COMP2022
Models of Computation

NFAs, DFAs, REs
have the same expressive power

Sasha Rubin

August 15, 2024

THE UNIVERSITY OF SYDNEY

Recap

Last lecture we saw how to convert a Regular Expression R into an equivalent NFA N.

Equivalent means they have the same language, i.e., $L(R) = L(N)$.

Let's revisit some slides from Lecture 1...

Matching problem for regular expressions

Input: regular expression R, string x (over alphabet Σ)
Output: `True` if x matches R, `False` otherwise

When you see a computational problem you should probably ask yourself this:

Is there a (fast) algorithm for solving the problem?

Matching problem for regular expressions

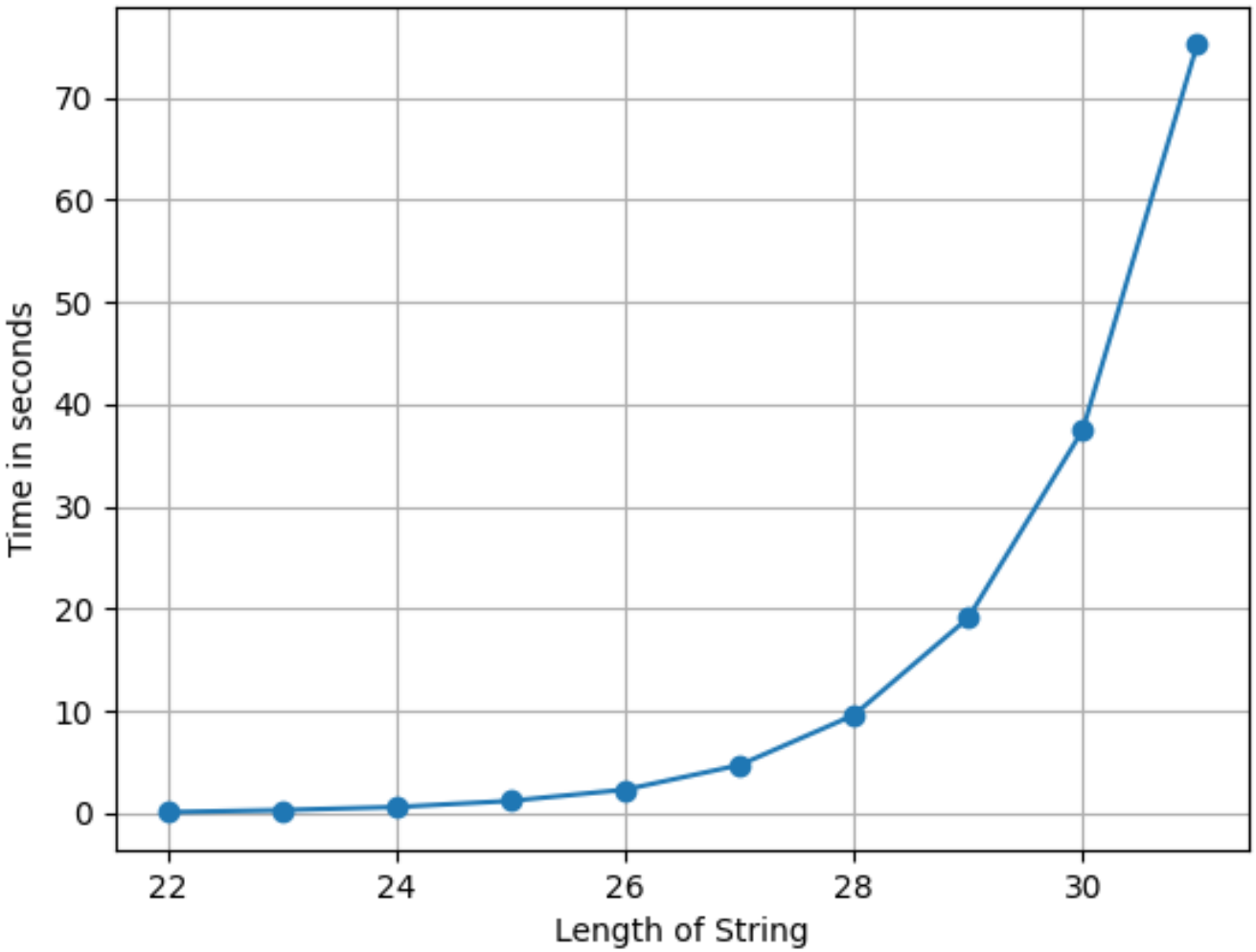
The naive recursive algorithm (Tutorial) runs in time that is exponential in the lengths R and x.

Is there an algorithm that runs in polynomial time?

What if we fix the regular expression R and only let the string x vary?

How does `Python` do?

```
1  import re
2  import time
3  r = '^(a+)+$'
4  for i in range(22,32):
5      s = 'a'*i + 'b'
6      start_time = time.time()
7      re.match(r,s)
8      print(i, (time.time() - start_time))
```


Time in seconds
0
10
20
30
40
50
60
70
22
24
26
28
30
Length of String

To design a polynomial time algorithm we will need to first learn about a model of computation called automata...

Matching problem for regular expressions

Input: regular expression R, string x (over alphabet Σ)
Output: `True` if x matches R, `False` otherwise

Steps:

1. Convert R to an equivalent NFA N
2. Check if $x \in L(N)$

What is the complexity/run-time?

The conversion to an NFA is polynomial (in fact, linear). In the Tutorial (Week 4), you will see how to solve the membership problem for NFAs in polynomial time (in fact, quadratic).

Takeaway

So, using automata, we see that the matching problem for regular expressions can be solved in polynomial time.

What about more complex problems about regular expressions?

Regular Expression Equivalence Problem:

> **Input**: two regular expression R_1, R_2
> **Output**: `True` if $L(R_1) = L(R_2)$, `False` otherwise

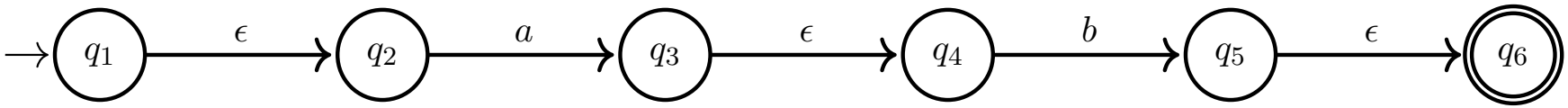
Plan: convert the regular expressions to DFAs (today's lecture), and check that the DFAs are equivalent (Tutorial in Week 3)

Where are we going?

1. From Regular Expressions to NFAs (last lecture)
2. From NFAs to NFAs without ϵ-transitions (today)
3. From NFAs without ϵ-transitions to DFAs (today)
4. From DFAs to Regular Expressions (today)

This is also covered in Sipser Chapter 1 (although he combines steps 2 and 3).

2. NFAs to NFAs without ϵ-transitions



2. NFAs to NFAs without ϵ-transitions

Theorem

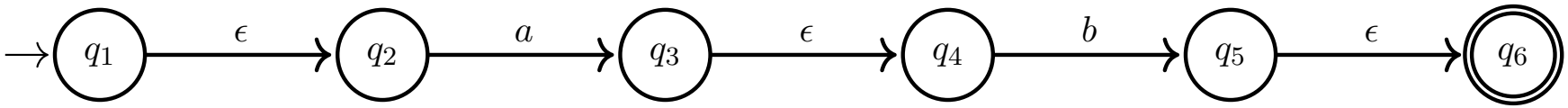
For every NFA N there is an NFA M without ϵ-transitions such that $L(N) = L(M)$.

Idea: "M skips over ϵ-transitions of N"

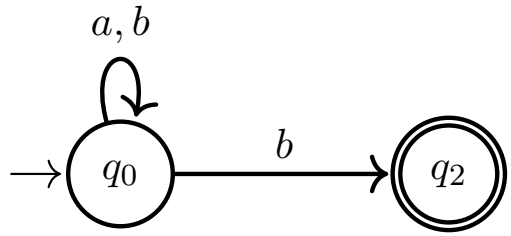
- So M is like N but we remove all the ϵ-transitions and add new transitions and new final states.
- Write $\mathrm{EC}(q, r)$ to mean that N can go from q to r using zero or more ϵ-transitions. Here EC stands for "Epsilon-Closure".
- When is q a final state of M?
  - if $\mathrm{EC}(q, r)$ and r is a final state of N.
- When is $q \xrightarrow{a} s$ a transition of M?
  - if $\mathrm{EC}(q, r)$ and $r \xrightarrow{a} s$.

How to compute if $\mathbf{EC}(q, r)$? (see Tutorial)

2. NFAs to NFAs without ϵ-transitions



3. NFAs without ϵ-transitions to DFAs



3. NFAs without ϵ-transitions to DFAs

Theorem

For every NFA N without ϵ-transitions there is a DFA M such that $L(M) = L(N)$.

Idea: "subset construction"

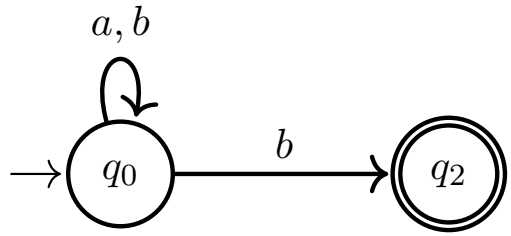
Q: How would you simulate the NFA if you were pretending to be a DFA?

A: Keep track of the set of possible states that the NFA is in.

Q: And when would you accept?

A: If at least one of the states I'm keeping track of is an accepting state of the NFA.

3. NFAs without ϵ-transitions to DFAs



3. NFAs without ϵ-transitions to DFAs

Theorem

For every NFA N without ϵ-transitions there is a DFA M such that $L(M) = L(N)$.

Construction

Given NFA $N = (Q, \Sigma, \delta, q_0, F)$ without ϵ-transitions the subset construction builds a DFA M:

- every set $X \subseteq Q$ is a state of M,
- the alphabet is Σ,
- for a state $X \subseteq Q$ of M, define $\delta'(X, a) = \bigcup_{q \in X} \delta(q, a)$,
- the initial state of M is the set $\{q_0\}$,
- for a state $X \subseteq Q$ of M, put X into F' if X contains an accepting state of N.

Where are we going?

1. From Regular Expressions to NFAs (last lecture)
2. From NFAs to NFAs without ϵ-transitions (today)
3. From NFAs without ϵ-transitions to DFAs (today)
4. From DFAs to Regular Expressions (today)

This is also covered in Sipser Chapter 1 (although he combines steps 2 and 3).

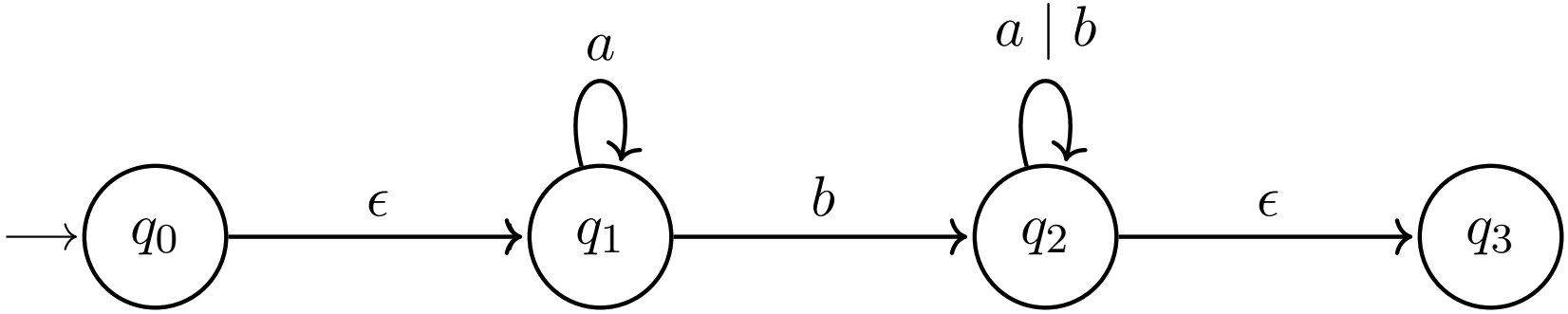
4. DFAs to Regular Expressions

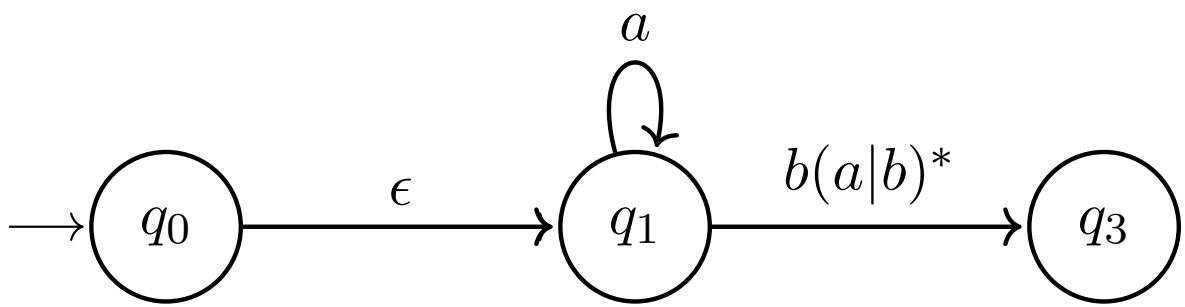
Theorem

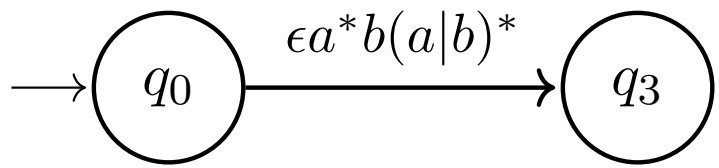
For every DFA M there is a regular expression R such that $L(M) = L(R)$.

Idea:

1. We convert M into a generalised nondeterministic finite automaton (GNFA) which is like an NFA except that it can have regular expressions label the transitions.
2. We successively remove states from the automaton, until there is just one transition (from the start state to the final state).
3. We return the regular expression on this last transition.


a
$a \mid b$
q_0
ϵ
q_1
b
q_2
ϵ
q_3


a
q_0
ϵ
q_1
$b(a|b)^*$
q_3


q_0
$\epsilon a^* b(a|b)^*$
q_3

GNFAs

- A run (aka computation) of a GNFA N on string w is a sequence of transitions

$$q_0 \xrightarrow{R_1} q_1 \xrightarrow{R_2} q_2 \xrightarrow{R_3} \dots \xrightarrow{R_m} q_m$$

  in N such that w matches the regular expression $R_1 R_2 \cdots R_m$.
- The run is accepting if $q_m \in F$.
- The language recognised by N is

$$L(N) = \{w \in \Sigma^* : w \text{ is accepted by } N\}$$

In the construction, we make sure that generalised NFAs are always of the following form:

1. the initial state has no incoming edges.
2. there is one final state, and it has no outgoing edges.
3. there are edges from every state (that is not final) to every state (that is not initial).

This simplifies the construction.

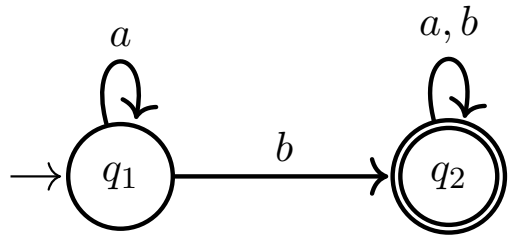
Here is how we convert the given DFA into a GNFA with these three restrictions.

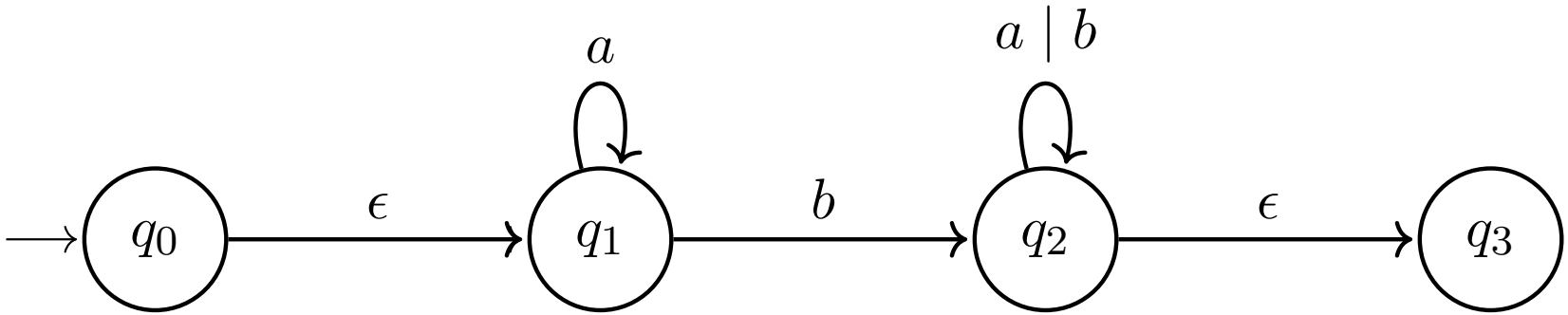
From DFAs to GNFAs

For every DFA M there is a GNFA N such that $L(M) = L(N)$.

1. Add an initial state and ϵ-transition to the old intial state.
2. Add a final state and ϵ-transitions to it from all the old final states.
3. Add transitions for every pair of states, including from a state to itself, (except leaving the final, or entering the initial).

From DFAs to GNFAs





. . . and some $\emptyset$-transitions which we usually don't draw since the picture gets cluttered.

Removing states from GNFAs

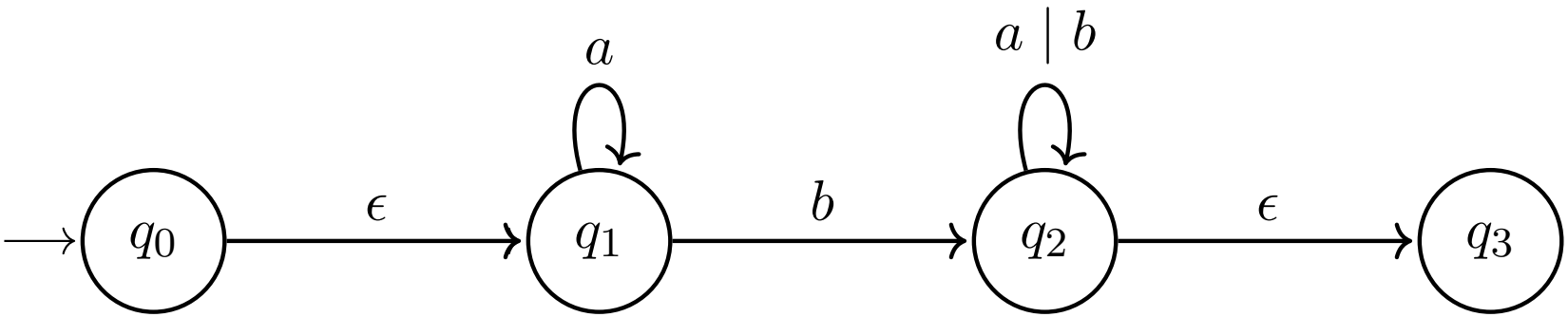
For every GNFA N with > 2 states there is a GNFA N' with one less state such that $L(N) = L(N')$.

1. Pick a state q to eliminate.
2. For every pair of remaining states (s, t), replace the label from s to t by the regular expression

$$R_{s,t} \mid (R_{s,q}\ R^*_{q,q}\ R_{q,t})$$

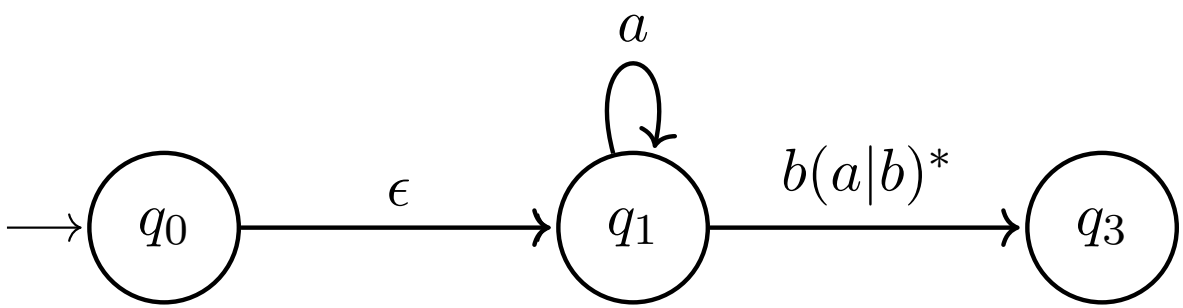
where $R_{x,y}$ is the regular expression labeling the transition from state x to state y.

Removing states from GNFAs



Eliminate q_2:

Removing states from GNFAs



Eliminate q_1:

Summary

1. A language is regular if it is recognised by some DFA.
2. The following models of computation describe the same languages:
   - DFA
   - NFA
   - NFA without epsilon transitions
   - Regular Expressions.
3. The regular languages are closed under the Boolean operations union, intersection, complementation, as well as concatenation and Kleene star.

Extra slides

Important DFA problems

There are natural computational problems associated with DFAs.

Are there (polynomial time) algorithms that solve them?

1. DFA Non-emptiness problem (see tutorial in week 3)

   Input: DFA M.
   Output: decide if $L(M) \neq \emptyset$.

2. DFA Equivalence problem (see tutorial in week 3)

   Input: DFAs M_1, M_2.
   Output: decide if $L(M_1) = L(M_2)$.

Important regular-expression problems

Using the solutions to the DFA problems, we can solve problems about regular expressions!

1. Regexps Non-emptiness problem

   Input: RE R.
   Output: decide if $L(R) \neq \emptyset$.
   - Idea: Convert R to DFA M and check if $L(M) \neq \emptyset$.

2. Regexps Equivalence problem

   Input: REs R_1, R_2.
   Output: decide if $L(R_1) = L(R_2)$.
   - Idea: Convert R_i to DFA M_i and check if $L(M_1) = L(M_2)$.

What is the complexity (running time) of these algorithm?