From Regular Expression to Scanner (FA)

Quick Review of Regular Expressions

- All strings of 1s and 0s ending in a 1

  $(\underline{0} \mid \underline{1})^* \underline{1}$

- All strings over lowercase letters where the vowels (a,e,i,o, & u) occur exactly once, in ascending order

  Let Cons be $(\underline{b} \mid \underline{c} \mid \underline{d} \mid \underline{f} \mid \underline{g} \mid \underline{h} \mid \underline{j} \mid \underline{k} \mid \underline{l} \mid \underline{m} \mid \underline{n} \mid \underline{p} \mid \underline{q} \mid \underline{r} \mid \underline{s} \mid \underline{t} \mid \underline{v} \mid \underline{w} \mid \underline{x} \mid \underline{y} \mid \underline{z})$

  $Cons^* \ \underline{a} \ Cons^* \ \underline{e} \ Cons^* \ \underline{i} \ Cons^* \ \underline{o} \ Cons^* \ \underline{u} \ Cons^*$

- All strings of 1s and 0s that do not contain three 0s in a row:

  $(\underline{1}^* (\varepsilon \mid \underline{01} \mid \underline{001}) \underline{1}^*)^* (\varepsilon \mid \underline{0} \mid \underline{00})$

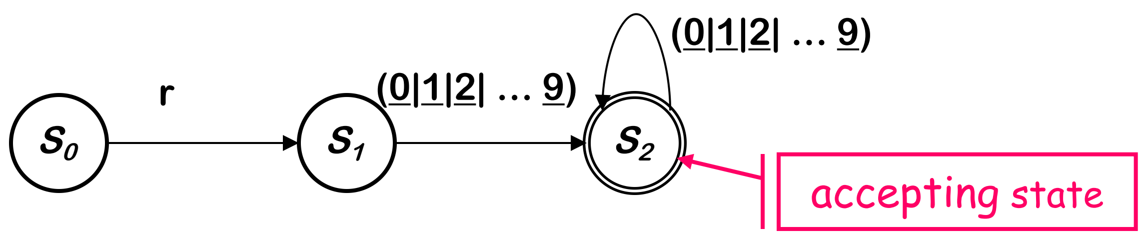
Example

Consider the problem of recognizing ILOC register names

Register → r (0|1|2| … | 9) (0|1|2| … | 9)*

- Allows registers of arbitrary number
- Requires at least one digit

RE corresponds to a recognizer (or DFA)



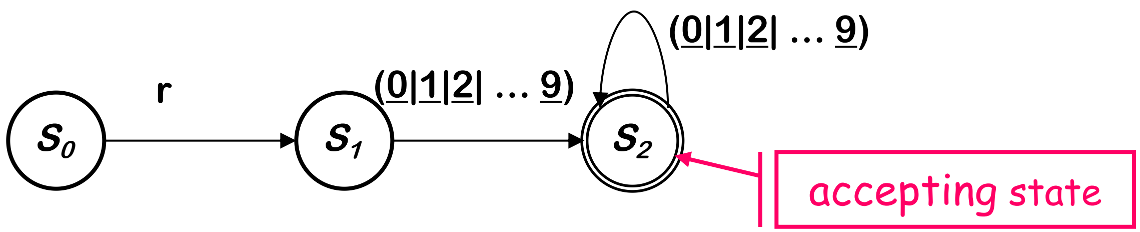
Recognizer for *Register*

Transitions on other inputs go to an error state, s_e

Example (continued)

DFA operation

- Start in state S_0 & make transitions on each input character
- DFA accepts a word $\underline{x}$ *iff* $\underline{x}$ leaves it in a final state (S_2)



Recognizer for *Register*

So,

- r17 takes it through s_0, s_1, s_2 and accepts
- r takes it through s_0, s_1 and fails
- a takes it straight to s_e

Example (continued)

To be useful, the recognizer must be converted into code

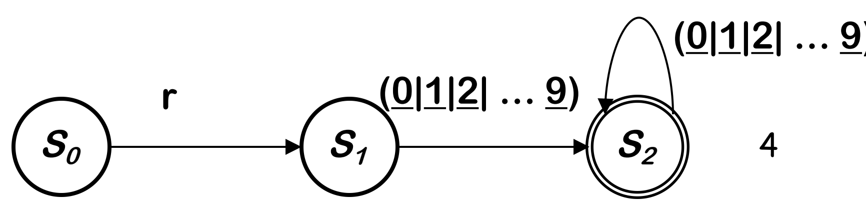
```
Char ← next character
State ← s0

while (Char ≠ EOF)
   State ← δ(State,Char)
    Char ← next character

if (State is a final state )
   then report success
   else  report failure
```

Skeleton recognizer

δ	r	0,1,2,3,4, 5,6,7,8,9	*All others*
s_0	s_1	s_e	s_e
s_1	s_e	s_2	s_e
s_2	s_e	s_2	s_e
s_e	s_e	s_e	s_e

Table encoding the RE

O(1) cost per character (or per transition)



Example (continued)

We can add “actions” to each transition

```
Char ← next character
State ← s0

while (Char ≠ EOF)
    Next ← δ(State,Char)
    Act  ← α(State,Char)
    perform action Act
    State ← Next
    Char ← next character

if (State is a final state )
    then report success
    else  report failure
```

Skeleton recognizer

δ / α	r	0,1,2,3,4, 5,6,7,8,9	*All others*
s_0	s_1 *start*	s_e *error*	s_e *error*
s_1	s_e *error*	s_2 *add*	s_e *error*
s_2	s_e *error*	s_2 *add*	s_e *error*
s_e	s_e *error*	s_e *error*	s_e *error*

Table encoding RE

Typical action is to capture the lexeme

What if we need a tighter specification?

r *Digit Digit** allows arbitrary numbers

- Accepts r00000
- Accepts r99999
- What if we want to limit it to r0 through r31 ?

Write a tighter regular expression

- *Register* → r ((0|1|2) (*Digit* | ε) | (4|5|6|7|8|9) | (3|30|31))
- *Register* → r0|r1|r2| … |r31|r00|r01|r02| … |r09

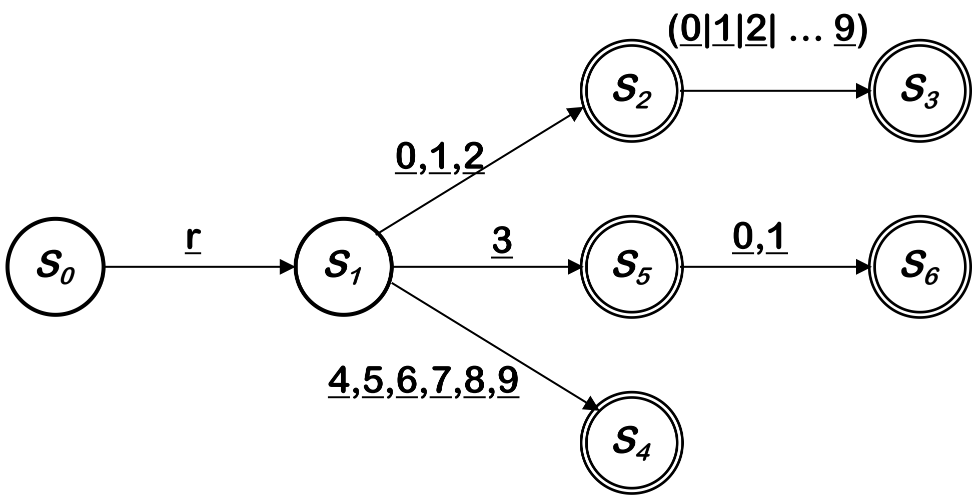
Produces a more complex DFA

- DFA has more states
- DFA has same cost per transition (*or per character*)
- DFA has same basic implementation

Tighter register specification (continued)

The DFA for

$Register \rightarrow \underline{r}\ (\ (\underline{0}|\underline{1}|\underline{2})\ (Digit\ |\ \varepsilon)\ |\ (\underline{4}|\underline{5}|\underline{6}|\underline{7}|\underline{8}|\underline{9})\ |\ (\underline{3}|\underline{30}|\underline{31})\ )$



- Accepts a more constrained set of register names
- Same set of actions, more states

Tighter register specification (continued)

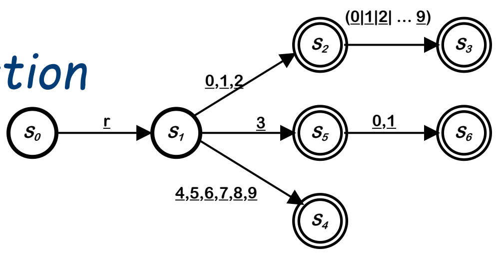
δ	r	0,1	2	3	4-9	All others
s_0	s_1	s_e	s_e	s_e	s_e	s_e
s_1	s_e	s_2	s_2	s_5	s_4	s_e
s_2	s_e	s_3	s_3	s_3	s_3	s_e
s_3	s_e	s_e	s_e	s_e	s_e	s_e
s_4	s_e	s_e	s_e	s_e	s_e	s_e
s_5	s_e	s_6	s_e	s_e	s_e	s_e
s_6	s_e	s_e	s_e	s_e	s_e	s_e
s_e	s_e	s_e	s_e	s_e	s_e	s_e

This table runs in the same skeleton recognizer

This table uses the same O(1) time per character

The extra precision costs us table space, not time

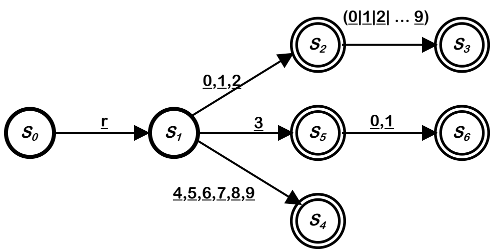
Table encoding RE for the tighter register specification



Tighter register specification (continued)

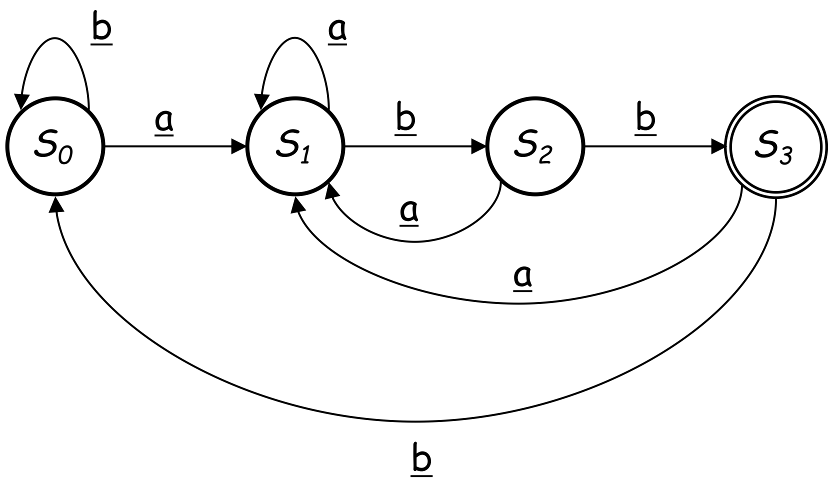
State Action	r	0,1	2	3	4,5,6 7,8,9	other
0	1 *start*	e	e	e	e	e
1	e	2 *add*	2 *add*	5 *add*	4 *add*	e
2	e	3 *add*	3 *add*	3 *add*	3 *add*	e *exit*
3,4	e	e	e	e	e	e *exit*
5	e	6 *add*	e	e	e	e *exit*
6	e	e	e	e	e	x *exit*
e	e	e	e	e	e	e

We care about path lengths (time) and finite size of set of states (representability), but we don't worry (much) about number of states.



Non-deterministic Finite Automata

What about an RE such as (a | b)* abb ?



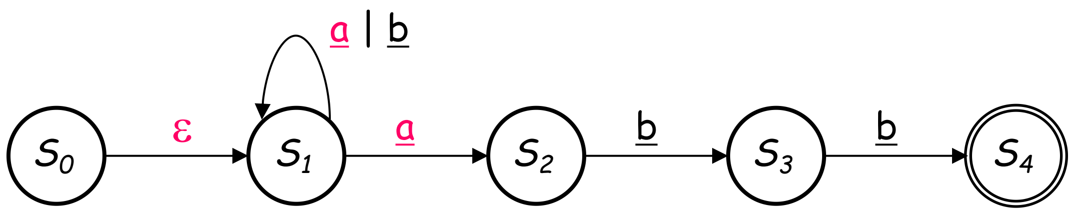
Each RE corresponds to a *deterministic finite automaton* (DFA)

- We know a DFA exists for each RE
- The DFA may be hard to build directly
- Automatic techniques will build it for us …

Nothing here that would change the O(1) cost per transition

Non-deterministic Finite Automata

Here is a simpler RE for (a | b)* abb



This recognizer is more intuitive

- Structure seems to follow the RE's structure

This recognizer is not a DFA

- S_0 has a transition on ε
- S_1 has two transitions on a

This is a *non-deterministic finite automaton* (NFA)

This NFA needs one more transition, at O(1) cost per transition

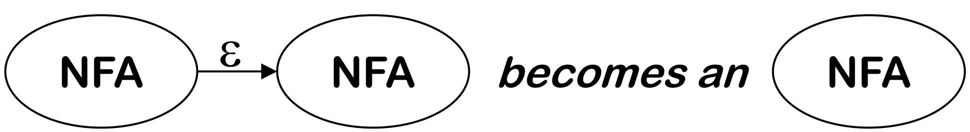
Non-deterministic Finite Automata

An NFA accepts a string *x* *iff* $\exists$ a path though the transition graph from s_0 to a final state such that the edge labels spell *x*, ignoring ε's

- Transitions on ε consume no input
- To "run" the NFA, start in s_0 and *guess* the right transition at each step
  - Always guess correctly
  - If some sequence of correct guesses accepts x then accept

Why study NFAs?

- They are the key to automating the RE$\rightarrow$DFA construction
- We can paste together NFAs with ε-transitions



Relationship between NFAs and DFAs

DFA is a special case of an NFA

- DFA has no ε transitions
- DFA's transition function is single-valued
- Same rules will work

DFA can be simulated with an NFA

– *Obviously*

NFA can be simulated with a DFA *(less obvious)*

- Simulate sets of possible states
- Possible exponential blowup in the state space
- Still, one state per character in the input stream

Rabin & Scott, 1959

Automating Scanner Construction

To convert a specification into code:

1. Write down the RE for the input language
2. Build a big NFA
3. Build the DFA that simulates the NFA
4. Systematically shrink the DFA
5. Turn it into code

Scanner generators

- Lex and Flex work along these lines
- Algorithms are well-known and well-understood
- Key issue is interface to parser *(define all parts of speech)*

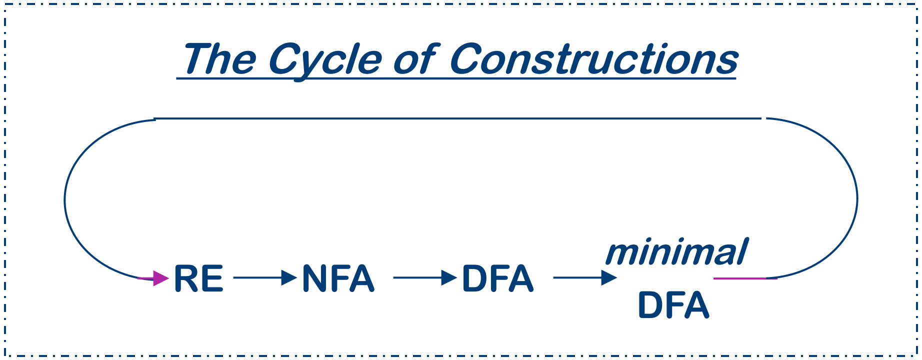
Where are we? Why are we doing this?

RE $\rightarrow$ NFA *(Thompson's construction)*

- Build an NFA for each term
- Combine them with ε-moves

NFA $\rightarrow$ DFA *(Subset construction)*

- Build the simulation



DFA $\rightarrow$ Minimal DFA

- Hopcroft's algorithm

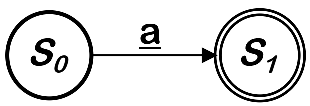
DFA $\rightarrow$ RE

- All pairs, all paths problem
- Union together paths from s_0 to a final state

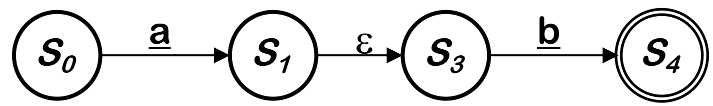
RE →NFA using Thompson's Construction

Key idea

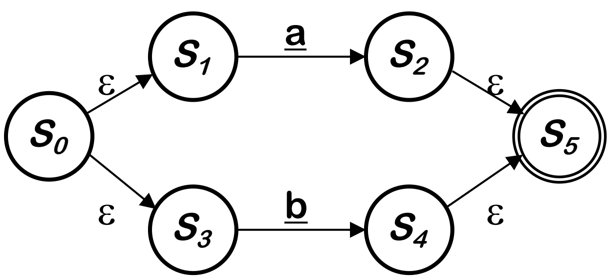
- NFA pattern for each symbol & each operator
- Join them with ε moves in precedence order



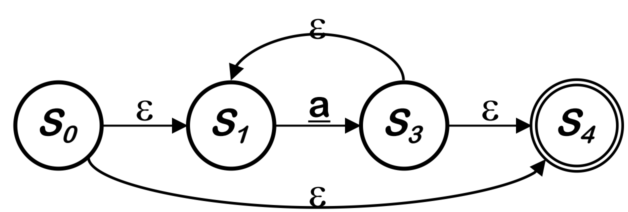
NFA for a



NFA for ab



NFA for a | b



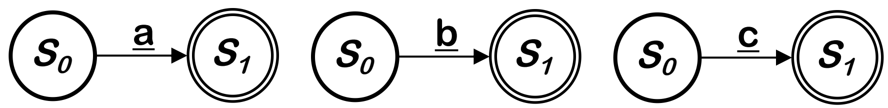
NFA for a*

Ken Thompson, CACM, 1968

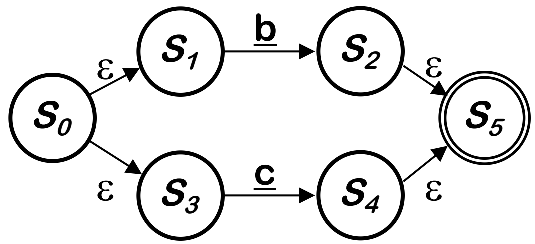
Example of Thompson's Construction

Let's try a (b | c)*

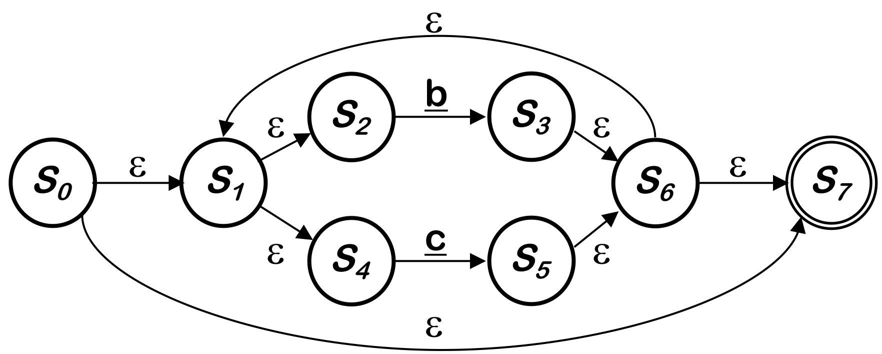
1. a, b, & c



2. b | c

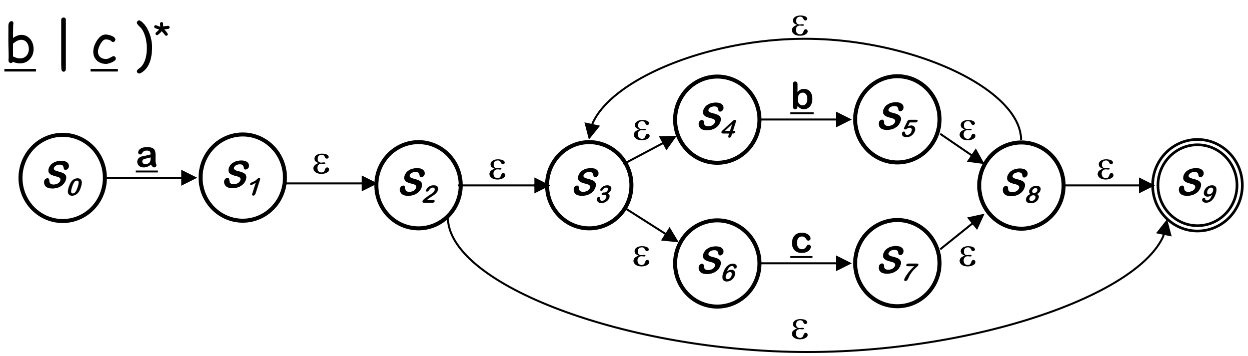


3. (b | c)*

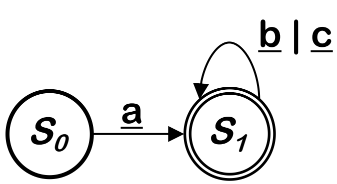


Example of Thompson's Construction *(con't)*

4. a (b | c)*



Of course, a human would design something simpler ...



But, we can automate production of the more complex NFA version ...

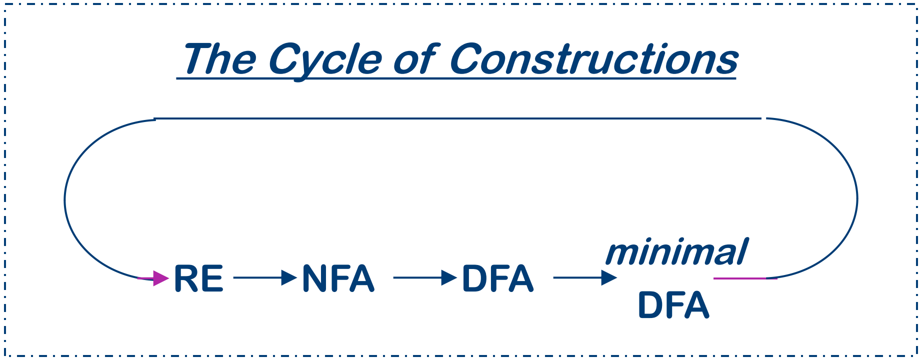
Where are we? Why are we doing this?

RE $\rightarrow$ NFA *(Thompson's construction)* ✓

- Build an NFA for each term
- Combine them with ε-moves

NFA $\rightarrow$ DFA *(subset construction)* $\Leftarrow$

- Build the simulation



DFA $\rightarrow$ Minimal DFA

- Hopcroft's algorithm

DFA $\rightarrow$ RE

- All pairs, all paths problem
- Union together paths from s_0 to a final state

NFA →DFA with Subset Construction

Need to build a simulation of the NFA

Two key functions

- *Move*(s_i , $\underline{a}$) is the set of states reachable from s_i by $\underline{a}$
- *ε-closure*(s_i) is the set of states reachable from s_i by ε

The algorithm:

- Start state derived from s_0 of the NFA
- Take its ε-closure $S_0 = \varepsilon\text{-closure}(\{s_0\})$
- Take the image of S_0, Move(S_0, α) for each $\alpha \in \Sigma$, and take its ε-closure
- Iterate until no more states are added

Sounds more complex than it is…

Rabin & Scott, 1959

NFA $\rightarrow$DFA with Subset Construction

The algorithm:

$s_0 \leftarrow \varepsilon$-closure({$n_0$})
$S \leftarrow \{ s_0 \}$
$W \leftarrow \{ s_0 \}$
while ($W \neq \emptyset$)
select and remove s from W
for each $\alpha \in \Sigma$
$t \leftarrow \varepsilon$-closure(Move(s,α))
T[s,α] $\leftarrow$ t
if ($t \notin S$) then
add t to S
add t to W

Let's think about why this works

s_0 is a set of states
S & W are sets of sets of states

The algorithm halts:

1. *S* contains no duplicates (test before adding)
2. $2^{\{\text{NFA states}\}}$ is finite
3. while loop adds to *S*, but does not remove from *S* *(monotone)*

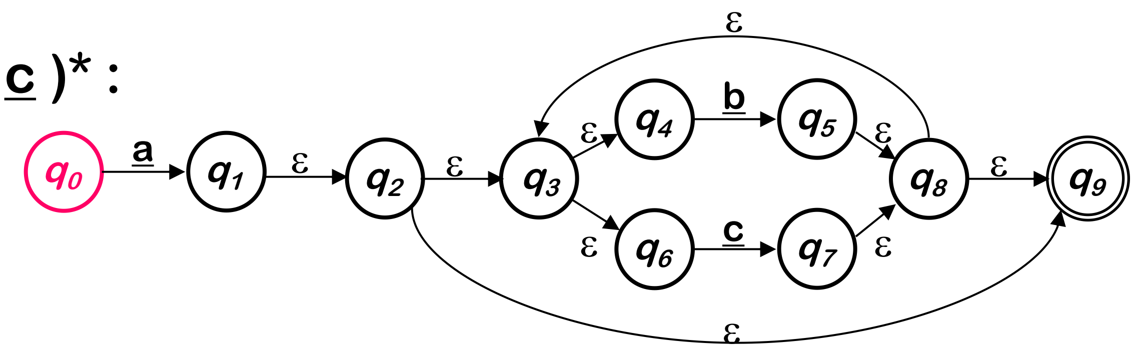
$\Rightarrow$ the loop halts

S contains all the reachable NFA states

This test is a little tricky

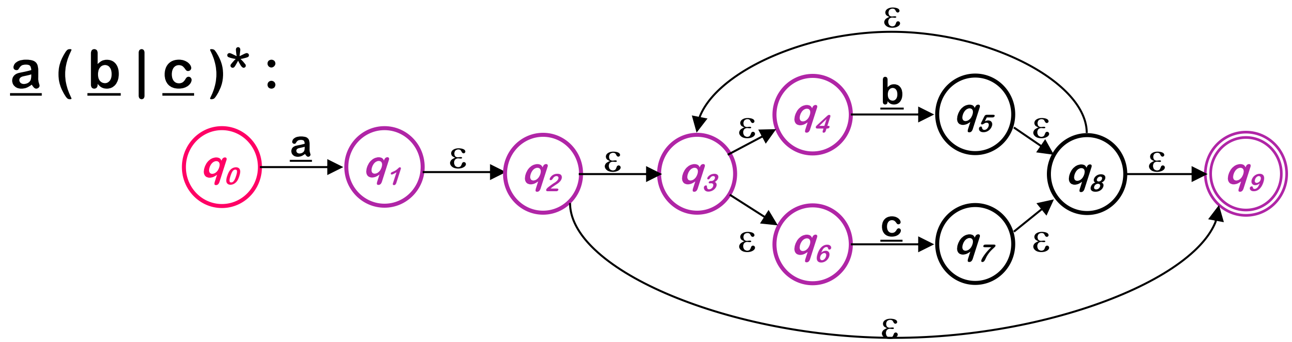
NFA →DFA with Subset Construction

a (b | c)* :



States		ε-closure(Move(s,*))		
DFA	NFA	a	b	c
s_0	q_0			

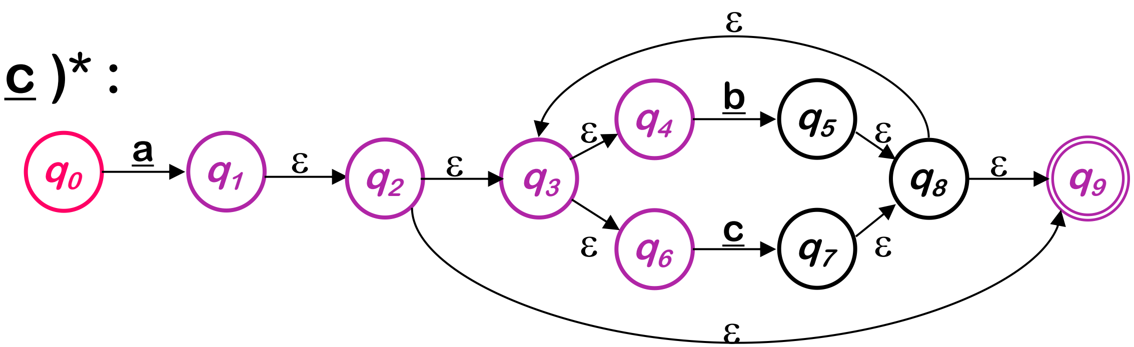
NFA →DFA with Subset Construction



States		ε-closure(Move(s,*))		
DFA	NFA	a	b	c
s_0	q_0	$q_1, q_2, q_3, q_4, q_6, q_9$		

NFA →DFA with Subset Construction

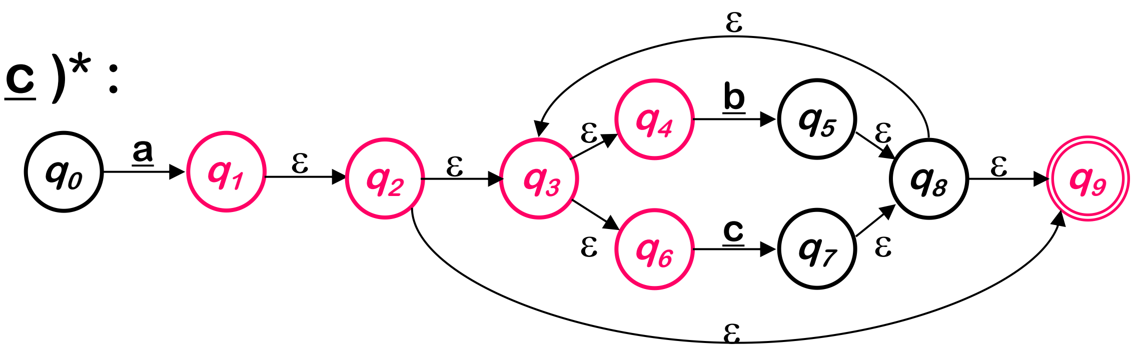
a (b | c)* :



States		ε-closure(Move(s,*))		
DFA	NFA	a	b	c
s_0	q_0	$q_1, q_2, q_3, q_4, q_6, q_9$	none	none

NFA →DFA with Subset Construction

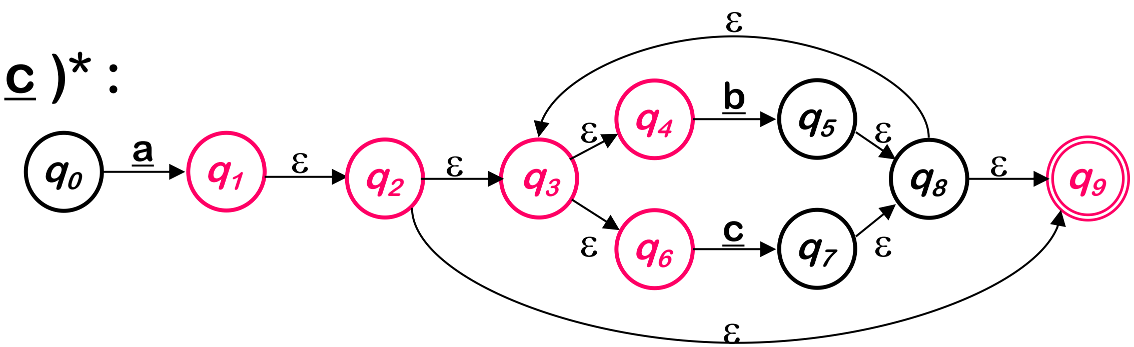
a (b | c)* :



States		ε-closure(Move(s,*))		
DFA	NFA	a	b	c
s_0	q_0	$q_1, q_2, q_3, q_4, q_6, q_9$	*none*	*none*
s_1	$q_1, q_2, q_3, q_4, q_6, q_9$			

NFA →DFA with Subset Construction

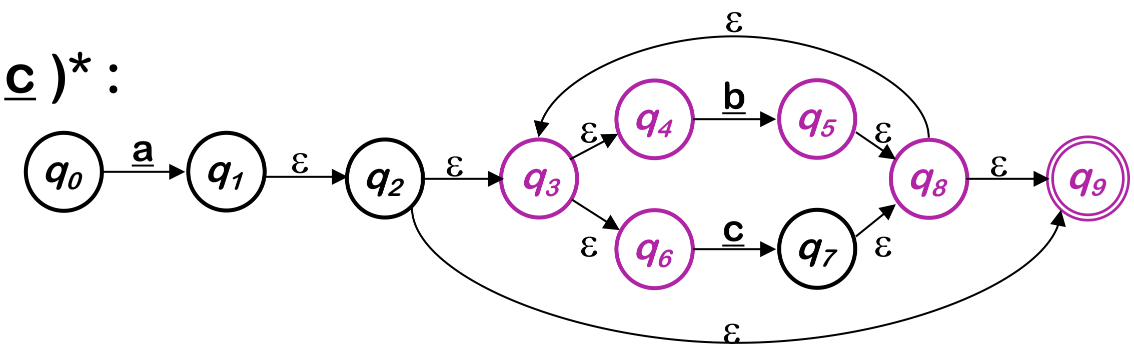
a (b | c)* :



States		ε-closure(Move(s,*))		
DFA	NFA	a	b	c
s_0	q_0	$q_1, q_2, q_3, q_4, q_6, q_9$	none	none
s_1	$q_1, q_2, q_3, q_4, q_6, q_9$	none		

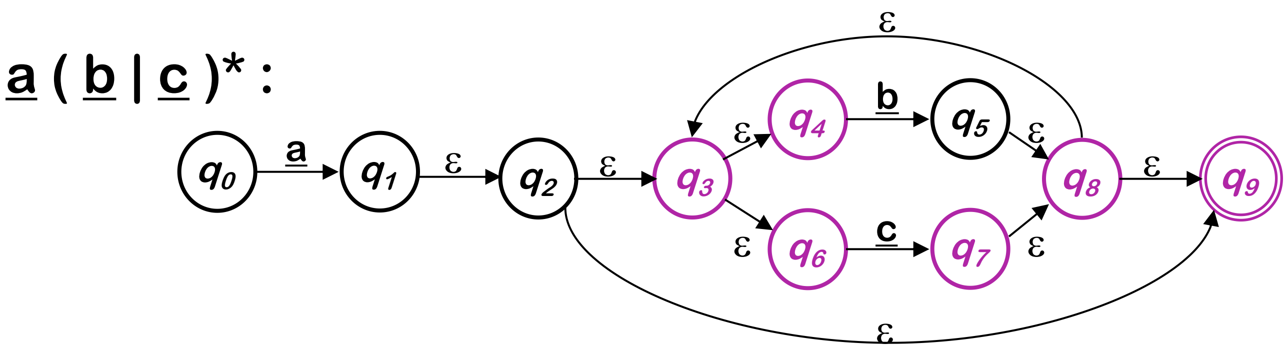
NFA →DFA with Subset Construction

a (b | c)* :



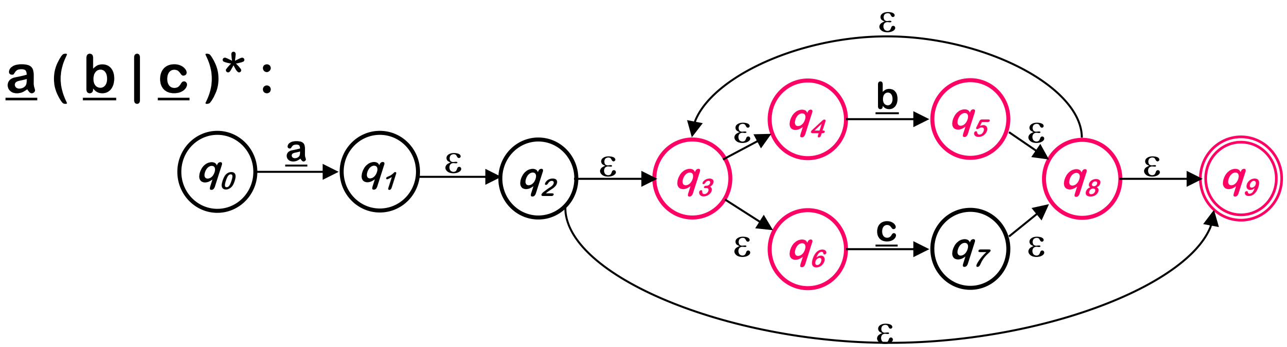
States		ε-closure(Move(s,*))		
DFA	NFA	a	b	c
s_0	q_0	q_1, q_2, q_3, q_4, q_6, q_9	none	none
s_1	q_1, q_2, q_3, q_4, q_6, q_9	none	q_5, q_8, q_9, q_3, q_4, q_6	

NFA →DFA with Subset Construction



States		ε-closure(Move(s,*))		
DFA	NFA	a	b	c
s_0	q_0	$q_1, q_2, q_3, q_4, q_6, q_9$	none	none
s_1	$q_1, q_2, q_3, q_4, q_6, q_9$	none	$q_5, q_8, q_9, q_3, q_4, q_6$	$q_7, q_8, q_9, q_3, q_4, q_6$

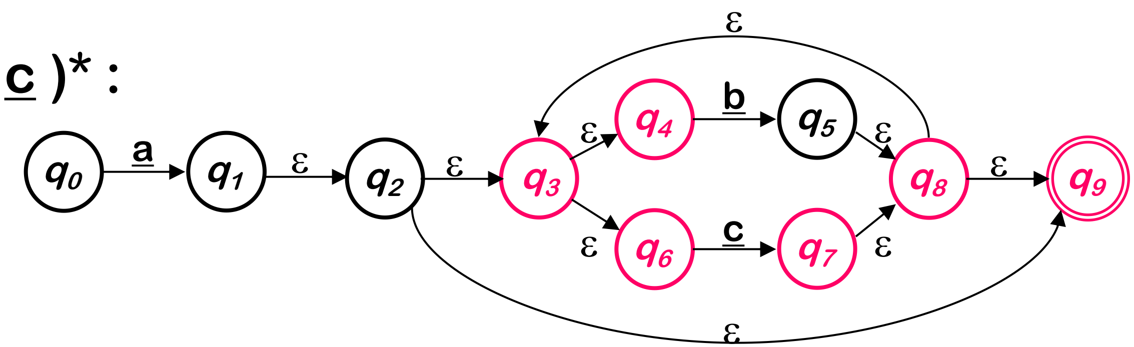
NFA →DFA with Subset Construction



States		ε-closure(Move(s,*))		
DFA	NFA	a	b	c
s_0	q_0	$q_1, q_2, q_3, q_4, q_6, q_9$	none	none
s_1	$q_1, q_2, q_3, q_4, q_6, q_9$	none	$q_5, q_8, q_9, q_3, q_4, q_6$	$q_7, q_8, q_9, q_3, q_4, q_6$
s_2	$q_5, q_8, q_9, q_3, q_4, q_6$			

NFA →DFA with Subset Construction

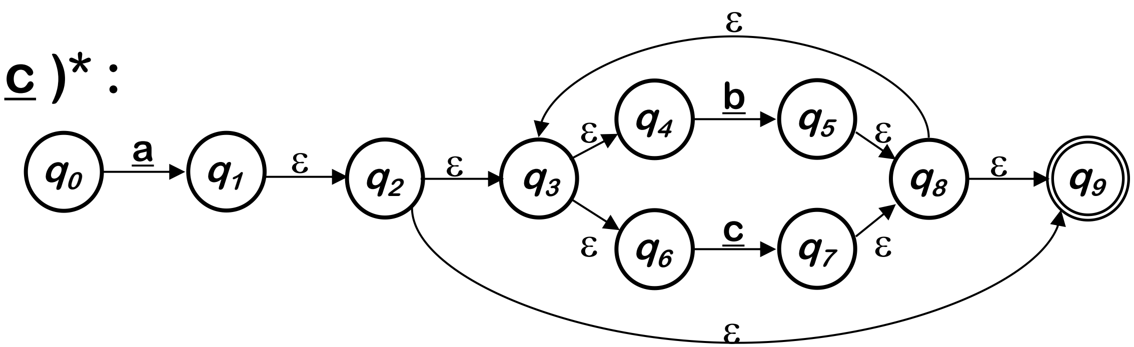
a (b | c)* :



States		ε-closure(Move(s,*))		
DFA	NFA	a	b	c
s_0	q_0	$q_1, q_2, q_3, q_4, q_6, q_9$	none	none
s_1	$q_1, q_2, q_3, q_4, q_6, q_9$	none	$q_5, q_8, q_9, q_3, q_4, q_6$	$q_7, q_8, q_9, q_3, q_4, q_6$
s_2	$q_5, q_8, q_9, q_3, q_4, q_6$			
s_3	$q_7, q_8, q_9, q_3, q_4, q_6$			

NFA →DFA with Subset Construction

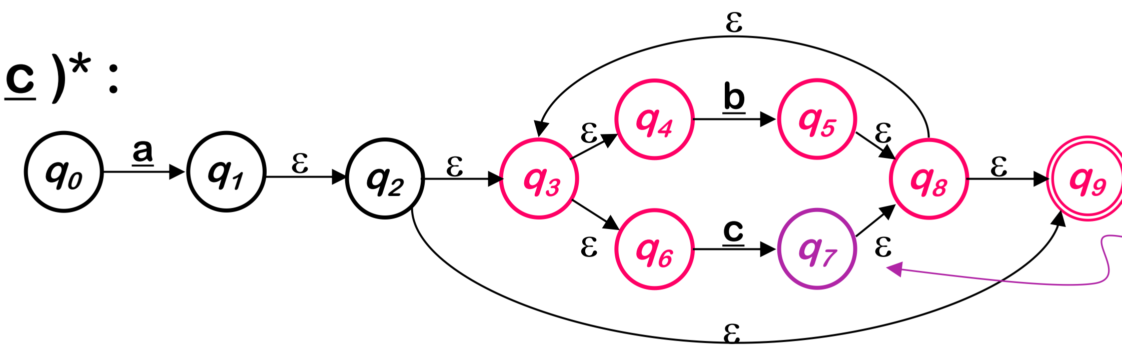
a (b | c)* :



States		ε-closure(Move(s,*))		
DFA	NFA	a	b	c
s_0	q_0	$q_1, q_2, q_3, q_4, q_6, q_9$	none	none
s_1	$q_1, q_2, q_3, q_4, q_6, q_9$	none	$q_5, q_8, q_9, q_3, q_4, q_6$	$q_7, q_8, q_9, q_3, q_4, q_6$
s_2	$q_5, q_8, q_9, q_3, q_4, q_6$	none		
s_3	$q_7, q_8, q_9, q_3, q_4, q_6$	none		

NFA →DFA with Subset Construction

a (b | c)* :

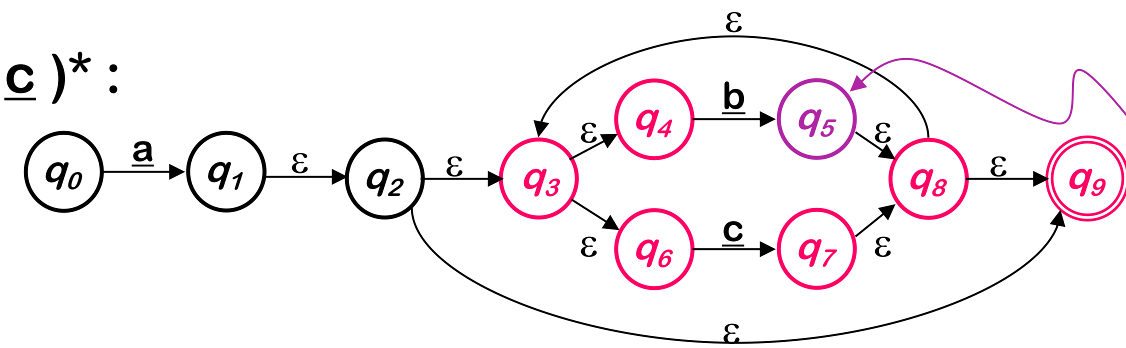


q_7 *is the core state of* s_3

States		ε-closure(Move(s,*))		
DFA	NFA	a	b	c
s_0	q_0	$q_1, q_2, q_3, q_4, q_6, q_9$	none	none
s_1	$q_1, q_2, q_3, q_4, q_6, q_9$	none	$q_5, q_8, q_9, q_3, q_4, q_6$	$q_7, q_8, q_9, q_3, q_4, q_6$
s_2	$q_5, q_8, q_9, q_3, q_4, q_6$	none	s_2	s_3
s_3	$q_7, q_8, q_9, q_3, q_4, q_6$	none		

NFA →DFA with Subset Construction

a (b | c)* :

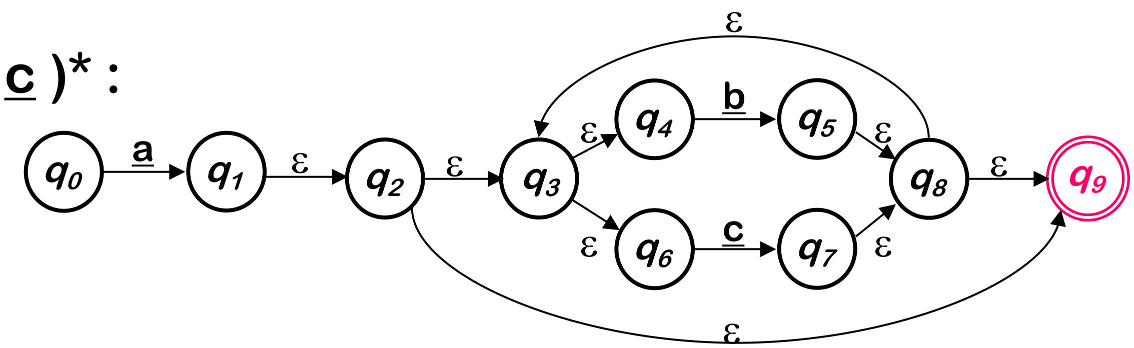


q_5 is the core state of s_2

States		ε-closure(Move(s,*))		
DFA	NFA	<u>a</u>	<u>b</u>	<u>c</u>
s_0	q_0	$q_1, q_2, q_3, q_4, q_6, q_9$	*none*	*none*
s_1	$q_1, q_2, q_3, q_4, q_6, q_9$	*none*	$q_5, q_8, q_9, q_3, q_4, q_6$	$q_7, q_8, q_9, q_3, q_4, q_6$
s_2	$q_5, q_8, q_9, q_3, q_4, q_6$	*none*	s_2	s_3
s_3	$q_7, q_8, q_9, q_3, q_4, q_6$	*none*	s_2	s_3

NFA →DFA with Subset Construction

a (b | c)* :

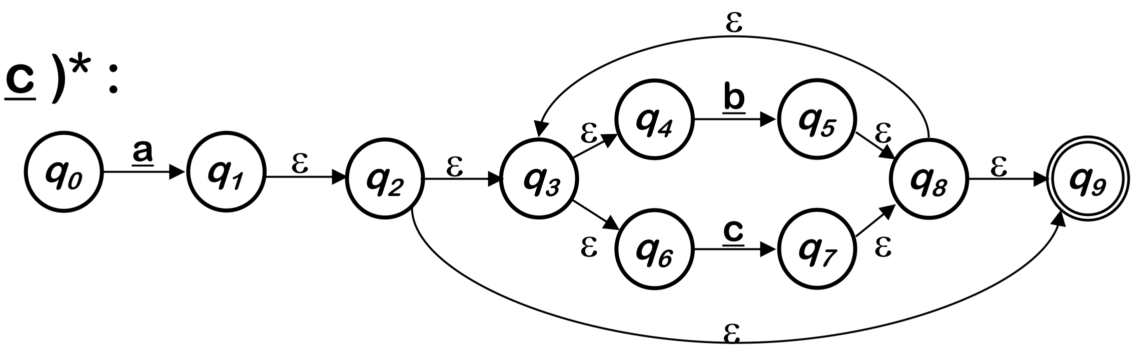


States		ε-closure(Move(s,*))		
DFA	NFA	a	b	c
s_0	q_0	$q_1, q_2, q_3, q_4, q_6, q_9$	*none*	*none*
s_1	$q_1, q_2, q_3, q_4, q_6, q_9$	*none*	$q_5, q_8, q_9, q_3, q_4, q_6$	$q_7, q_8, q_9, q_3, q_4, q_6$
s_2	$q_5, q_8, q_9, q_3, q_4, q_6$	*none*	s_2	s_3
s_3	$q_7, q_8, q_9, q_3, q_4, q_6$	*none*	s_2	s_3

Final states because of q_9

NFA →DFA with Subset Construction

a (b | c)* :

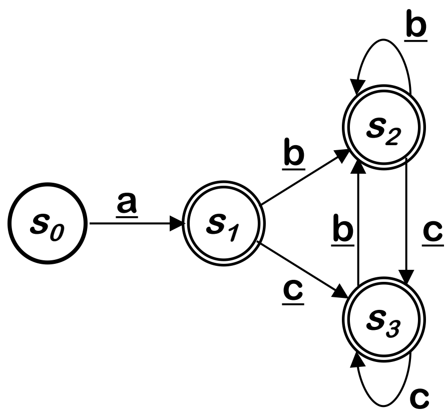


States		ε-closure(Move(s,*))		
DFA	NFA	a	b	c
s_0	q_0	s_1	*none*	*none*
s_1	$q_1, q_2, q_3, q_4, q_6, q_9$	*none*	s_2	s_3
s_2	$q_5, q_8, q_9, q_3, q_4, q_6$	*none*	s_2	s_3
s_3	$q_7, q_8, q_9, q_3, q_4, q_6$	*none*	s_2	s_3

Transition table for the DFA

NFA →DFA with Subset Construction

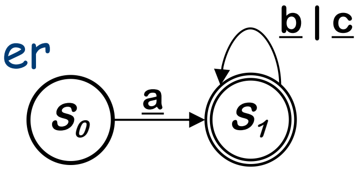
The DFA for a (b | c)*



	a	b	c
s_0	s_1	*none*	*none*
s_1	*none*	s_2	s_3
s_2	*none*	s_2	s_3
s_3	*none*	s_2	s_3

- Much smaller than the NFA (no ε-transitions)
- All transitions are deterministic
- Use same code skeleton as before

But, remember our goal:



Where are we? Why are we doing this?

RE $\rightarrow$ NFA *(Thompson's construction)* ✓

- Build an NFA for each term
- Combine them with ε-moves

NFA $\rightarrow$ DFA *(subset construction)* ✓

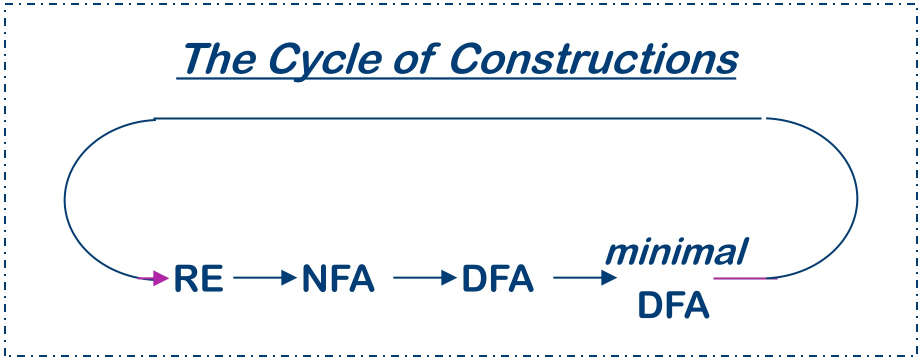
- Build the simulation

DFA $\rightarrow$ Minimal DFA $\Leftarrow$

- Hopcroft's algorithm

DFA $\rightarrow$ RE

- All pairs, all paths problem
- Union together paths from s_0 to a final state

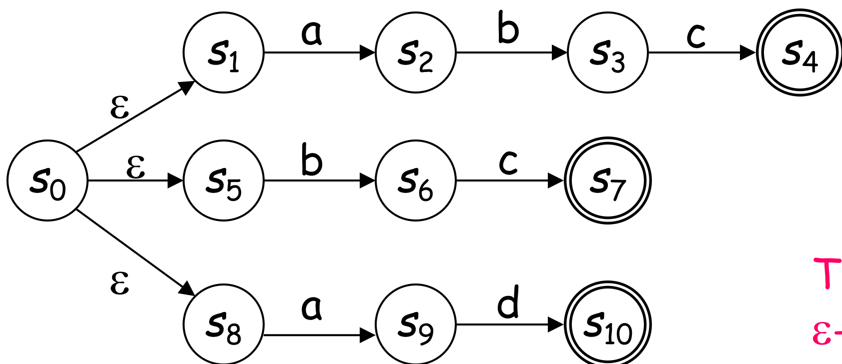


Not enough time to teach Hopcroft's algorithm today

Alternative Approach to DFA Minimization

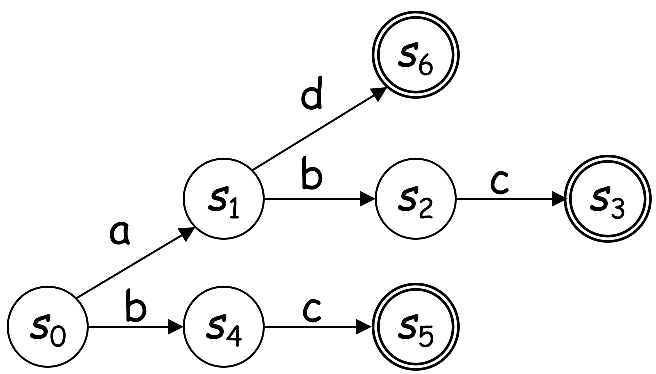
The Intuition

- The subset construction merges prefixes in the NFA



abc | bc | ad

Thompson's construction would leave ε-transitions between each single-character automaton



Subset construction eliminates ε-transitions and merges the paths for a. It leaves duplicate tails, such as bc.

Alternative Approach to DFA Minimization

Idea: use the subset construction twice

- For an NFA *N*
  - Let *reverse(N)* be the NFA constructed by making initial states final (& vice-versa) and reversing the edges
  - Let *subset(N)* be the DFA that results from applying the subset construction to *N*
  - Let *reachable(N)* be *N* after removing all states that are not reachable from the initial state
- Then,

reachable(subset(reverse[reachable(subset(reverse(N))]))

is the minimal DFA that implements *N* [Brzozowski, 1962]

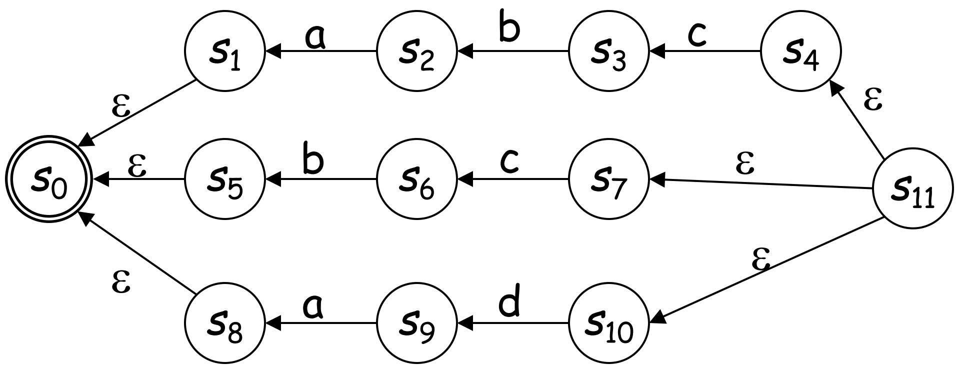
This result is not intuitive, but it is true.

Neither algorithm dominates the other.

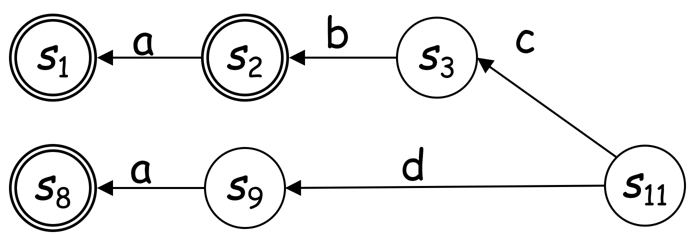
Alternative Approach to DFA Minimization

Step 1

- The subset construction on *reverse(NFA)* merges suffixes in original NFA



Reversed NFA

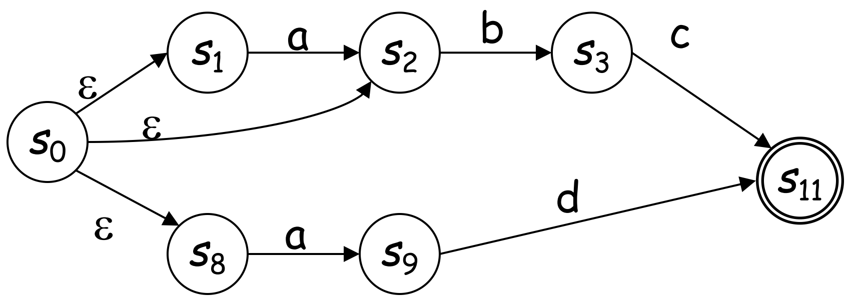


subset(reverse(NFA))

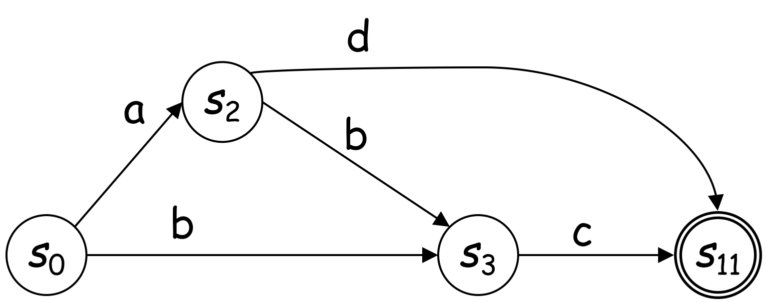
Alternative Approach to DFA Minimization

Step 2

- Reverse it again & use subset to merge prefixes …



Reverse it, again



Minimal DFA

And subset it, again

The Cycle of Constructions