From two-way to one-way automata - three regular-expression based methods

Mans Hulden
University of Colorado

August 18 2015

Overview

- Regular-expression based methods to “compile” two-way automata (2DFAs/2NFAs) to 1DFAs

- Three subtly different methods, all using a similar approach - also provide simple equivalence proofs

- All are “generate-and-test”-style approaches that use auxiliary marker symbols as a intermediate to simulate the transitions of a 2DFA/2NFA

Background & motivation

- Automata are widely used in NLP applications, usually implemented as (very) extended regular expressions compiled into 1DFA
- Wide array of tools available for calculating with 1DFAs (and one-way transducers)
- No available implementation of 2DFA/2NFA > 1DFA conversion
- Two-way automata can compactly encode checking of multiple “long-distance dependencies” in strings

Two-way automata - notation

A 5-tuple: $(\Sigma, Q, Q_0, \delta, F)$

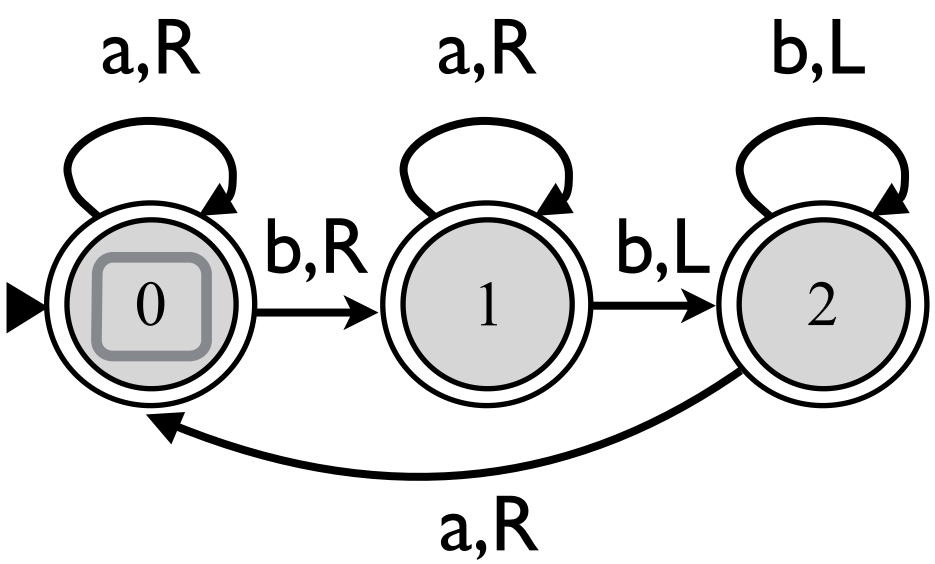
alphabet   states   initial states   transition function   final states

$\delta: Q \times \Sigma \rightarrow 2^{Q \times \{L,S,R\}}$

"left" "stay" "right"

A 2-way automaton M accepts a word *w* iff there exists some choice of transitions that lead to a final state with the read head at the right edge of a word

Example



b a b

string
read head

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

$Q = \{0,1,2\}$

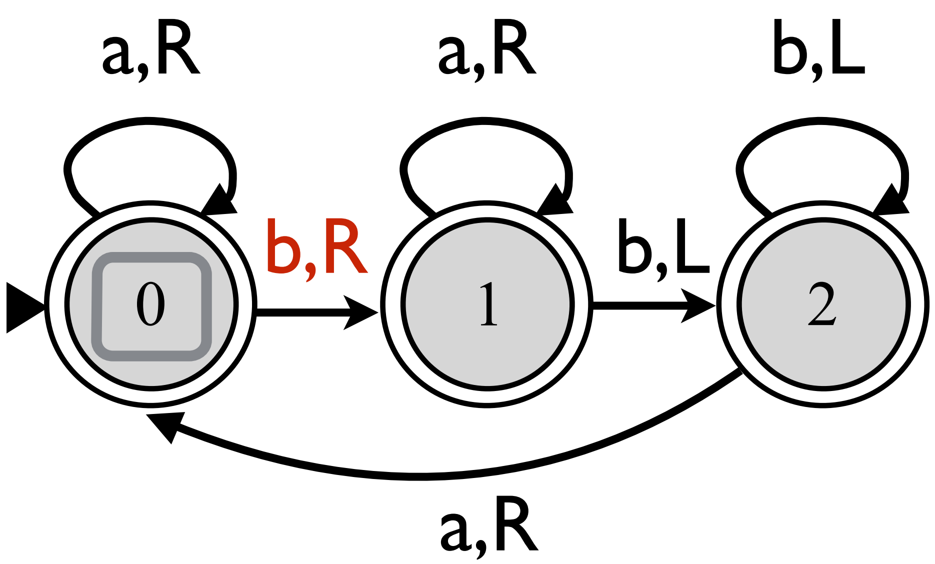
$Q_0 = \{0\}$

$\delta(0,a) = (0,R)$, $\delta(0,b) = (1,R)$,

$\delta(1,a) = (1,R)$, $\delta(1,b) = (2,L)$,

$\delta(2,a) = (0,R)$, $\delta(2,b) = (2,L)$

Example

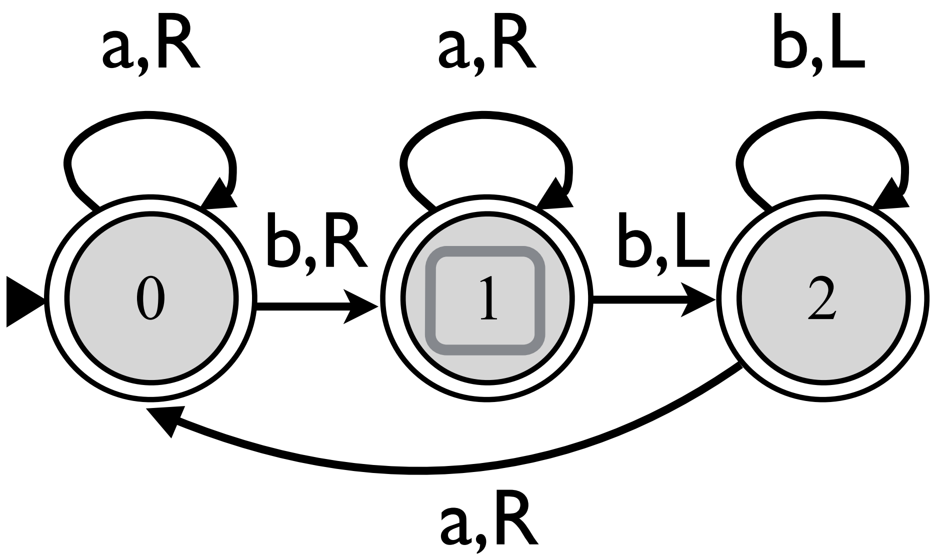


b a b

string
read head

$\Sigma = \{a,b\}$
$Q = \{0,1,2\}$
$Q_0 = \{0\}$
$\delta(0,a) = (0,R)$, $\delta(0,b) = (1,R)$,
$\delta(1,a) = (1,R)$, $\delta(1,b) = (2,L)$,
$\delta(2,a) = (0,R)$, $\delta(2,b) = (2,L)$

Example



b a b

string
read head

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

$Q = \{0,1,2\}$

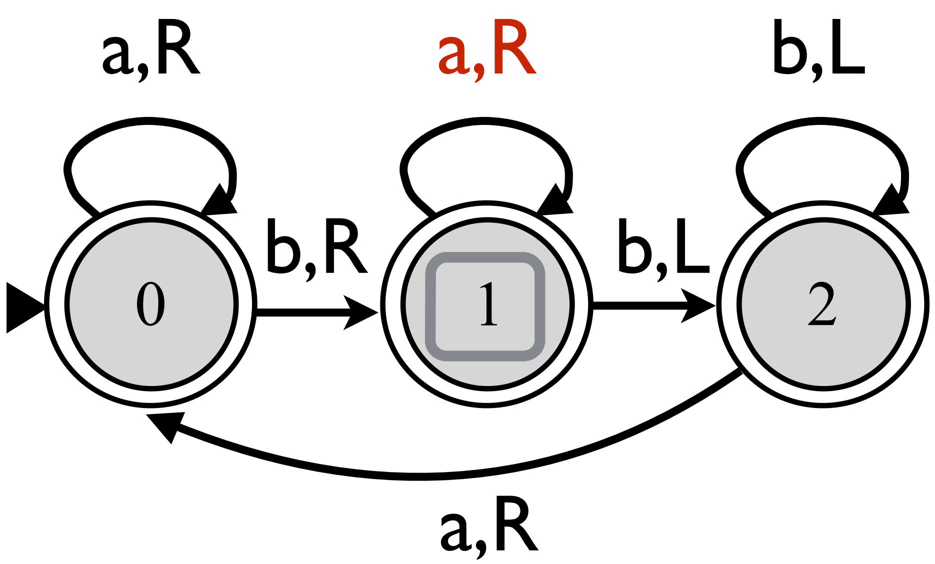
$Q_0 = \{0\}$

$\delta(0,a) = (0,R)$, $\delta(0,b) = (1,R)$,

$\delta(1,a) = (1,R)$, $\delta(1,b) = (2,L)$,

$\delta(2,a) = (0,R)$, $\delta(2,b) = (2,L)$

Example



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

$Q = \{0,1,2\}$

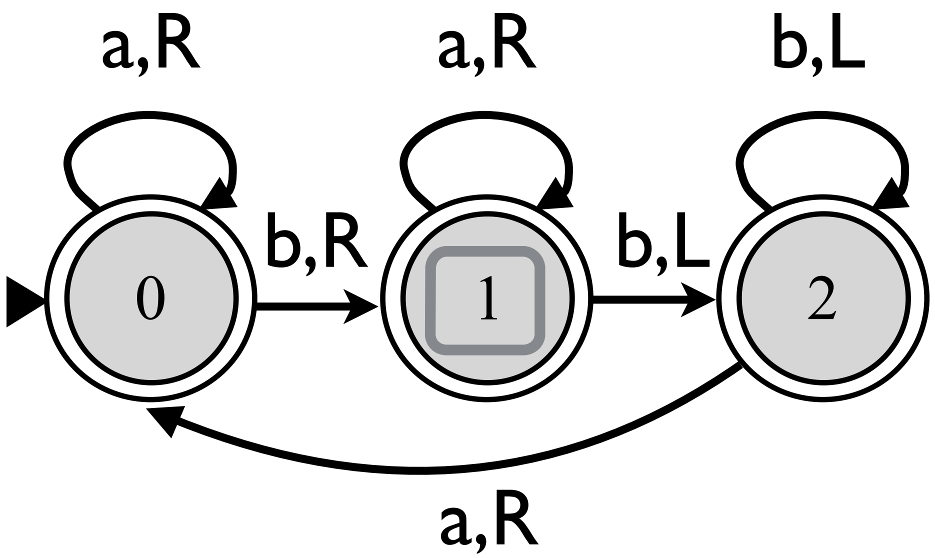
$Q_0 = \{0\}$

$\delta(0,a) = (0,R)$, $\delta(0,b) = (1,R)$,

$\delta(1,a) = (1,R)$, $\delta(1,b) = (2,L)$,

$\delta(2,a) = (0,R)$, $\delta(2,b) = (2,L)$

Example



b a b

string
read head

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

$Q = \{0,1,2\}$

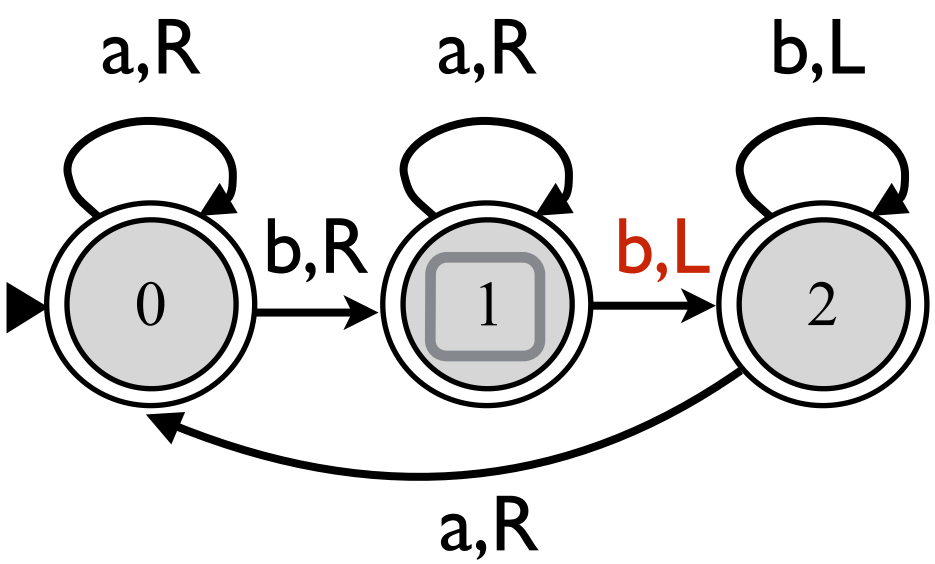
$Q_0 = \{0\}$

$\delta(0,a) = (0,R)$, $\delta(0,b) = (1,R)$,

$\delta(1,a) = (1,R)$, $\delta(1,b) = (2,L)$,

$\delta(2,a) = (0,R)$, $\delta(2,b) = (2,L)$

Example

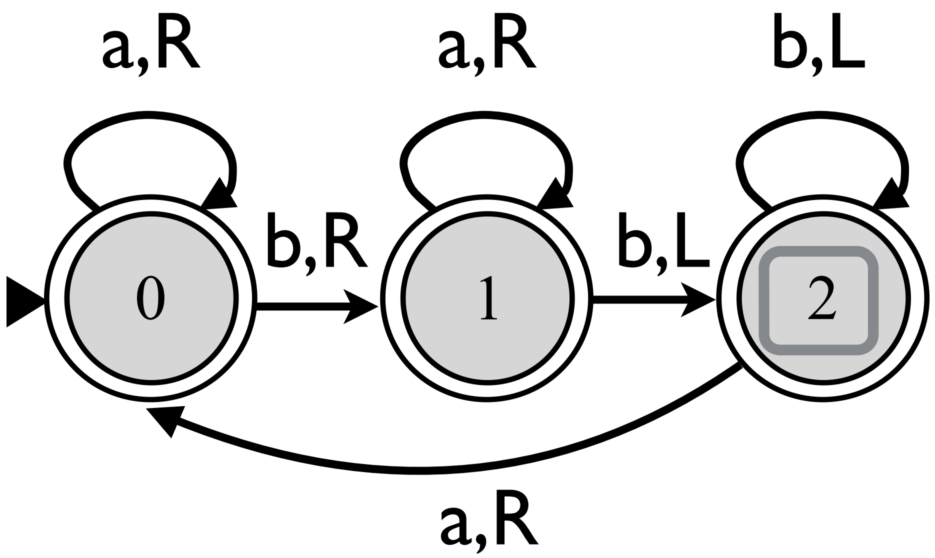


b a b

string
read head

$\Sigma = \{a,b\}$
$Q = \{0,1,2\}$
$Q_0 = \{0\}$
$\delta(0,a) = (0,R)$, $\delta(0,b) = (1,R)$,
$\delta(1,a) = (1,R)$, $\delta(1,b) = (2,L)$,
$\delta(2,a) = (0,R)$, $\delta(2,b) = (2,L)$

Example



b a b

string
read head

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

$Q = \{0,1,2\}$

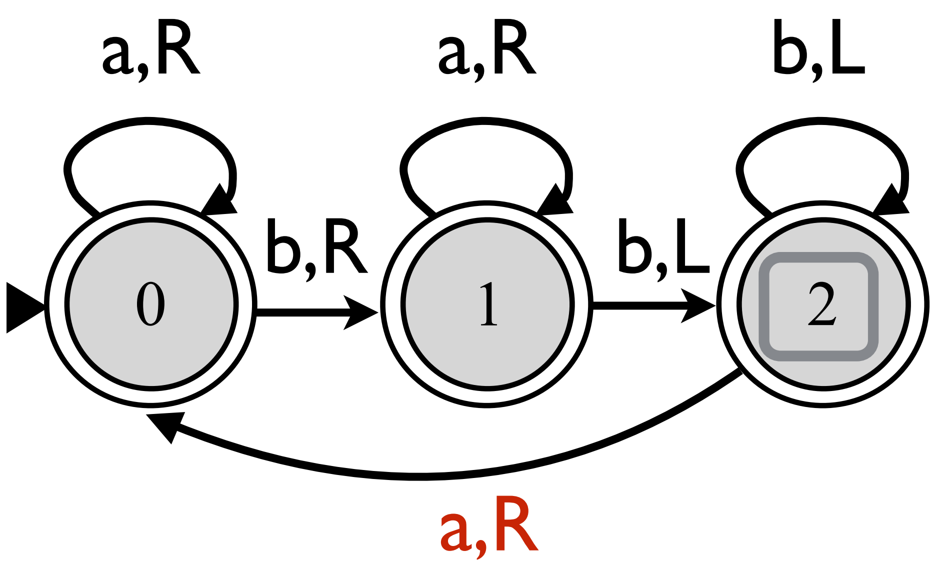
$Q_0 = \{0\}$

$\delta(0,a) = (0,R)$, $\delta(0,b) = (1,R)$,

$\delta(1,a) = (1,R)$, $\delta(1,b) = (2,L)$,

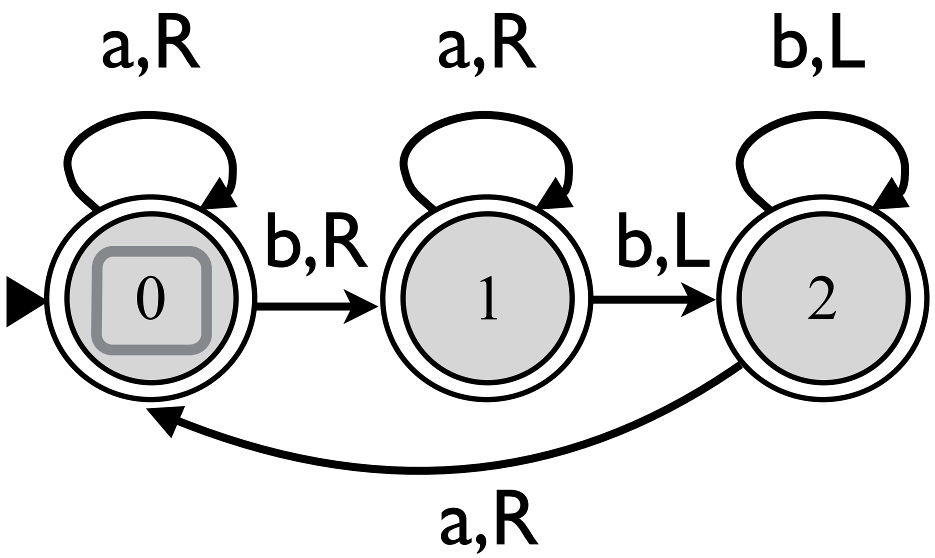
$\delta(2,a) = (0,R)$, $\delta(2,b) = (2,L)$

Example



$\Sigma = \{a,b\}$
$Q = \{0,1,2\}$
$Q_0 = \{0\}$
$\delta(0,a) = (0,R)$, $\delta(0,b) = (1,R)$,
$\delta(1,a) = (1,R)$, $\delta(1,b) = (2,L)$,
$\delta(2,a) = (0,R)$, $\delta(2,b) = (2,L)$

Example



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

$Q = \{0,1,2\}$

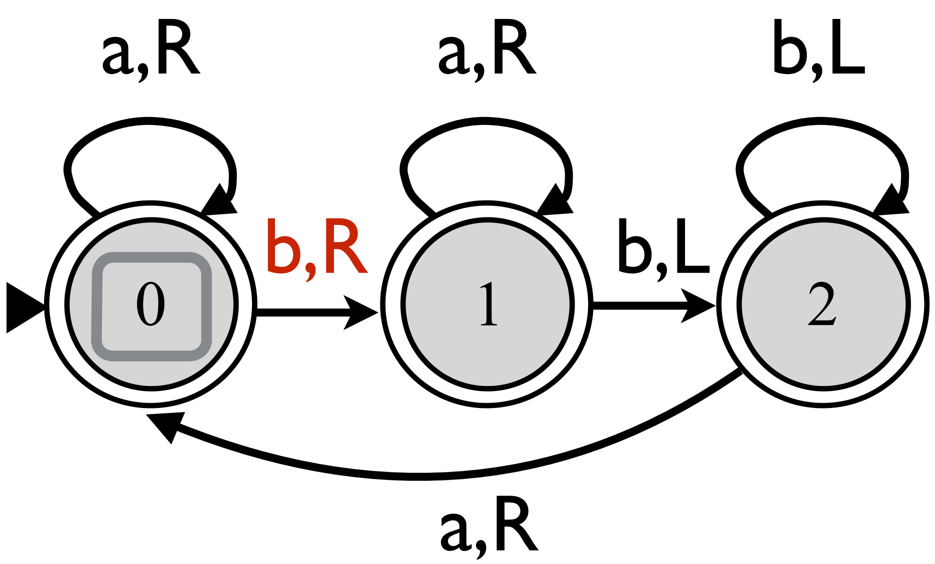
$Q_0 = \{0\}$

$\delta(0,a) = (0,R)$, $\delta(0,b) = (1,R)$,

$\delta(1,a) = (1,R)$, $\delta(1,b) = (2,L)$,

$\delta(2,a) = (0,R)$, $\delta(2,b) = (2,L)$

Example

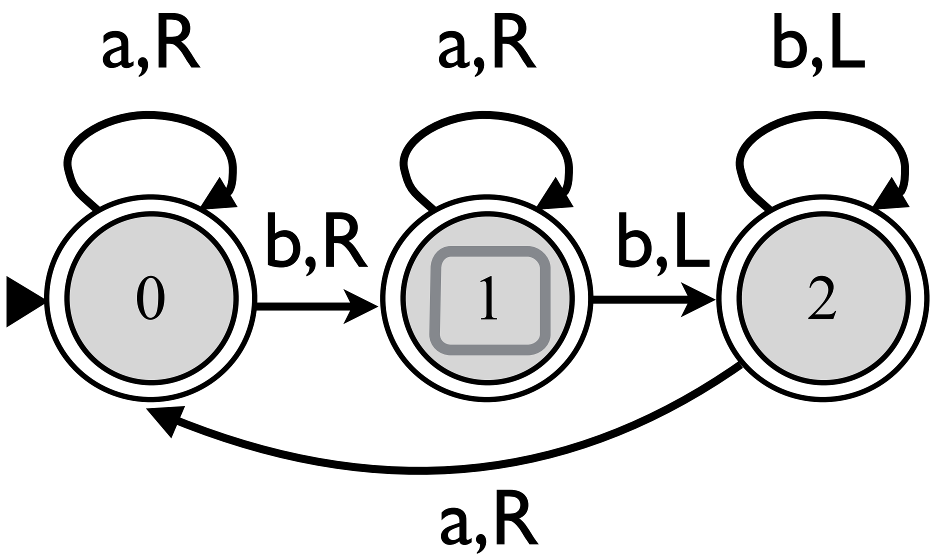


b a b

string
read head

$\Sigma = \{a,b\}$
$Q = \{0,1,2\}$
$Q_0 = \{0\}$
$\delta(0,a) = (0,R)$, $\delta(0,b) = (1,R)$,
$\delta(1,a) = (1,R)$, $\delta(1,b) = (2,L)$,
$\delta(2,a) = (0,R)$, $\delta(2,b) = (2,L)$

Example



b a b

string
read head

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

$Q = \{0,1,2\}$

$Q_0 = \{0\}$

$\delta(0,a) = (0,R)$, $\delta(0,b) = (1,R)$,

$\delta(1,a) = (1,R)$, $\delta(1,b) = (2,L)$,

$\delta(2,a) = (0,R)$, $\delta(2,b) = (2,L)$

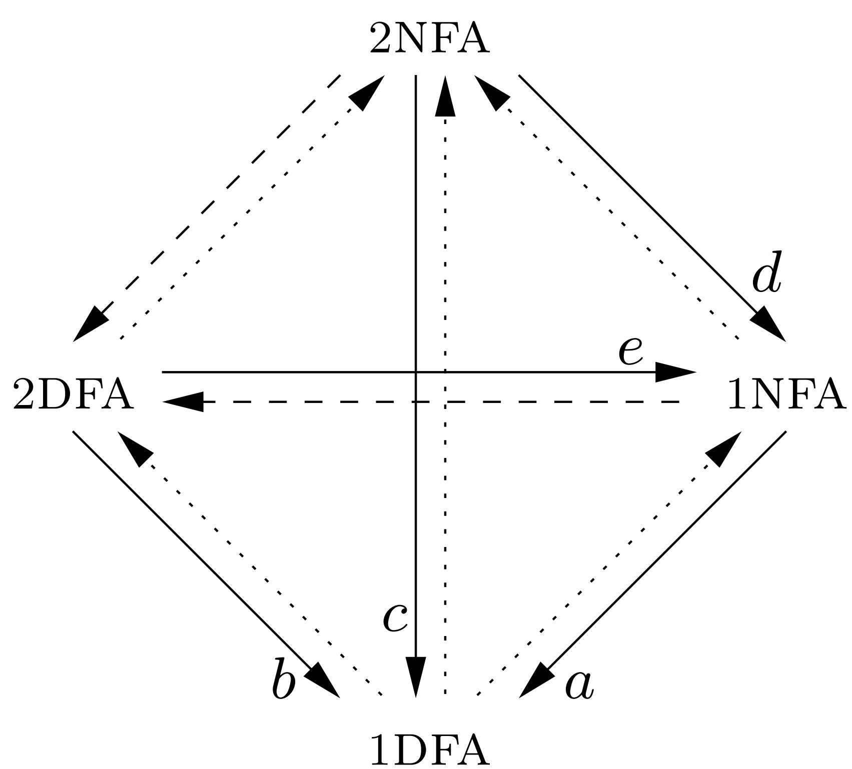
Properties of two-way automata

Equivalent to 1DFA in generative capacity - recognize only the regular sets (Rabin and Scott, 1959; Shepherdson 1959; Vardi (1989))

Conversion “tradeoffs” between different two-way automata and one-way automata mostly well understood

Properties of two-way automata

Tradeoffs



$$a = 2^n - 1$$

$$b = n\bigl(n^n - (n-1)^n\bigr)$$

$$c = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \binom{n}{i}\binom{n}{j}\bigl(2^i - 1\bigr)^j$$

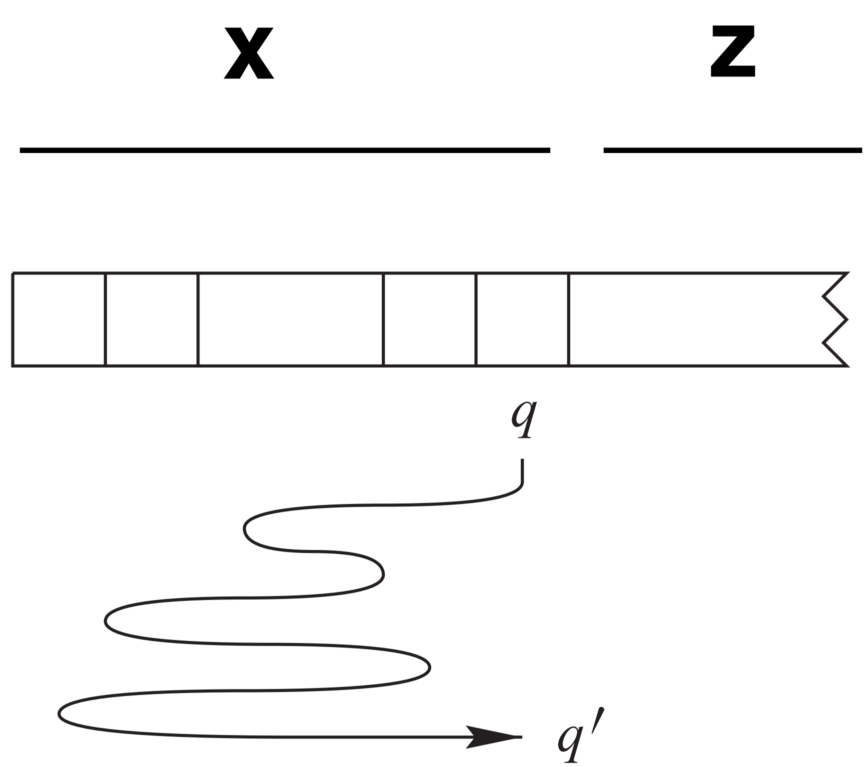
$$d = e = \binom{2n}{n+1}$$

$$f = n$$

from Kapoutsis (2005)

(dashed arrows open)

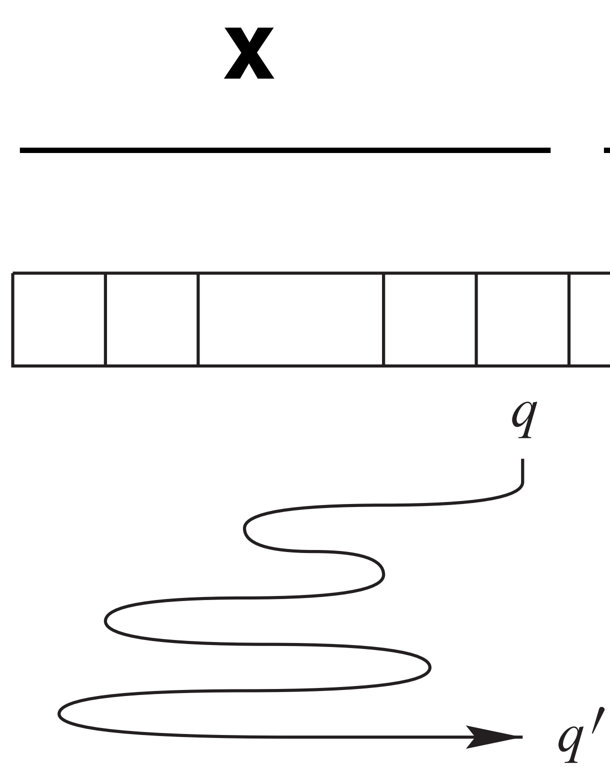
Traditional 2DFA~1DFA conversion: crossing sequences



Shepherdson's (1959) observation: Consider some string xz; if 2DFA M is in some state q when reading the last symbol of x, M's behavior (eventual exit state q') is completely determined by x (and independent of z)

We can record all M's potential exit states with a finite table (of so-called crossing functions)

Traditional 2DFA~1DFA conversion: crossing sequences



We can construct an equivalent 1DFA by simulating the behavior of potentially all prefixes x up to some length, and analyzing their crossing functions (there are potentially $(n+1)^n$ unique functions.)

Each crossing function becomes a state in a 1DFA (roughly)

Current method

We want to take advantage of the existence of efficient toolkits for compiling extended regular expressions into minimal 1DFAs

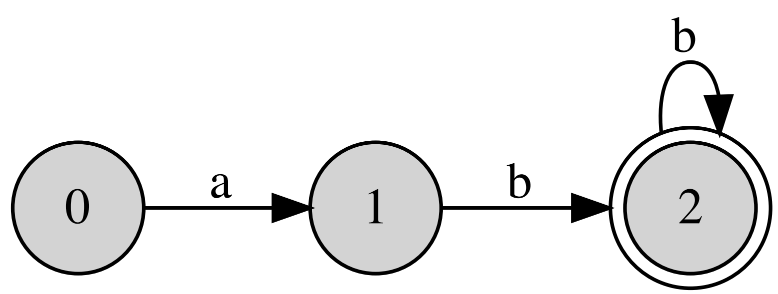
Idea:

(1) Use a string encoding that contains marker symbols that "simulate" the moves of a given 2DFA/2NFA.
(2) Constrain marker symbols to correspond to legitimate accepting paths in a 2DFA/2NFA.
(3) Remove markers by homomorphism

Step (2) should be expressible as a regular language (preferably locally testable; similar to n-slt strategies for simulating 1-way automata (as in Medvedev(1964))

Current method

Recall: almost trivial state-symbol encoding gives result that every regular language over Σ is the homomorphic image of a 3-testable language over (Σ ∪ Γ) , e.g.

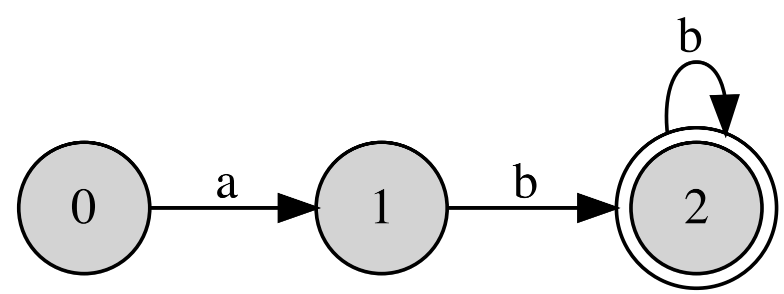


$\Sigma = \{a,b\}$
$\Gamma = \{0,1,2\}$

$h(0\ a\ 1\ b\ 2\ b\ 2) = abb$

Current method

Recall: almost trivial state-symbol encoding gives result that every regular language over Σ is the homomorphic image of a 3-testable language over (Σ ∪ Γ) , e.g.

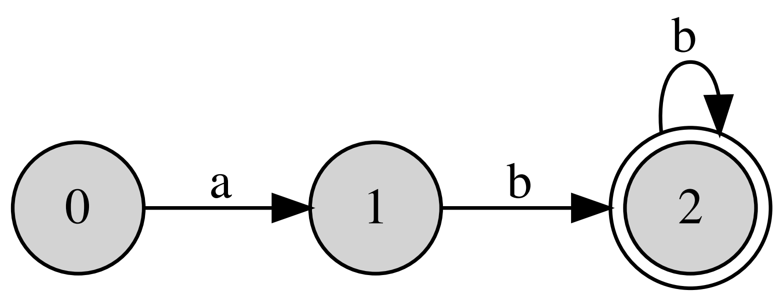


$\Sigma = \{a,b\}$
$\Gamma = \{0,1,2\}$

$h(\boxed{0\ a\ 1}\ b\ 2\ b\ 2) = abb$

Current method

Recall: almost trivial state-symbol encoding gives result that every regular language over Σ is the homomorphic image of a 3-testable language over (Σ ∪ Γ) , e.g.

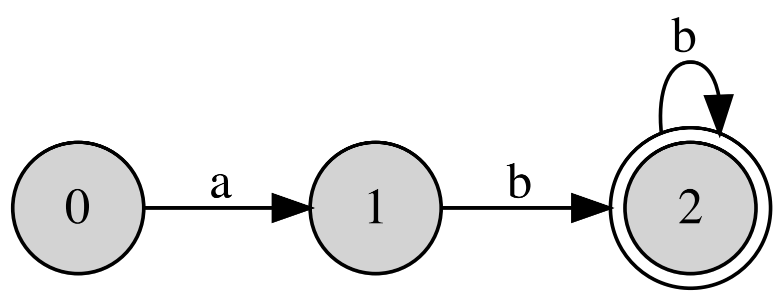


$\Sigma = \{a,b\}$
$\Gamma = \{0,1,2\}$

$$h(0\ \boxed{a\ 1\ b}\ 2\ b\ 2) = abb$$

Current method

Recall: almost trivial state-symbol encoding gives result that every regular language over Σ is the homomorphic image of a 3-testable language over (Σ ∪ Γ) , e.g.

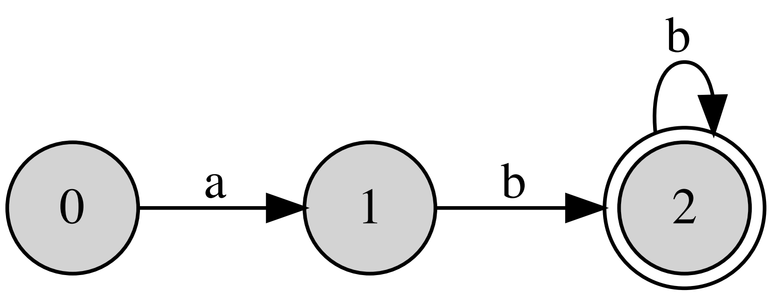


$\Sigma = \{a,b\}$
$\Gamma = \{0,1,2\}$

$$h(0\ a\ \boxed{1\ b\ 2}\ b\ 2) = abb$$

Current method

Recall: almost trivial state-symbol encoding gives result that every regular language over Σ is the homomorphic image of a 3-testable language over (Σ ∪ Γ) , e.g.

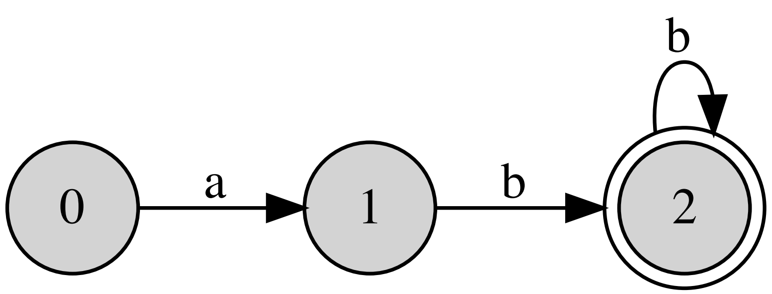


$\Sigma = \{a,b\}$
$\Gamma = \{0,1,2\}$

$h(0\ a\ 1\ \boxed{b\ 2\ b}\ 2) = abb$

Current method

Recall: almost trivial state-symbol encoding gives result that every regular language over Σ is the homomorphic image of a 3-testable language over (Σ ∪ Γ) , e.g.



$\Sigma = \{a,b\}$
$\Gamma = \{0,1,2\}$

$h(0\ a\ 1\ b\ \boxed{2\ b\ 2}) = abb$

Two-way string encoding

Define a regular language L_{base} containing symbols from Σ (of 2DFA/2NFA M), intersperse three-symbol "control" sequences from $\Gamma = \{q_0, \ldots, q_n, L,R,S\}$ between every $a \in \Sigma$

source target direction

01R b 11R 20R a 12L 01R b 11

string over Σ

Method 1: 2DFA to regular expression

Here, we want to constrain the control strings to reflect acceptance of a string in Σ^* by a 2DFA M

We start with the language L_{base} which, given M:

(1) starts with the symbol corresponding to the initial state

(2) control sequences reflect actual transitions in M

(3) ends with a pair corresponding to a final state in M

(1) (2) (3)

01R b 11R 20R a 12L 01R b 11

Method 1: 2DFA to regular expression

Also, define another language $L_{license}$ that complies with the following: a two-symbol control sequence pq only occurs if there is a preceding “R”, following “L” or simultaneous “S” transition to p; or if p is the initial state and leftmost in the string.

01 is allowed because $0 \in Q_0$

01R b 11R 20R a 12L 01R b 11

Method 1: 2DFA to regular expression

Also, define another language $L_{license}$ that complies with the following: a two-symbol control sequence pq only occurs if there is a preceding “R”, following “L” or simultaneous “S” transition to p; or if p is the initial state and leftmost in the string.

11 is allowed because of 1R “preceding”

01R b 11R 20R a 12L 01R b 11

Method 1: 2DFA to regular expression

Def a simple homomorphism $h(a) = \boldsymbol{\varepsilon}$ for all $a \in \Gamma$

Example:

$h($01R b 11R 20R a 12L 01R b 11$) = bab$

Method 1: 2DFA to regular expression

L_{base} and $L_{license}$ are obviously regular (can in fact be made k-testable for some k that depends on the structure of M)

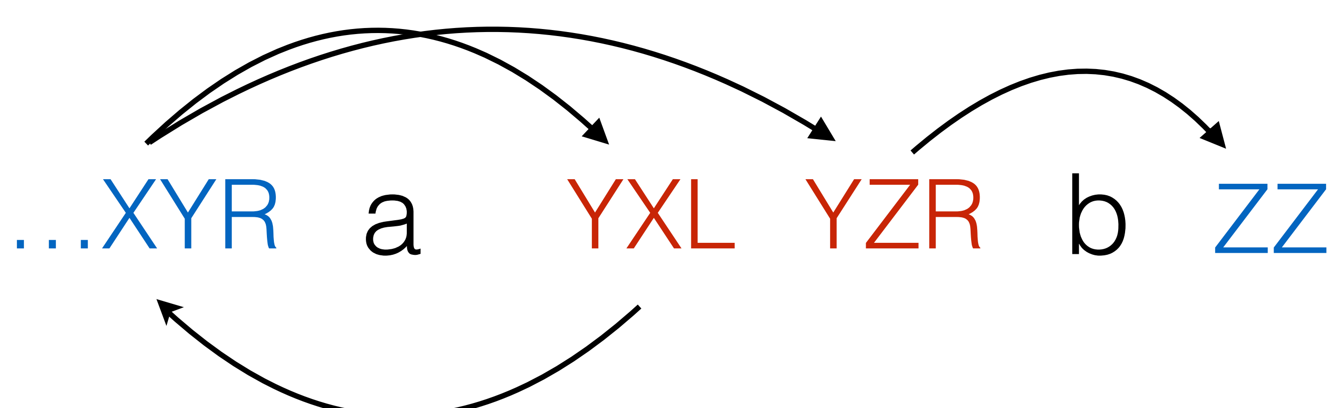
Claim: A 2DFA M accepts a word w iff $w \in h(L_{base} \cap L_{license})$

Sketch (⇒): induction on the number of steps in the computation of M

01R b 11R 20R a 12L 01R b 11

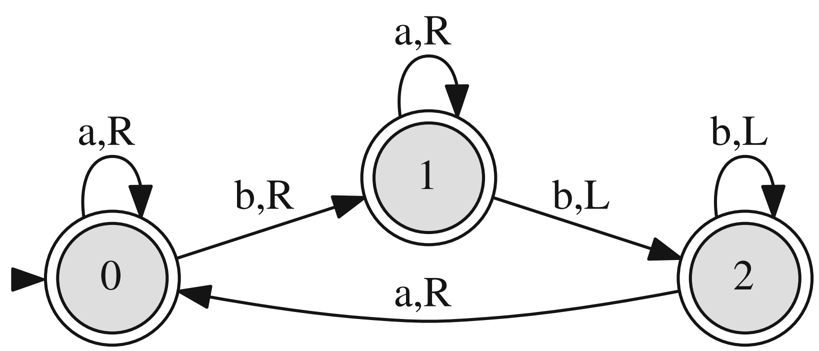
Method 1: 2DFA to regular expression

Sketch (⇐): All strings in ($L_{base} \cap L_{license}$) end in some sequence qq. This must be permitted by some other sequence pq, etc. which eventually needs to end in the initial state at the left edge (because M is deterministic, this backward sequence cannot end in a loop).



impossible config.

(Implementation aside)




```
def S     [a|b];                  # Alphabet
def Q     [0|1|2];                # States
def h(X)  [X .o. \S -> 0].2;      # The homomorphism

def Ta     0 0 R | 1 1 R | 2 0 R ;
def Tb     0 1 R | 1 2 L | 2 2 L ;
def Lend   0 0   | 1 1   | 2 2 ;

def Lbase [Ta+ a|Tb+ b]* Lend;

def ifQPR(P,Q,R) ~[~P Q ~R];


def L Lbase & ifQPR(0|?* 0 S \S*|?* Q 0 R \S* S \S*, 0 Q, \S* S \S* Q 0 L ?*|\S* 0 S ?*) &
              ifQPR(  ?* 1 S \S*|?* Q 1 R \S* S \S*, 1 Q, \S* S \S* Q 1 L ?*|\S* 1 S ?*) &
              ifQPR(  ?* 2 S \S*|?* Q 2 R \S* S \S*, 2 Q, \S* S \S* Q 2 L ?*|\S* 2 S ?*) ;

regex h(L);
```

Method 2: 2NFA to regular expression

The above method 1 can't be used to model a 2NFA (construction hinges on determinism)

Method 2: 2NFA to regular expression

Same encoding as before, but:

(1) Add symbol C (for crash) to Γ; Γ = {L,R,S,C}
(2) ppC is added for all missing transitions from p with some symbol in Σ, and ppC for nonfinal states p
(3) Instead of permitting successors if predecessor is present, $L_{license}$ *requires* all possible successors transitions to be present for any predecessor, except
(4) When a control sequence C is present

requires

1 is nonfinal

01R b 11R 20R a 12L 01R b 11C

Method 2: 2NFA to regular expression

Intuition:

We're accepting only strings where all corresponding paths eventually crash in the 2NFA

All accepting paths in $L = L_{base} \cap L_{license}$ end in a crash. A word w is not in $h(L_{base} \cap L_{license})$ iff all paths crash for w with 2NFA M.

$L_2 = \Sigma^* - h(L_{base} \cap L_{license})$;

A 2NFA M accepts a word w iff $w \in L_2$

Method 2: 2NFA to regular expression

Note: this method can be seen as a regular expression model of Vardi's (1989) set-based proof that 2NFA are regular:

Lemma 3.1: *Let* $A = (\Sigma, S, S_0, \rho, F)$ *be a two-way automaton, and* $w = a_0, \ldots, a_n$ *be a word in* $\Sigma^\star$. *A does not accept* w *if and only if there exists a sequence* $T_0, \ldots, T_{n+1}$ *of subsets of* S *such that the following conditions hold:*

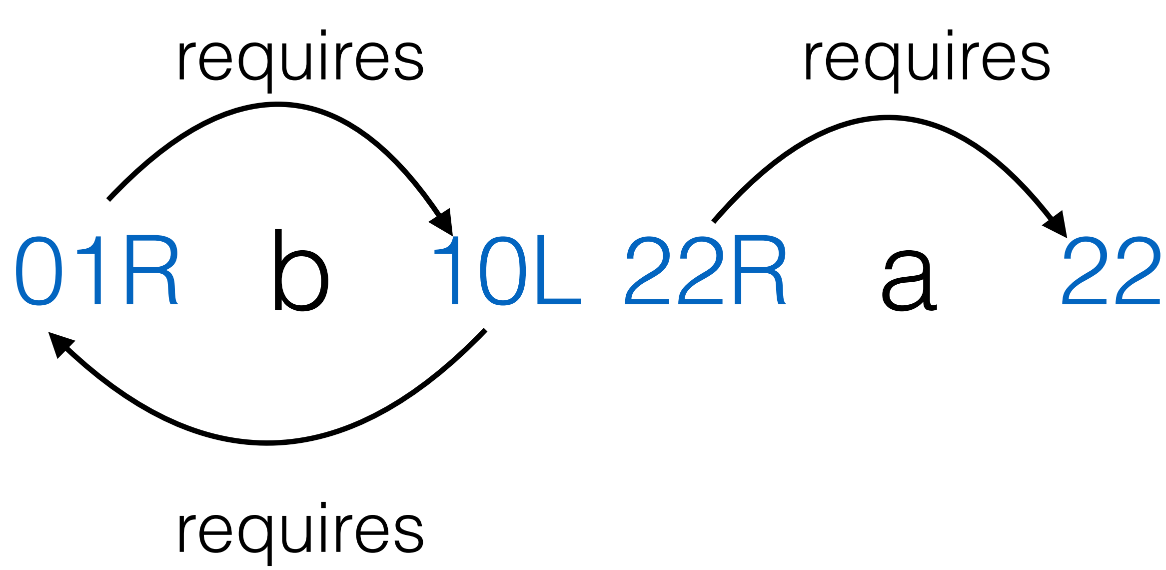
1. $S_0 \subseteq T_0$,
2. $T_{n+1} \cap F = \emptyset$, *and*
3. *for* $0 \leq i \leq n$, *if* $s \in T_i$, $(s', k) \in \rho(s, a)$, *and* $i + k > 0$, *then* $s' \in T_{i+k}$.

Vardi (1989)

Method 3: 2NFA to regular expression (no complement)

Note:

We can't perform a direct construction by modifying method 1 and have $L_{license}$ "require" a subsequent transition since this may lead to acceptance of spurious (nonminimal) paths, e.g.:



valid transitions
valid final state
"requirements" fulfilled

Method 3: 2NFA to regular expression (no complement)

If we remove the requirement that strings end in “final states”, we can see that such a language L would accept strings like:

01R b 10L 22R a 22 (1)

01R b 10L a (2)

Observation:
all nonminimal paths (1) can be produced from paths of type (2) by adding at least one symbol from Γ to strings in L

Method 3: 2NFA to regular expression (no complement)

We can now characterize directly the strings accepted by a 2NFA:

$L_3 = h((L - \mathrm{Insert}(L)) \cap \Delta^*FF)$

add at least one symbol from Γ somewhere
(filters out spurious/nonminimal paths)

must end in final state

A 2NFA M accepts a word w iff w in L_3

Method 3: 2NFA to regular expression (no complement)

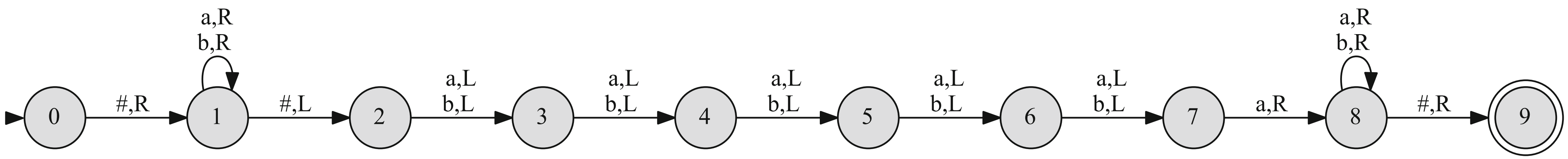
$L_3 = h((L - \text{Insert}(L)) \cap \Delta^*QQ)$

A 2NFA M accepts a word w iff w in L_3

Proof sketch: if L contains what is a nonminimal (spurious) path in the computation of a 2NFA M, then L also contains the corresponding minimal path. Hence L-Insert(L) will reflect removal of all spurious paths and L_3 contains a word w iff 2NFA M accepts.

Experiments on order of intersection

“Check if nth last symbol is a” (n = 6 here)



k	size($L_{base} \cap L_{license_0} \cap \ldots \cap L_{license_k}$)	size($L_{license_0} \cap \ldots \cap L_{license_k}$)
0	33	58
1	77	1,394
2	112	29,634
3	166	589,570
4	204	11,271,170
5	226	NF
6	210	NF
7	131	NF
8	138	NF
9	181	NF

Expensive

Summary

2DFA/2NFA are convertible to regular expressions with an intermediate simulation encoding

The proofs and construction bypass complications of analyzing “crossing sequences” and stress the local nature of 2DFA/2NFA computations

Useable in practice

Thank you

Code and examples at:
https://github.com/mhulden/2nfa