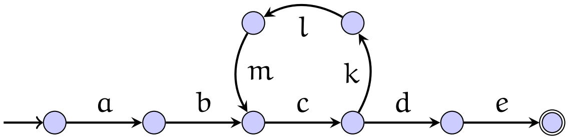
intuition



- What is the length of longest string generated by this FSA?

Pumping lemma

intuition



- What is the length of longest string generated by this FSA?
- Any FSA generating an infinite language has to have a loop (application of recursive rule(s) in the grammar)
- Part of every string longer than some number will include repetition of the same substring (‘cklm’ above)

Pumping lemma

definition

For every regular language L, there exist an integer p such that a string $x \in L$ can be factored as $x = uvw$,

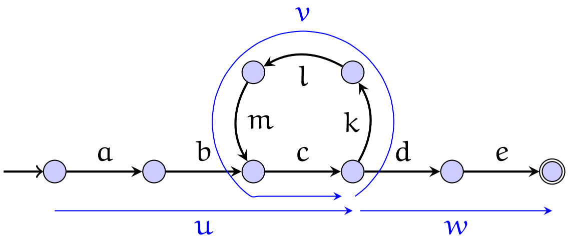
- $uv^i w \in L, \forall i \geqslant 0$
- $v \neq \epsilon$
- $|uv| \leqslant p$

Pumping lemma

definition

For every regular language L, there exist an integer p such that a string $x \in L$ can be factored as $x = uvw$,

- $uv^i w \in L, \forall i \geqslant 0$
- $v \neq \epsilon$
- $|uv| \leqslant p$



How to use pumping lemma

- We use pumping lemma to prove that a language is not regular
- Proof is by contradiction:
  - Assume the language is regular
  - Find a string x in the language, for all splits of $x = uvw$, at least one of the pumping lemma conditions does not hold
    - $uv^iw \in L \; (\forall i \geqslant 0)$
    - $v \neq \epsilon$
    - $|uv| \leqslant p$

Pumping lemma example

prove $L = a^n b^n$ is not regular

- Assume L is regular: there mast be a p such that, if uvw is in the language
  1. $uv^i w \in L \ (\forall i \geqslant 0)$
  2. $v \neq \epsilon$
  3. $|uv| \leqslant p$
- Pick the string $a^p b^p$
- For the sake of example, assume $p = 5$, $x = aaaaabbbbb$
- Three different ways to split

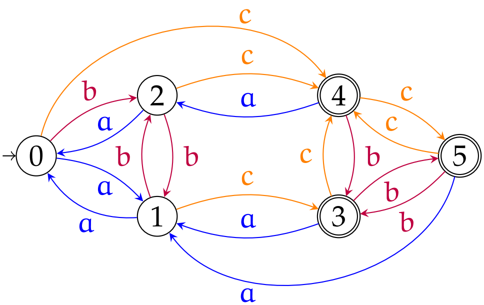
$\underbrace{a}_{u}\underbrace{aaa}_{v}\underbrace{abbbbb}_{w}$	violates 1
$\underbrace{aaaa}_{u}\underbrace{ab}_{v}\underbrace{bbbb}_{w}$	violates 1 & 3
$\underbrace{aaaaab}_{u}\underbrace{bbb}_{v}\underbrace{b}_{w}$	violates 1 & 3

DFA minimization

- For any regular language, there is a unique *minimal* DFA
- By finding the minimal DFA, we can also prove equivalence (or not) of different FSA
- In general the idea is:
  - Throw away unreachable states (easy)
  - Merge equivalent states
- There are two well-known algorithms for minimization:
  - Hopcroft's algorithm: find and eliminate equivalent states by partitioning the set of states
  - Brzozowski's algorithm: 'double reversal'

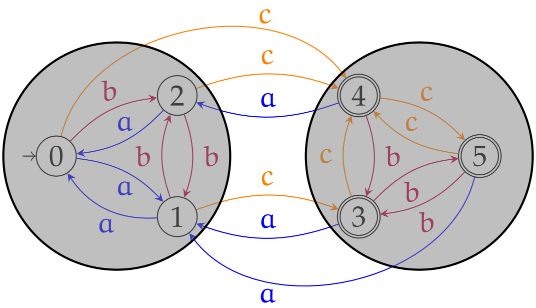
Finding equivalent states

Intuition



Finding equivalent states

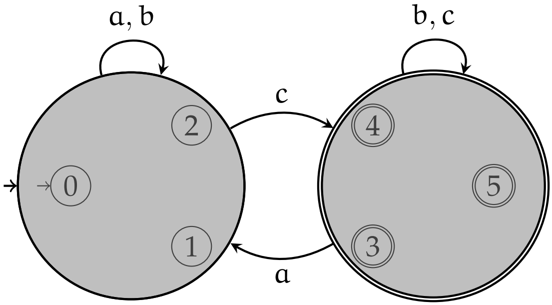
Intuition



The edges leaving the group of nodes are identical.
Their *right languages* are the same.

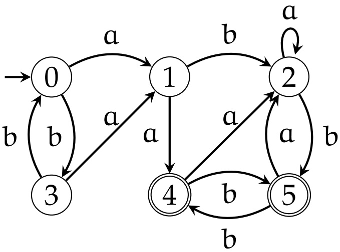
Finding equivalent states

Intuition

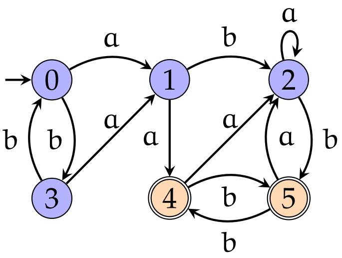


The edges leaving the group of nodes are identical.
Their *right languages* are the same.

Minimization by partitioning



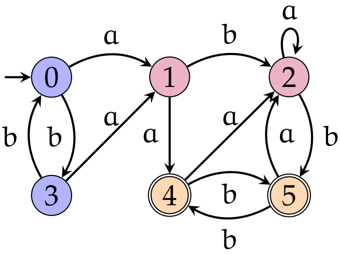
Minimization by partitioning



- Accepting & non-accepting states form a partition

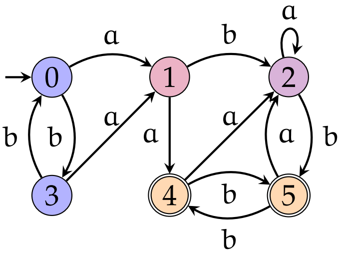
  $Q_2 = \{0, 1, 2, 3\}$, $Q_2 = \{4, 5\}$

Minimization by partitioning



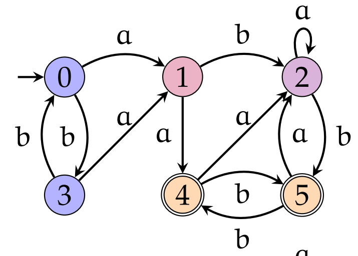
- Accepting & non-accepting states form a partition
  $Q_2 = \{0, 1, 2, 3\}$, $Q_2 = \{4, 5\}$
- if any two nodes go to different sets for any of the symbols split
- $Q_1 = \{0, 1\}$, $Q_3 = \{2, 3\}$, $Q_2 = \{4, 5\}$

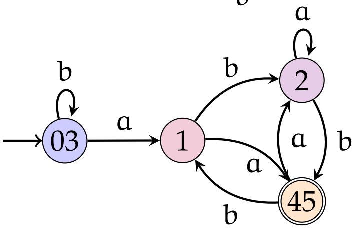
Minimization by partitioning



- Accepting & non-accepting states form a partition
  $Q_2 = \{0, 1, 2, 3\}$, $Q_2 = \{4, 5\}$
- if any two nodes go to different sets for any of the symbols split
- $Q_1 = \{0, 1\}$, $Q_3 = \{2, 3\}$, $Q_2 = \{4, 5\}$
- $Q_1 = \{0, 1\}$, $Q_3 = \{2\}$, $Q_4 = \{3\}$, $Q_2 = \{4, 5\}$

Minimization by partitioning

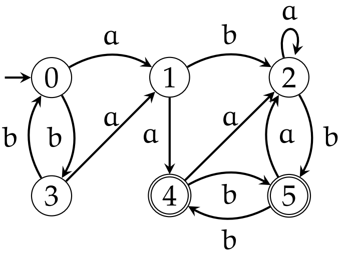




- Accepting & non-accepting states form a partition
  $Q_2 = \{0, 1, 2, 3\}$, $Q_2 = \{4, 5\}$
- if any two nodes go to different sets for any of the symbols split
- $Q_1 = \{0, 1\}$, $Q_3 = \{2, 3\}$, $Q_2 = \{4, 5\}$
- $Q_1 = \{0, 1\}$, $Q_3 = \{2\}$, $Q_4 = \{3\}$, $Q_2 = \{4, 5\}$
- Stop when we cannot split any of the sets

Minimization by partitioning

tabular version

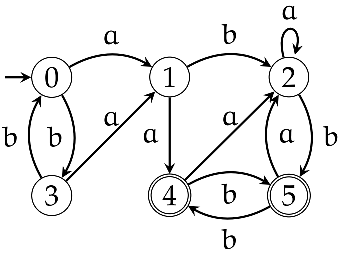


- Create a state-by-state table, mark *distinguishable* pairs: (q_1, q_2) such that $(\Delta(q_1, x), \Delta(q_2, x))$ is a distinguishable pair for any $x \in \Sigma$

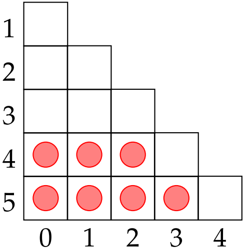
	0	1	2	3	4
1					
2					
3					
4					
5					

Minimization by partitioning

tabular version

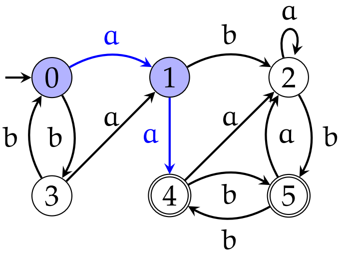


- Create a state-by-state table, mark *distinguishable* pairs: (q_1, q_2) such that $(\Delta(q_1, x), \Delta(q_2, x))$ is a distinguishable pair for any $x \in \Sigma$

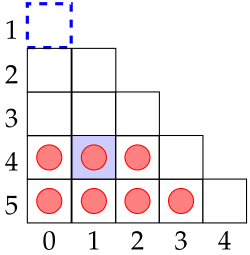


Minimization by partitioning

tabular version

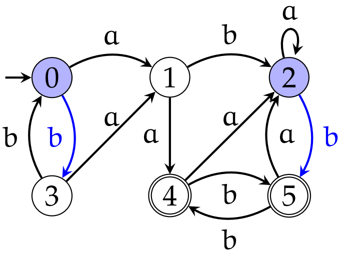


- Create a state-by-state table, mark *distinguishable* pairs: (q_1, q_2) such that $(\Delta(q_1, x), \Delta(q_2, x))$ is a distinguishable pair for any $x \in \Sigma$

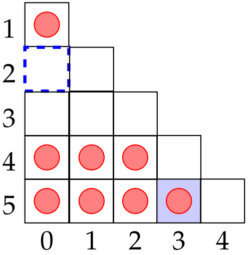


Minimization by partitioning

tabular version

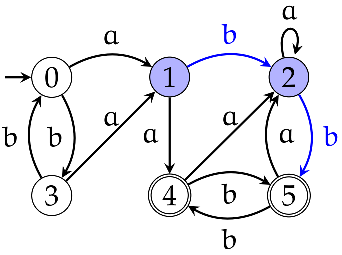


- Create a state-by-state table, mark *distinguishable* pairs: (q_1, q_2) such that $(\Delta(q_1, x), \Delta(q_2, x))$ is a distinguishable pair for any $x \in \Sigma$

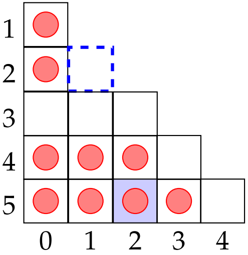


Minimization by partitioning

tabular version

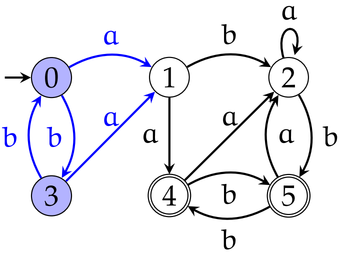


- Create a state-by-state table, mark *distinguishable* pairs: (q_1, q_2) such that $(\Delta(q_1, x), \Delta(q_2, x))$ is a distinguishable pair for any $x \in \Sigma$



Minimization by partitioning

tabular version

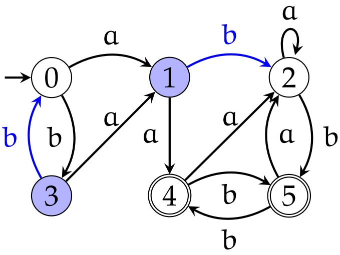


- Create a state-by-state table, mark *distinguishable* pairs: (q_1, q_2) such that $(\Delta(q_1, x), \Delta(q_2, x))$ is a distinguishable pair for any $x \in \Sigma$

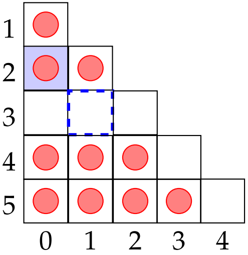
	0	1	2	3	4
1	●				
2	●	●			
3					
4	●	●	●		
5	●	●	●	●	

Minimization by partitioning

tabular version

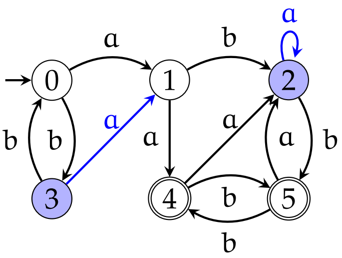


- Create a state-by-state table, mark *distinguishable* pairs: (q_1, q_2) such that $(\Delta(q_1, x), \Delta(q_2, x))$ is a distinguishable pair for any $x \in \Sigma$

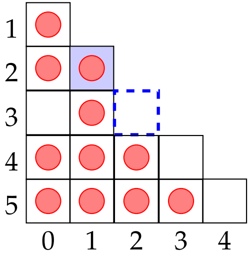


Minimization by partitioning

tabular version

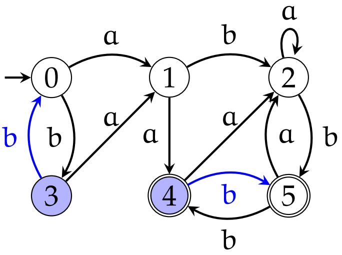


- Create a state-by-state table, mark *distinguishable* pairs: (q_1, q_2) such that $(\Delta(q_1, x), \Delta(q_2, x))$ is a distinguishable pair for any $x \in \Sigma$

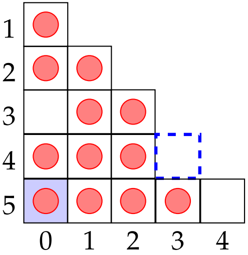


Minimization by partitioning

tabular version

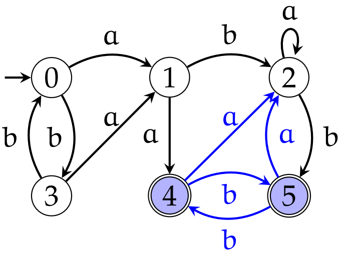


- Create a state-by-state table, mark *distinguishable* pairs: (q_1, q_2) such that $(\Delta(q_1, x), \Delta(q_2, x))$ is a distinguishable pair for any $x \in \Sigma$

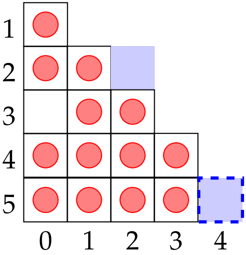


Minimization by partitioning

tabular version

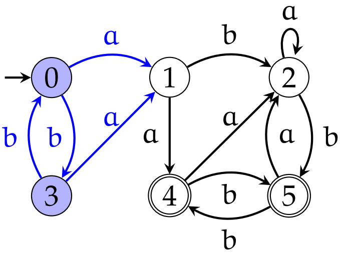


- Create a state-by-state table, mark *distinguishable* pairs: (q_1, q_2) such that $(\Delta(q_1, x), \Delta(q_2, x))$ is a distinguishable pair for any $x \in \Sigma$

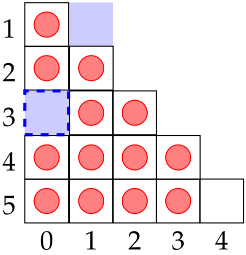


Minimization by partitioning

tabular version

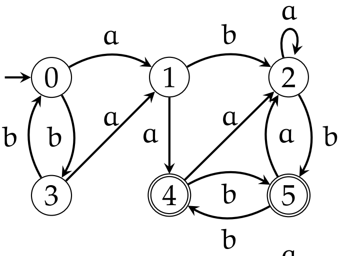


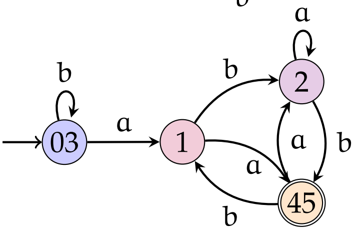
- Create a state-by-state table, mark *distinguishable* pairs: (q_1, q_2) such that $(\Delta(q_1, x), \Delta(q_2, x))$ is a distinguishable pair for any $x \in \Sigma$



Minimization by partitioning

tabular version





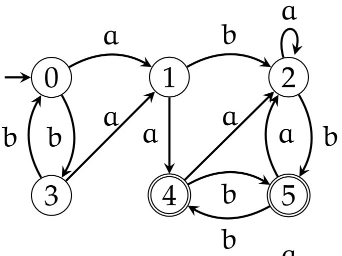
- Create a state-by-state table, mark *distinguishable* pairs: (q_1, q_2) such that $(\Delta(q_1, x), \Delta(q_2, x))$ is a distinguishable pair for any $x \in \Sigma$

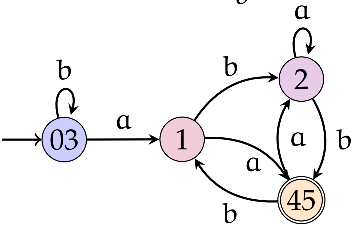
	0	1	2	3	4
1	●				
2	●	●			
3		●	●		
4	●	●	●	●	
5	●	●	●	●	

- Merge indistinguishable states

Minimization by partitioning

tabular version





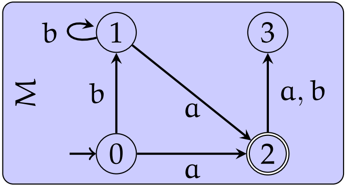
- Create a state-by-state table, mark *distinguishable* pairs: (q_1, q_2) such that $(\Delta(q_1, x), \Delta(q_2, x))$ is a distinguishable pair for any $x \in \Sigma$

	0	1	2	3	4
1	●				
2	●	●			
3		●	●		
4	●	●	●	●	
5	●	●	●	●	

- Merge indistinguishable states
- The algorithm can be improved by choosing which cell to visit carefully

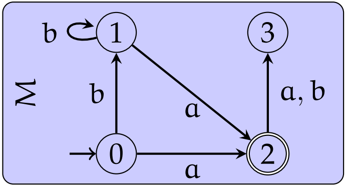
Brzozowski's algorithm

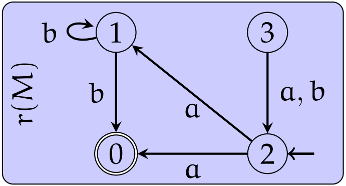
double reverse (r), determinize (d)



Brzozowski's algorithm

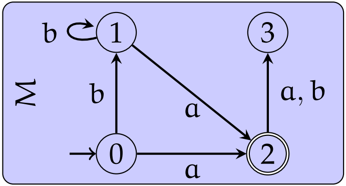
double reverse (r), determinize (d)

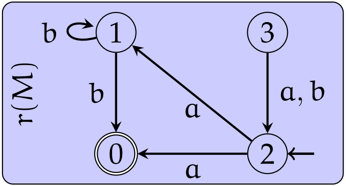


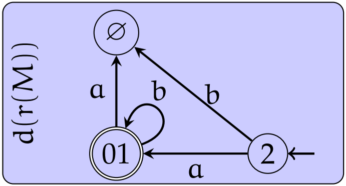


Brzozowski's algorithm

double reverse (r), determinize (d)

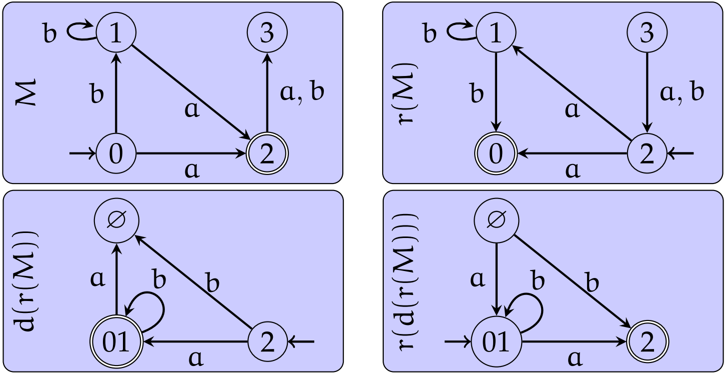






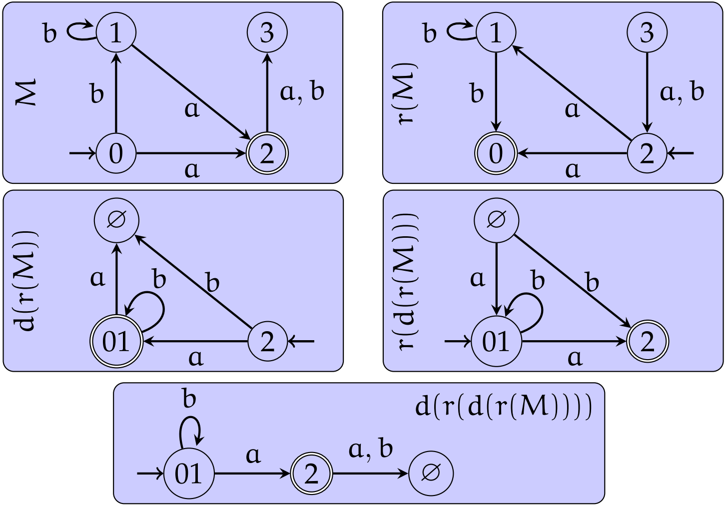
Brzozowski's algorithm

double reverse (r), determinize (d)



Brzozowski's algorithm

double reverse (r), determinize (d)



Minimization algorithms

final remarks

- There are many versions of the 'partitioning' algorithm. General idea is to form equivalence classes based on *right-language* of each state.
- Partitionin algorithm has $O(n \log n)$ complexity
- 'Double reversal' algorithm has exponential worst-time complexity
- Double reversal algorithm can also be used with NFA's (resulting in the minimal equivalent DFA – NFA minimization is intractable)
- In practice, there is no clear winner, different algorithms run faster in different input

Regular expressions

- Another way to specify a regular language (RL) is use of *regular expressions* (RE)
- Every RL can be expressed by a RE, and every RE defines a RL
- A RL x defines a RL $\mathcal{L}(\mathtt{x})$
- Relations between RE and RL

  - $\mathcal{L}(\varnothing) = \varnothing$,
  - $\mathcal{L}(\epsilon) = \epsilon$,
  - $\mathcal{L}(\mathtt{a}) = a$
  - $\mathcal{L}(\mathtt{ab}) = \mathcal{L}(a)\mathcal{L}(b)$
  - $\mathcal{L}(\mathtt{a*}) = \mathcal{L}(a)^*$
  - $\mathcal{L}(\mathtt{a|b}) = \mathcal{L}(a) \cup \mathcal{L}(b)$ (some author use the notation a+b, we will use a|b as in many practical implementations)

  where, $a, b \in \Sigma$, ϵ is empty string, $\varnothing$ is the language that accepts nothing (e.g., $\Sigma^* - \Sigma^*$)
- Note: no stadard complement operation in RE

Regular

some extensions

- Concatenation (`ab`), Kleene star (`a*`) and union (`a|b`) are all we need to define regular expressions
- Parentheses can be used to group the sub-expressions. Otherwise, the priority of the operators as specified above `a|bc* = a|(b(c*))`
- In practice some short-hand notations are common
  - `. = (a`$_1$`|...|a`$_n$`)`, for $\Sigma = \{a_1, \ldots, a_n\}$
  - `a+ = aa*`
  - `[a-c] = (a|b|c)`
  - `[^a-c] = . - (a|b|c)`
  - `\d = (0|1|...|8|9)`
  - …
- And some non-regular extensions, like `(a*)b\1` (sometimes the term *regexp* is used for expressions with non-regular extensions)

Some properties of regular expressions

Kleene algebra

These identities are often used to simplify regular expressions.

- $\epsilon\texttt{u} = \texttt{u}$
- $\varnothing\texttt{u} = \varnothing$
- $\texttt{u(vw)} = \texttt{(uv)w}$
- $\varnothing\texttt{*} = \epsilon$
- $\epsilon\texttt{*} = \epsilon$
- $\texttt{(u*)*} = \texttt{u*}$
- $\texttt{u|v} = \texttt{v|u}$
- $\texttt{u|u} = \texttt{u}$
- $\texttt{u|}\varnothing = \texttt{u}$
- $\texttt{u|}\epsilon = \texttt{u}$
- $\texttt{u|(v|w)} = \texttt{(u|v)|w}$
- $\texttt{u(v|w)} = \texttt{uv|uw}$
- $\texttt{(u|v)*} = \texttt{(u*|v*)*}$

Note: most of these follow from set theory, and some can be derived from others.

Some properties of regular expressions

Kleene algebra

These identities are often used to simplify regular expressions.

- $\epsilon\mathtt{u} = \mathtt{u}$
- $\varnothing\mathtt{u} = \varnothing$
- $\mathtt{u(vw)} = \mathtt{(uv)w}$
- $\varnothing* = \epsilon$
- $\epsilon* = \epsilon$
- $\mathtt{(u*)*} = \mathtt{u*}$
- $\mathtt{u|v} = \mathtt{v|u}$
- $\mathtt{u|u} = \mathtt{u}$
- $\mathtt{u}|\varnothing = \mathtt{u}$
- $\mathtt{u}|\epsilon = \mathtt{u}$
- $\mathtt{u|(v|w)} = \mathtt{(u|v)|w}$
- $\mathtt{u(v|w)} = \mathtt{uv|uw}$
- $\mathtt{(u|v)*} = \mathtt{(u*|v*)*}$

An exercise

Simplify a|ab*

Note: most of these follow from set theory, and some can be derived from others.

Some properties of regular expressions

Kleene algebra

These identities are often used to simplify regular expressions.

- $\epsilon\mathtt{u} = \mathtt{u}$
- $\emptyset\mathtt{u} = \emptyset$
- $\mathtt{u(vw)} = \mathtt{(uv)w}$
- $\emptyset * = \epsilon$
- $\epsilon * = \epsilon$
- $\mathtt{(u*)*} = \mathtt{u*}$
- $\mathtt{u|v} = \mathtt{v|u}$
- $\mathtt{u|u} = \mathtt{u}$
- $\mathtt{u}|\emptyset = \mathtt{u}$
- $\mathtt{u}|\epsilon = \mathtt{u}$
- $\mathtt{u|(v|w)} = \mathtt{(u|v)|w}$
- $\mathtt{u(v|w)} = \mathtt{uv|uw}$
- $\mathtt{(u|v)*} = \mathtt{(u*|v*)*}$

An exercise

Simplify a|ab*

$$\mathtt{a|ab*} \quad = \quad \mathtt{a}\epsilon\mathtt{|ab*}$$

Note: most of these follow from set theory, and some can be derived from others.

Some properties of regular expressions

Kleene algebra

These identities are often used to simplify regular expressions.

- $\epsilon \mathtt{u} = \mathtt{u}$
- $\varnothing \mathtt{u} = \varnothing$
- $\mathtt{u(vw)} = \mathtt{(uv)w}$
- $\varnothing * = \epsilon$
- $\epsilon * = \epsilon$
- $\mathtt{(u*)*} = \mathtt{u*}$
- $\mathtt{u|v} = \mathtt{v|u}$
- $\mathtt{u|u} = \mathtt{u}$
- $\mathtt{u|}\varnothing = \mathtt{u}$
- $\mathtt{u|}\epsilon = \mathtt{u}$
- $\mathtt{u|(v|w)} = \mathtt{(u|v)|w}$
- $\mathtt{u(v|w)} = \mathtt{uv|uw}$
- $\mathtt{(u|v)*} = \mathtt{(u*|v*)*}$

An exercise

Simplify $\mathtt{a|ab*}$

$$\begin{aligned} \mathtt{a|ab*} &= \mathtt{a}\epsilon\mathtt{|ab*} \\ &= \mathtt{a(}\epsilon\mathtt{|b*)} \end{aligned}$$

Note: most of these follow from set theory, and some can be derived from others.

Some properties of regular expressions

Kleene algebra

These identities are often used to simplify regular expressions.

- $\epsilon \mathtt{u} = \mathtt{u}$
- $\varnothing \mathtt{u} = \varnothing$
- $\mathtt{u(vw)} = \mathtt{(uv)w}$
- $\varnothing * = \epsilon$
- $\epsilon * = \epsilon$
- $\mathtt{(u*)*} = \mathtt{u*}$
- $\mathtt{u|v} = \mathtt{v|u}$
- $\mathtt{u|u} = \mathtt{u}$
- $\mathtt{u}|\varnothing = \mathtt{u}$
- $\mathtt{u}|\epsilon = \mathtt{u}$
- $\mathtt{u|(v|w)} = \mathtt{(u|v)|w}$
- $\mathtt{u(v|w)} = \mathtt{uv|uw}$
- $\mathtt{(u|v)*} = \mathtt{(u*|v*)*}$

An exercise

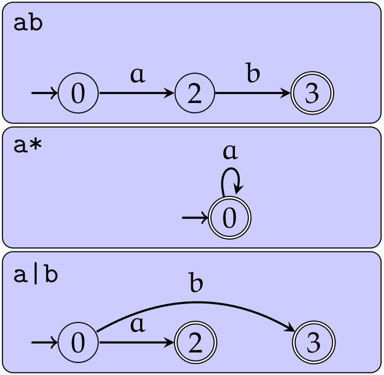
Simplify $\mathtt{a|ab*}$

$$\begin{aligned} \mathtt{a|ab*} &= \mathtt{a}\epsilon\mathtt{|ab*} \\ &= \mathtt{a}(\epsilon\mathtt{|b*}) \\ &= \mathtt{ab*} \end{aligned}$$

Note: most of these follow from set theory, and some can be derived from others.

Converting between RE and FSA

Converting to NFA is easy:



Note the similarity with operations on regular languages discussed earlier.

- For more complex expressions, one can replace the paths for individual symbols with corresponding automata
- Using ϵ transitions may be ease the task
- The reverse conversion (from automata to regular expressions) is also easy:
  - identify the patterns on the left, collapse paths to single transitions with regular expressions

Wrapping up

- FSA and regular expressions express regular languages
- FSA have two flavors: DFA, NFA (or maybe three ϵ-NFA)
- DFA recognition is linear
- Any NFA can be converted to a DFA (with a worst case of exponential increase of nodes)
- Regular languages and FSA are closed under
  - Concatenation
  - Kleene star
  - Complement
  - Reversal
  - Union
  - Intersection
- Every FSA has a unique minimal DFA

Wrapping up

- FSA and regular expressions express regular languages
- FSA have two flavors: DFA, NFA (or maybe three ϵ-NFA)
- DFA recognition is linear
- Any NFA can be converted to a DFA (with a worst case of exponential increase of nodes)
- Regular languages and FSA are closed under
  - Concatenation
  - Kleene star
  - Complement
  - Reversal
  - Union
  - Intersection
- Every FSA has a unique minimal DFA

Next:

- Finite state transducers (FSTs)
- Applications of FSA and FSTs

References / additional reading material

- Hopcroft and Ullman (Chapter 2&3 1979) (and its successive editions) covers (almost) all topics discussed here
- Jurafsky and Martin (Chapter 2 2009)
- Other textbook references include:
  - Sipser (2006)
  - Kozen (2013)

References / additional reading material (cont.)

- Hopcroft, John E. and Jeffrey D. Ullman (1979). *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley Series in Computer Science and Information Processing. Addison-Wesley. ISBN: 9780201029888.
- Jurafsky, Daniel and James H. Martin (2009). *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. second. Pearson Prentice Hall. ISBN: 978-0-13-504196-3. URL: http://web.stanford.edu/~jurafsky/slp3/.
- Kozen, Dexter C. (2013). *Automata and Computability*. Undergraduate Texts in Computer Science. Berlin Heidelberg: Springer.
- Sipser, Michael (2006). *Introduction to the Theory of Computation*. second. Thomson Course Technology. ISBN: 0-534-95097-3.