Finite state automata

Data Structures and Algorithms for Computational Linguistics III (ISCL-BA-07)

Çağrı Çöltekin
ccoltekin@sfs.uni-tuebingen.de

University of Tübingen
Seminar für Sprachwissenschaft

Winter Semester 2024/25

version: c8d9876 @2025-01-08

Why study finite-state automata?

- 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
  - Spell checking
  - Shallow parsing/chunking
  - ...

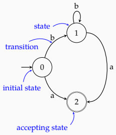
Finite-state automata (FSA)

- A finite-state machine is in one of a finite-number of states in a given time
- The machine changes its state 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.

FSA as a graph

- An FSA is a directed graph
- States are represented as nodes
- Transitions are labeled edges
- One of the states is the *initial state*
- Some states are accepting states



Languages and automata

- Recognizing strings from a language defined by a grammar is a fundamental question in computer science
- The efficiency of computation, and required properties of computing device depends on the grammar (and the language)
- A well-known hierarchy of grammars both in computer science and linguistics is the *Chomsky hierarchy*
- Each grammar in the Chomsky hierarchy corresponds to an abstract computing device (an automaton)
- The class of *regular grammars* are the class that corresponds to *finite state automata*

How to describe a language?

Formal grammars

A formal *grammar* is a finite specification of a (formal) language.

- We consider languages as sets of strings, for a finite language, we can (conceivably) list all strings
- How to define an infinite language?
  - Is the definition $\{ba, baa, baaa, baaaa, \ldots\}$ 'formal enough'?
  - Using regular expressions, we can define it as baa^*
  - We will introduce a more general method for defining languages

Phrase structure grammars

- A phrase structure grammar is a generative device
- If a given string can be generated by the grammar, the string is in the language
- The grammar generates *all* and the *only* strings that are valid in the language
- A phrase structure grammar has the following components
  - Σ A set of *terminal* symbols
  - N A set of *non-terminal* symbols
  - $S \in N$ A special non-terminal, called the start symbol
  - R A set of *rewrite rules* or *production rules* of the form:

    $$\alpha \rightarrow \beta$$

    which means that the sequence α can be rewritten as β (both α and β are sequences of terminal and non-terminal symbols)
- The strings in the language of the grammar is those that can be derived from S using the rewrite operations

Chomsky hierarchy and automata

<table>
<tr><th><i>Grammar class</i></th><th><i>Rules</i></th><th><i>Automata</i></th></tr>
<tr><td>Unrestricted grammars</td><td>$\alpha \rightarrow \beta$</td><td>Turing machines</td></tr>
<tr><td>Context-sensitive grammars</td><td>$\alpha A \beta \rightarrow \alpha \gamma \beta$</td><td>Linear-bounded automata</td></tr>
<tr><td>Context-free grammars</td><td>$A \rightarrow \alpha$</td><td>Pushdown automata</td></tr>
<tr><td>Regular grammars</td><td>$A \rightarrow a$ $A \rightarrow a$
$A \rightarrow aB$ $A \rightarrow B\,a$</td><td>Finite state automata</td></tr>
</table>

Regular grammars: 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	Right regular
1. $A \rightarrow a$	1. $A \rightarrow a$
2. $A \rightarrow Ba$	2. $A \rightarrow aB$
3. $A \rightarrow \epsilon$	3. $A \rightarrow \epsilon$

Regular languages: some properties/operations

- $\mathcal{L}_1\mathcal{L}_2$ Concatenation of two languages $\mathcal{L}_1$ and $\mathcal{L}_2$: any sentence of $\mathcal{L}_1$ followed by any sentence of $\mathcal{L}_2$
- $\mathcal{L}^*$ Kleene star of $\mathcal{L}$: $\mathcal{L}$ concatenated with itself 0 or more times
- $\mathcal{L}^R$ Reverse of $\mathcal{L}$: reverse of any string in $\mathcal{L}$
- $\overline{\mathcal{L}}$ Complement of $\mathcal{L}$: all strings in $\Sigma^*_{\mathcal{L}}$ except the ones in $\mathcal{L}$ $(\Sigma^*_{\mathcal{L}} - \mathcal{L})$
- $\mathcal{L}_1 \cup \mathcal{L}_2$ Union of languages $\mathcal{L}_1$ and $\mathcal{L}_2$: strings that are in any of the languages
- $\mathcal{L}_1 \cap \mathcal{L}_2$ Intersection of languages $\mathcal{L}_1$ and $\mathcal{L}_2$: strings that are in both languages

Regular languages are closed under all of these operations.

Three ways to define a regular language

- A language is regular if there is regular grammar that generates/recognizes it
- A language is regular if there is an FSA that generates/recognizes it
- A language is regular if we can define a regular expressions for the language

DFA: formal definition

Formally, a 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 that takes a state and a symbol in the alphabet, and returns another state $(\Delta : Q \times \Sigma \rightarrow Q)$

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

DFA: formal definition

an example

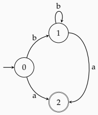
$\Sigma = \{a, b\}$

$Q = \{q_0, q_1, q_2\}$

$q_0 = q_0$

$F = \{q_2\}$

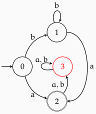
$\Delta = \{(q_0, a) \to q_2, \quad (q_0, b) \to q_1,$
$\quad (q_1, a) \to q_2, \quad (q_1, b) \to q_1\}$



Another note on DFA

error or sink state

- 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



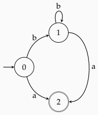
DFA: the transition table

transition table

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

→ marks the start state

* marks the accepting state(s)



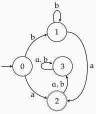
DFA: the transition table

transition table

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

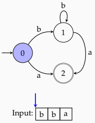
→ marks the start state

* marks the accepting state(s)



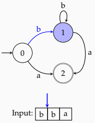
DFA recognition

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



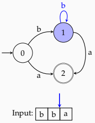
DFA recognition

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



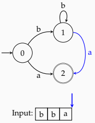
DFA recognition

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



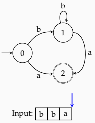
DFA recognition

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

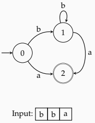
1. Start at q_0
2. Process an input symbol, move accordingly
3. Accept if in a final state at the end of the input



DFA recognition

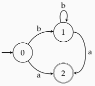
1. Start at q_0
2. Process an input symbol, move accordingly
3. Accept if in a final state at the end of the input

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



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

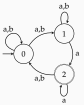
Q a finite set of states

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

		symbol	
		a	b
state	→**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 to choose between states 0 or 1
- Transition table cells have *sets* of states

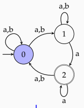
Dealing with non-determinism

- Follow one of the links, store alternatives, and *backtrack* on failure
- Follow all options in parallel

NFA recognition

as search (with backtracking)

Agenda



Input: a b a

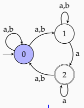
1. Start at q_0
2. Take the next input, place all possible actions to an *agenda*
3. Get the next action from the agenda, act
4. At the end of input

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

NFA recognition

as search (with backtracking)

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



Input: a b a

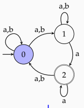
1. Start at q_0
2. Take the next input, place all possible actions to an *agenda*
3. Get the next action from the agenda, act
4. At the end of input

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

NFA recognition

as search (with backtracking)

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



Input: a b a

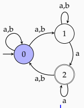
1. Start at q_0
2. Take the next input, place all possible actions to an *agenda*
3. Get the next action from the agenda, act
4. At the end of input

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

NFA recognition

as search (with backtracking)

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



Input: a b a

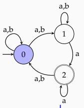
1. Start at q_0
2. Take the next input, place all possible actions to an *agenda*
3. Get the next action from the agenda, act
4. At the end of input

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

NFA recognition

as search (with backtracking)

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



Input: a b a

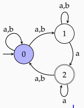
1. Start at q_0
2. Take the next input, place all possible actions to an *agenda*
3. Get the next action from the agenda, act
4. At the end of input

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

NFA recognition

as search (with backtracking)

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



Input: a b a

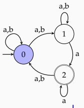
1. Start at q_0
2. Take the next input, place all possible actions to an *agenda*
3. Get the next action from the agenda, act
4. At the end of input

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

NFA recognition

as search (with backtracking)

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



Input: a b a

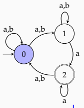
1. Start at q_0
2. Take the next input, place all possible actions to an *agenda*
3. Get the next action from the agenda, act
4. At the end of input

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

NFA recognition

as search (with backtracking)

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



Input: a b a

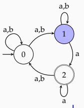
1. Start at q_0
2. Take the next input, place all possible actions to an *agenda*
3. Get the next action from the agenda, act
4. At the end of input

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

NFA recognition

as search (with backtracking)

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



Input: a b a

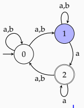
1. Start at q_0
2. Take the next input, place all possible actions to an *agenda*
3. Get the next action from the agenda, act
4. At the end of input

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

NFA recognition

as search (with backtracking)

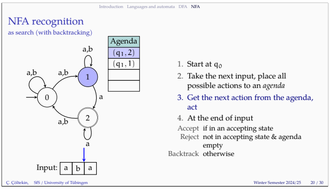
Agenda
$(q_1, 2)$
$(q_1, 1)$



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



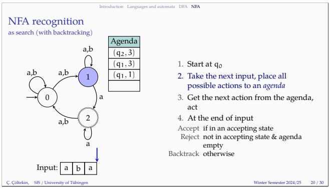
NFA recognition
as search (with backtracking)

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

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

Input: a b a



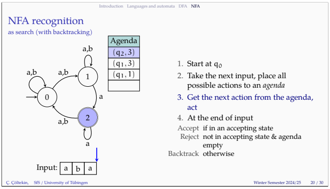
NFA recognition
as search (with backtracking)

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

Input: a b a



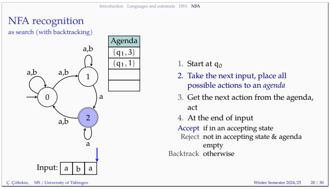
NFA recognition
as search (with backtracking)

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

Input: a b a



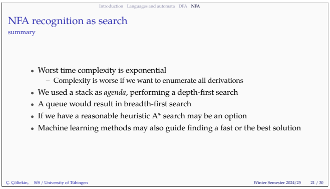
NFA recognition
as search (with backtracking)

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

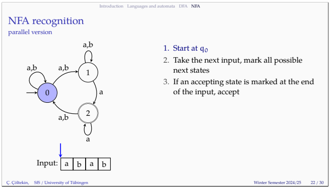
Accept if in an accepting state
Reject not in accepting state & agenda empty
Backtrack otherwise

Input: a b a



NFA recognition as search
summary

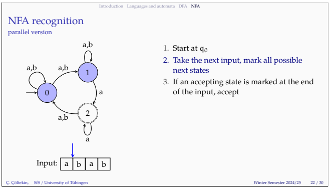
- Worst time complexity is exponential
  - Complexity is worse if we want to enumerate all derivations
- We used a stack as *agenda*, performing a depth-first search
- A queue would result in breadth-first search
- If we have a reasonable heuristic A* search may be an option
- Machine learning methods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

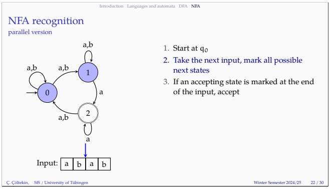
Input: a b a b



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

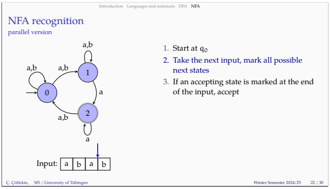
Input: a b a b



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

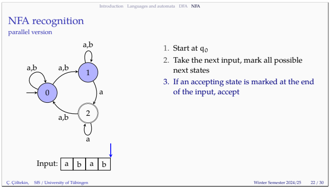
Input: a b a b



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

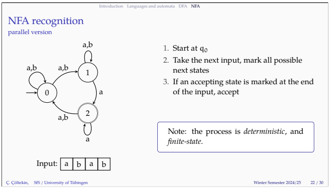
Input: a b a b



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

Input: a b a b

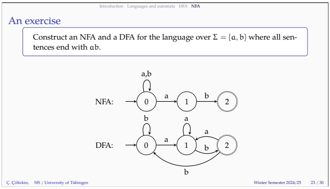


NFA recognition
parallel version

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

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

Input: a b a b



An exercise

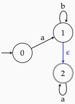
Construct an NFA and a DFA for the language over $\Sigma = \{a, b\}$ where all sentences end with ab.

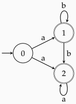
NFA:

DFA:

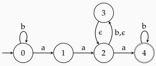
One more complication: ϵ transitions

- An extension of NFA, ϵ-NFA, allows moving without consuming an input symbol, indicated by an ϵ-transition (sometimes called a λ-transition)
- Any ϵ-NFA can be converted to an NFA





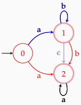
ϵ-transitions need attention



- How does the (depth-first) NFA recognition algorithm we described earlier work on this automaton?
- Can we do without ϵ transitions?

ϵ removal

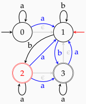
- Intuition: if $i \xrightarrow{a} j \xrightarrow{\epsilon} k$, then $i \xrightarrow{a} k$.
- We start with finding the ϵ-*closure* of all states
  - ϵ-closure$(q_0) = \{q_0\}$
  - ϵ-closure$(q_1) = \{q_1, q_2\}$
  - ϵ-closure$(q_2) = \{q_2\}$
- For each incoming arc (q_i, q_j) with label ℓ to a node q_j
  - add a new arc (q_i, q_k) with label ℓ, for all $q_k \in$ ϵ-closure(q_j)
  - remove all ϵ transitions (q_j, q_k) for all $q_k \in$ ϵ-closure(q_j)
- ϵ-transitions from the initial state, and to/from the accepting states need further attention (next slide)
- Remove useless states, if any



ϵ removal

another (less trivial) example

- Compute the ϵ-closure:
  - ϵ-closure$(q_0) = \{q_0, q_1\}$
  - ϵ-closure$(q_1) = \{q_1\}$
  - ϵ-closure$(q_2) = \{q_2, q_3\}$
  - ϵ-closure$(q_3) = \{q_3, q_1\}$
- For each incoming arc $\ell(q_i, q_j)$ to each node q_j
  - add $\ell(q_i, q_k)$ for all $q_k \in$ ϵ-closure(q_j)
  - if q_i is initial, mark q_j initial
  - if q_j is accepting, mark q_i accepting
  - remove all $\epsilon(q_j, q_k)$ for all $q_k \in$ ϵ-closure(q_j)



NFA–DFA equivalence

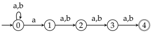
- The language recognized by every NFA is recognized by some DFA
- The set of DFA is a subset of the set of NFA (a DFA is also an NFA)
- The same is true for ϵ-NFA
- All recognize/generate regular languages
- NFA can automatically be converted to the equivalent DFA

Why do we use an NFA then?

- NFA (or ϵ-NFA) are often easier to construct
  - Intuitive for humans (cf. earlier exercise)
  - 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

Summary

- FSA are efficient tools with many applications
- FSA have two flavors: DFA, NFA (or maybe three: ε-NFA)
- DFA recognition is linear, recognition with NFA may require exponential time
- Reading suggestion: Hopcroft and Ullman (1979, Ch. 2&3) (and its successive editions), Jurafsky and Martin (2009, Ch. 2)

Next:

- FSA determinization, minimization
- Reading suggestion: Hopcroft and Ullman (1979, Ch. 2&3) (and its successive editions), Jurafsky and Martin (2009, Ch. 2)

Acknowledgments, credits, references

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 edition. Pearson Prentice Hall. ISBN: 978-0-13-504196-3.