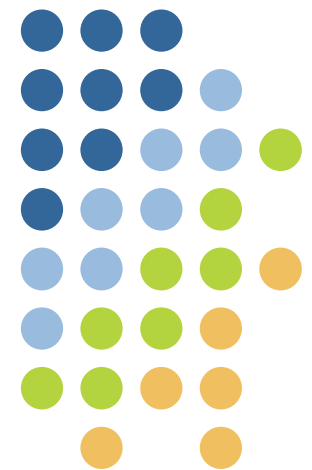


COMP 181

Lecture 3

More scanning

Monday, Sept. 19, 2005





Problem set

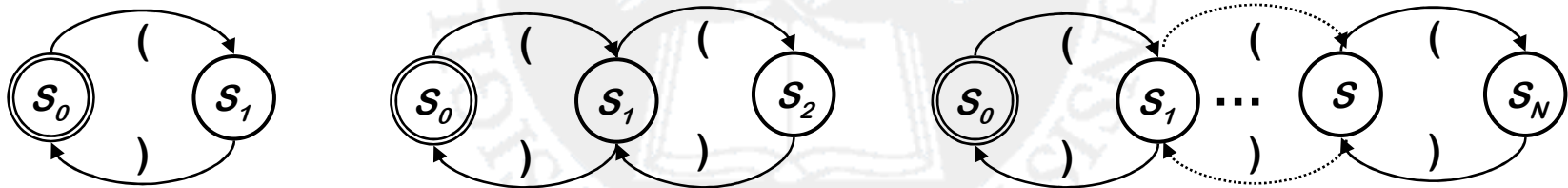
- #1 – no (see below)
- #2 – yes.
 - *Trivial*: any language of finite strings is regular
 - Example: just enumerate all the strings
- #3 – yes.
 - Finite number of matches: () or (()) or ((()))
 - Strings are just repetitions of three patterns
- #4 – no.
 - This is the key problem...





Problem set

- Matching parentheses
 - Not a regular language
 - ➔ No RE, no DFA can represent this language
- Intuition:
 - How to make sure numbers match?

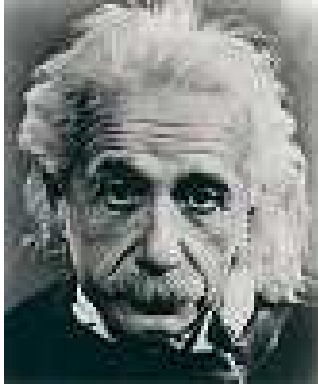


- Not a finite automaton
- Proof: *Pumping lemma*





Introduction



"God does not play dice with the universe."

What does this quote refer to?

- Newtonian physics: nature is deterministic
- Quantum mechanics: nature is non-deterministic
- Einstein didn't like this
- We will take Einstein's view...





Scanner construction

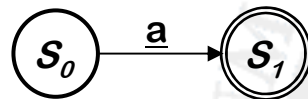
- [0] Define tokens as regular expressions
- [1] Construct NFA for all REs
 - Connect REs with ϵ transitions
 - *Thompson's construction*
- [2] Convert NFA into a DFA
 - DFA is a simulation of NFA
 - Possible much larger than NFA
- [3] Minimize the DFA
 - *Hopcroft's algorithm*
- [4] Generate implementation



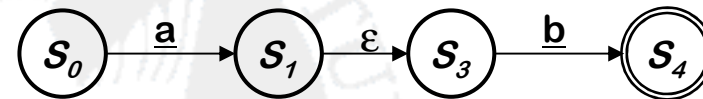


[1] Thompson's construction

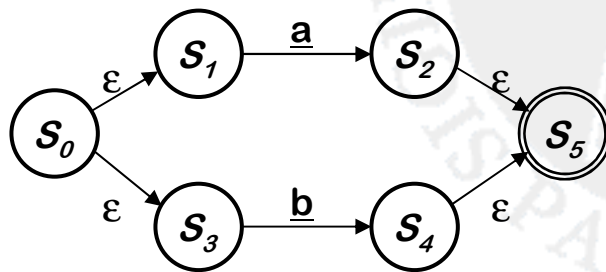
- **Key idea:**
 - NFA pattern for each RE operator
 - Join them together using ϵ transitions



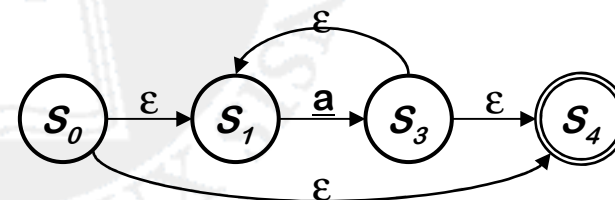
NFA for a



NFA for ab



NFA for a | b



NFA for a*

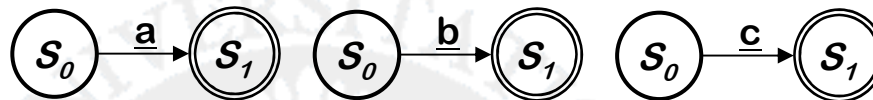




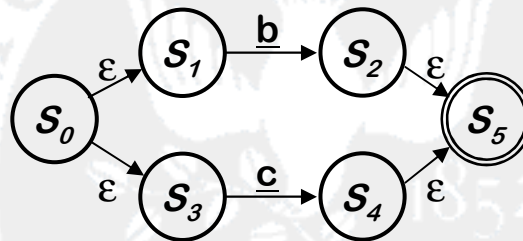
Example

Regular expression: $\underline{a} (\underline{b} \mid \underline{c})^*$

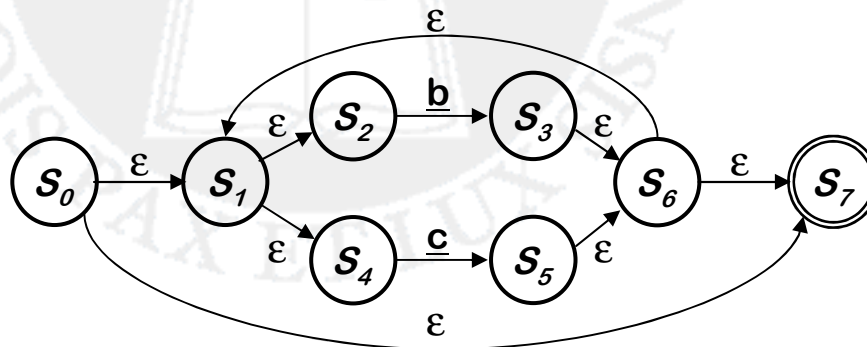
- \underline{a} , \underline{b} , & \underline{c}



- $\underline{b} \mid \underline{c}$



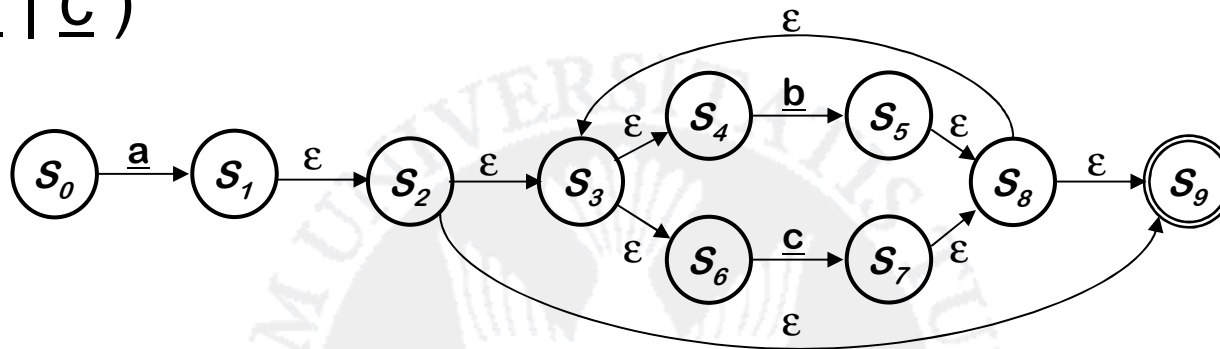
- $(\underline{b} \mid \underline{c})^*$



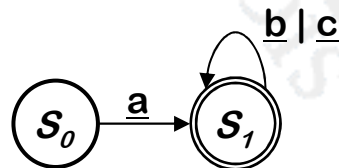


Example

- $\underline{a} (\underline{b} | \underline{c})^*$



- Note: a human could design something simpler...



But, we can automate production of the more complex one ...





Problem

- How to implement NFA scanner code?
 - Non-determinism is a problem
 - Explore all possible paths?

- Observation:

We can build a DFA that simulates the NFA

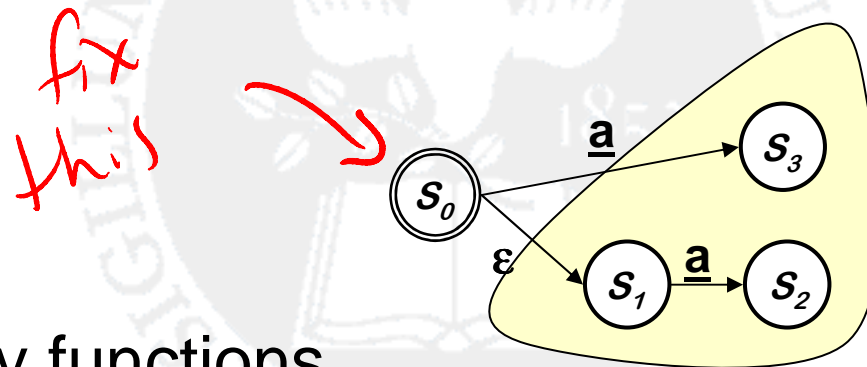
- Accepts the same language
- Explores all paths simultaneously





[2] NFA to DFA

- Subset construction
- Algorithm
 - **Intuition**: each DFA state represents the *possible* states reachable after one input in the NFA



- Two key functions
 - **next(s_i , \underline{a})** – the set of states reachable from s_i on \underline{a}
 - **ϵ -closure(s_i)** – the set of states reachable from s_i on ϵ





Subset construction

- Algorithm

Start with the e-closure over the initial state

Build a set of subsets of NFA states

```
 $s_0 \leftarrow \varepsilon\text{-closure}(q_0)$   
 $S \leftarrow \{s_0\}$   
 $worklist \leftarrow \{s_0\}$   
while ( worklist is not empty )  
  remove s from worklist  
  for each  $\alpha \in \Sigma$   
     $t \leftarrow \varepsilon\text{-closure}(next(s, \alpha))$   
    if ( $t \notin S$ ) then  
      add t to S  
      add t to worklist  
      add transition ( $s, \alpha, t$ )
```

Initialize the worklist to this one subset

While there are more subsets on the worklist, remove the next subset

Apply each input symbol to the set of NFA states to produce a new set.

If we haven't seen that subset before, add it to S and the worklist, and record the set-to-set transition





Does it work?

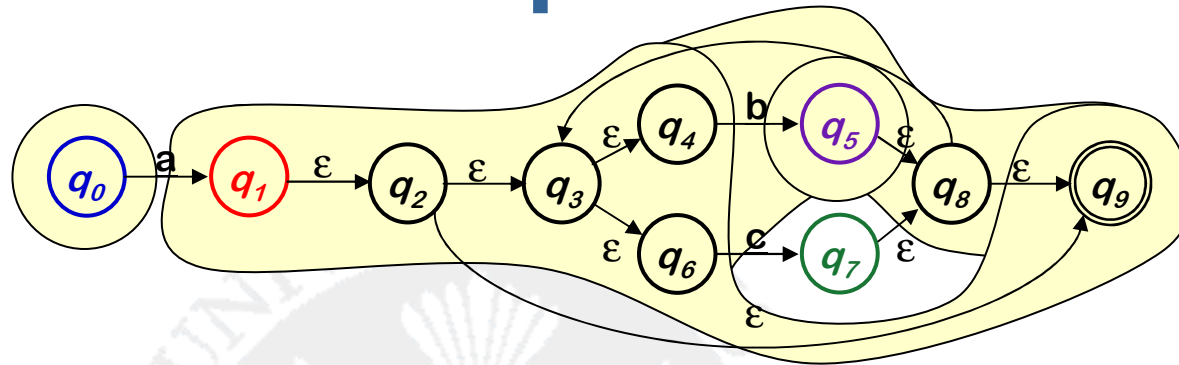
- The algorithm halts
 - S contains no duplicate subsets
 - $2^{|\text{NFA}|}$ is finite
 - Main loop adds to S, but does not remove
 - It is a **monotone** function*
- S contains all the reachable NFA states
 - Tries all input symbols, builds all NFA configurations*
- **Note:** important class of compiler algorithms
 - **Fixpoint** computation
 - Monotonic update function
 - Convergence is guaranteed





NFA to DFA example

$a(b|c)^*$:



ϵ -closure(next(s, α))

Subsets S	NFA states	<u>a</u>	<u>b</u>	<u>c</u>
s_0	q_0	$q_1, q_2, q_3, q_4, q_6, q_9$	--	--
s_1	$q_1, q_2, q_3, q_4, q_6, q_9$	--	$q_5, q_8, q_9, q_3, q_4, q_6$	$q_7, q_8, q_9, q_3, q_4, q_6$
s_2	$q_5, q_8, q_9, q_3, q_4, q_6$	--	same as s_2	same as s_3
s_3	$q_7, q_8, q_9, q_3, q_4, q_6$	--	same as s_2	same as s_3

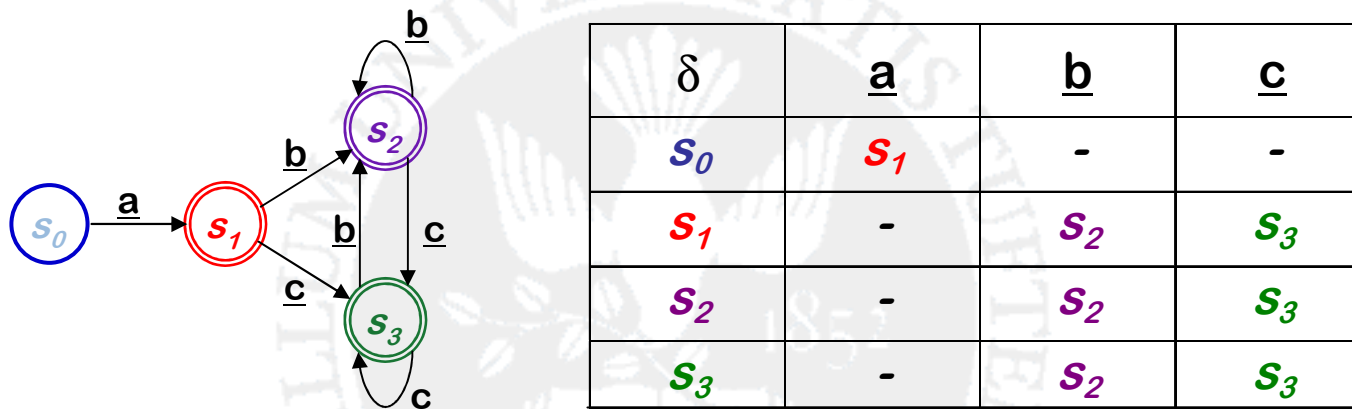
Accepting states





NFA to DFA example

- Convert each subset in S into a state:



- All transitions are deterministic
- Use same code engine to execute
- Smaller than NFA, but still bigger than necessary





[3] DFA minimization

- Hopcroft's algorithm
 - Discover sets of *equivalent* states in DFA
 - Represent each set with a single state
 - Two states are equivalent iff:
 - The set of paths leading to them are the same
 - For all input symbols, transitions lead to equivalent states
- ➔ This is the key to the algorithm





DFA minimization

- A partition P of the states S
 - Each $s \in S$ is in exactly one set $p_i \in P$
 - Idea:
If two states s and t transition to different partitions, then they must be in different partitions
- Algorithm: iteratively partition the DFA's states
 - Group states into maximal size sets, *optimistically*
 - Iteratively subdivide those sets, as needed
 - States that remain grouped together are equivalent





DFA minimization

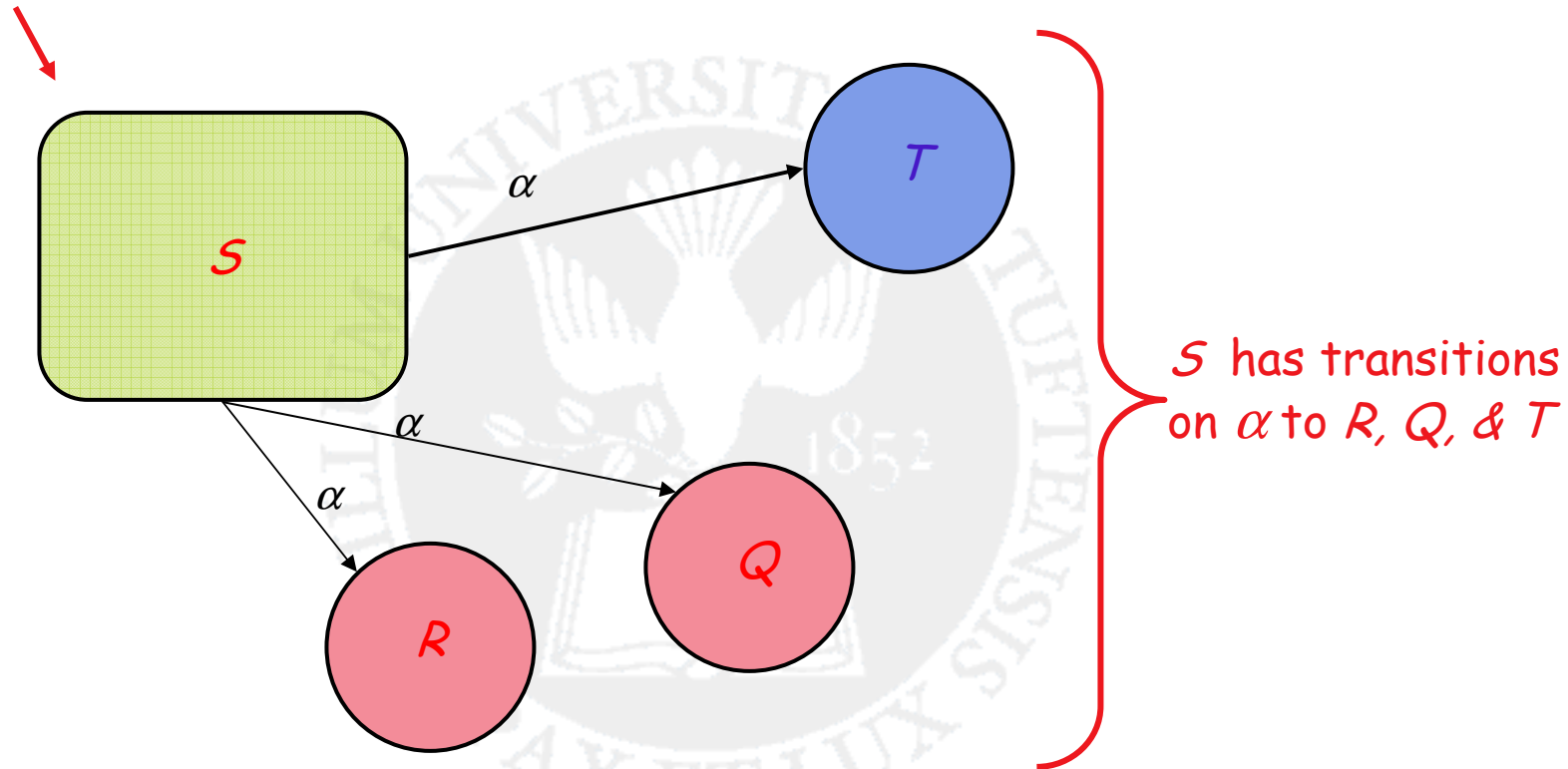
- Details:
 - Given DFA $(S, \Sigma, \delta, s_0, F)$
 - Initial partition: $P_0 = \{F, S-F\}$
Intuition: final states are always different
- Splitting a set around symbol a
 - Assume s_a & $s_b \in p_i$, and $\delta(s_a, \underline{a}) = s_x$, & $\delta(s_b, \underline{a}) = s_y$
 - Split p_i if:
 - If s_x & s_y are not in the same set
 - If s_a has a transition on a, but s_b does not
 - Intuition: one state in DFA cannot have two transitions on a





Splitting S around α

Original set S

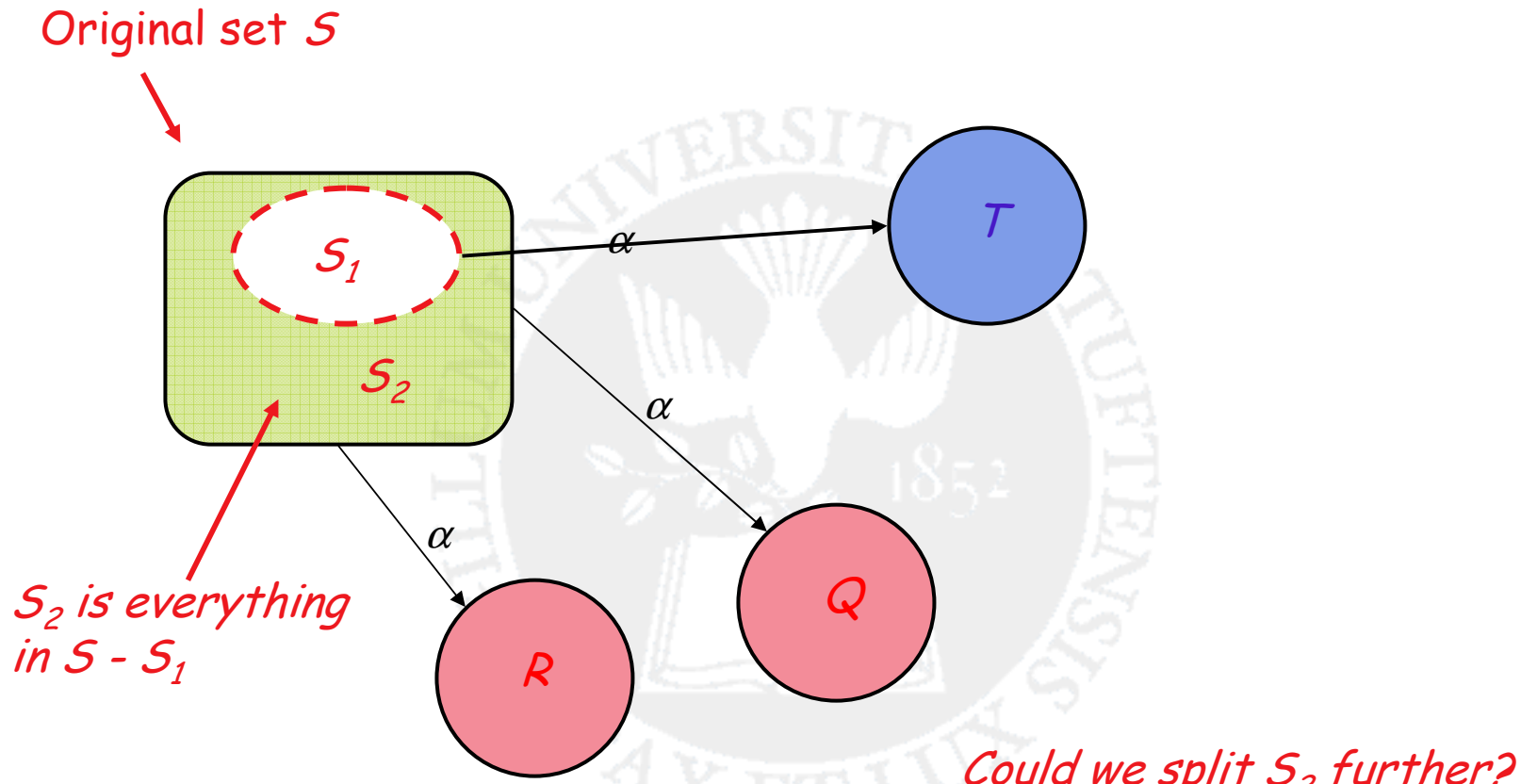


The algorithm partitions S around α





Splitting S around α



Yes, but it does not help asymptotically





DFA minimization algorithm

```
P ← { F, {Q-F}  
while (P is still changing)  
  T ← {}  
  for each set S ∈ P  
    for each α ∈ Σ  
      partition S by α  
      into S1 and S2  
      T ← T ∪ S1 ∪ S2  
  if T ≠ P then  
    P ← T
```

Start with two sets: final states, everything else

Build a new partitioning

For each set and each input symbol, try to partition the set

Collect the resulting sets in a new partition, see if it's different

This is a fixed-point algorithm!





Does it work?

- Algorithm halts
 - Partition $P \in 2^S$
 - Start off with 2 subsets of S $\{F\}$ and $\{S-F\}$
 - *While* loop takes $P_i \rightarrow P_{i+1}$ by splitting 1 or more sets
 - P_{i+1} is at least one step closer to partition with $|S|$ sets
 - Maximum of $|S|$ splits
- Note that
 - Partitions are never combined
 - Initial partition ensures that final states are intact





DFA minimization

Refining the algorithm

- As written, it examines every $S \in P$ on each iteration
 - This does a lot of unnecessary work
 - Only need to examine S if some T , reachable from S , has been split
- Reformulate the algorithm using a “worklist”
 - Start worklist with initial partition, F and $\{Q-F\}$
 - When it splits S into S_1 and S_2 , place S_2 on worklist

➔ This version looks at each $S \in P$ many fewer times
Well-known, widely used algorithm due to John Hopcroft





Hopcroft's algorithm

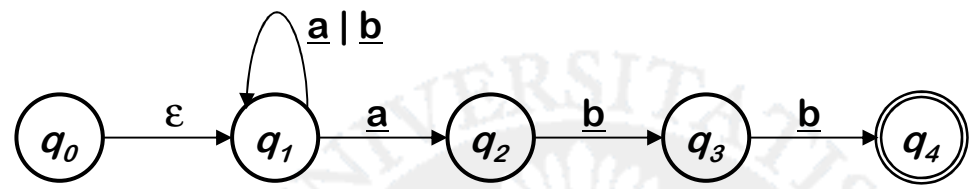
```
W ← {F, Q-F}; P ← {F, Q-F}; // W is the worklist, P the current partition
while ( W is not empty ) do begin
  select and remove S from W; // S is a set of states
  for each  $\alpha$  in  $\Sigma$  do begin
    let  $I_\alpha \leftarrow \delta_\alpha^{-1}(S)$ ; //  $I_\alpha$  is set of all states that can reach S on  $\alpha$ 
    for each R in P such that  $R \cap I_\alpha$  is not empty
      and R is not contained in  $I_\alpha$  do begin
        partition R into  $R_1$  and  $R_2$  such that  $R_1 \leftarrow R \cap I_\alpha$ ;  $R_2 \leftarrow R - R_1$ ;
        replace R in P with  $R_1$  and  $R_2$ ;
        if  $R \in W$  then replace R with  $R_1$  in W and add  $R_2$  to W;
        else if  $\|R_1\| \leq \|R_2\|$ 
          then add  $R_1$  to W;
          else add  $R_2$  to W;
        end
      end
    end
  end
end
end
```





Full example

- Remember $(\underline{a} \mid \underline{b})^* \underline{abb}$?



Our first NFA

- Applying the subset construction:

Iter.	State	Contains	ϵ -closure(move(s_i, a))	ϵ -closure(move(s_i, b))
0	s_0	q_0, q_1	q_1, q_2	q_1
1	s_1	q_1, q_2	q_1, q_2	q_1, q_3
	s_2	q_1	q_1, q_2	q_1
2	s_3	q_1, q_3	q_1, q_2	q_1, q_4
3	s_4	q_1, q_4	q_1, q_2	q_1

contains q_4
(final state)

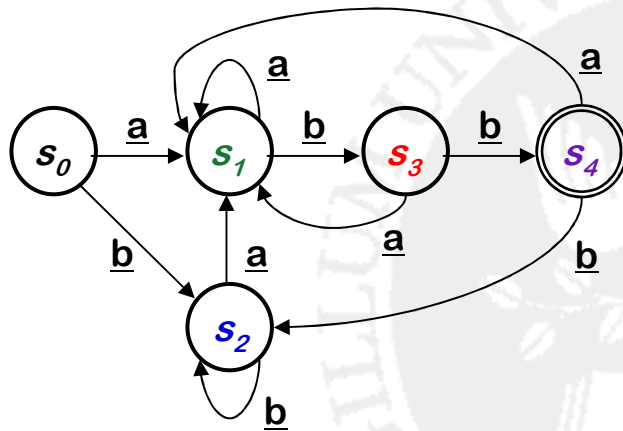
- Iteration 3 adds nothing to S , so the algorithm halts





Full example

- DFA for $(\underline{a} \mid \underline{b})^* \underline{abb}$



δ	<u>a</u>	<u>b</u>
s_0	s_1	s_2
s_1	s_1	s_3
s_2	s_1	s_2
s_3	s_1	s_4
s_4	s_1	s_2

- About the same size as NFA

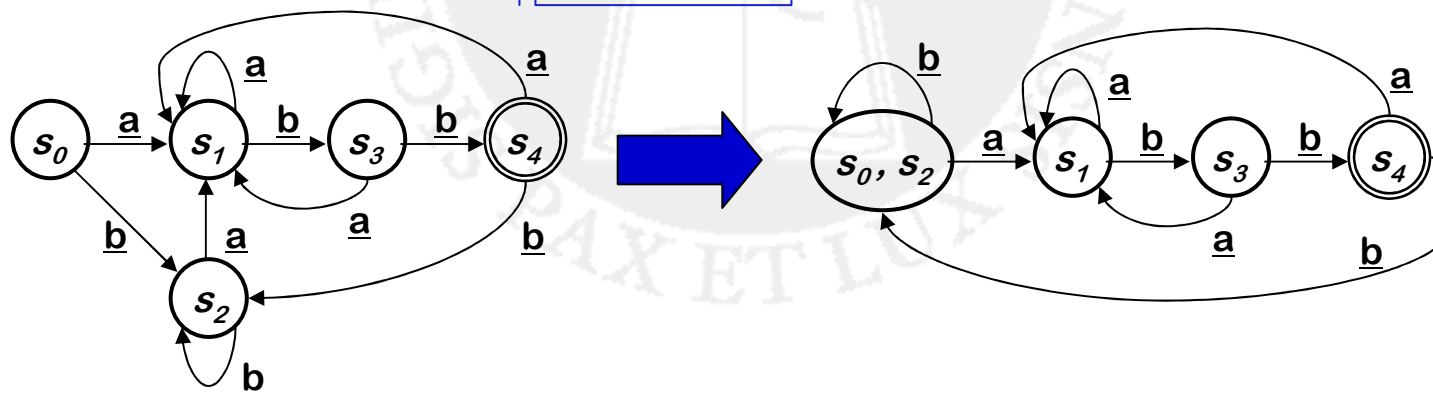




Full example – minimization

	<i>Current Partition</i>	<i>Worklist</i>	<i>s</i>	<i>Split on a</i>	<i>Split on b</i>
P_0	$\{s_4\} \{s_0, s_1, s_2, s_3\}$	$\{s_4\}$ $\{s_0, s_1, s_2, s_3\}$	$\{s_4\}$	none	$\{s_0, s_1, s_2\}$ $\{s_3\}$
P_1	$\{s_4\} \{s_3\} \{s_0, s_1, s_2\}$	$\{s_0, s_1, s_2\}$ $\{s_3\}$	$\{s_3\}$	none	$\{s_0, s_2\} \{s_1\}$
P_2	$\{s_4\} \{s_3\} \{s_1\} \{s_0, s_2\}$	$\{s_0, s_2\} \{s_1\}$	$\{s_1\}$	none	none

final state



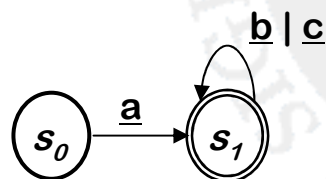
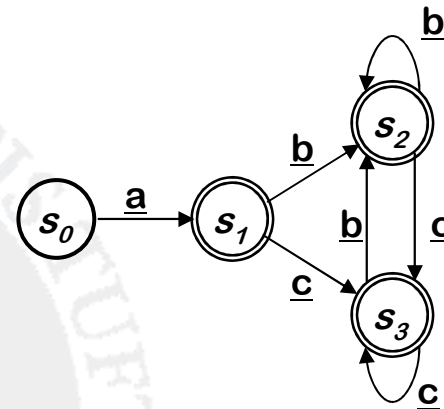


Another example

- What about $\underline{a} (\underline{b} | \underline{c})^*$

	<i>Current Partition</i>	<i>Split on</i>		
		<u>a</u>	<u>b</u>	<u>c</u>
P_0	$\{s_1, s_2, s_3\} \{s_0\}$	<i>none</i>	<i>none</i>	<i>none</i>

final states



Previously, we observed that a human would design a simpler automaton than Thompson's construction & the subset construction did.

Minimizing that DFA produces the one that a human would design!





Implementation

- Table driven
 - Read and classify character
 - Select action
 - Find the next state, assign to state variable
 - Repeat
- Alternative: direct coding
 - Each state is a chunk of code
 - Transitions test and branch directly
 - Very ugly code – but who cares?
 - Very efficient

This is how lex/flex work: states are encoded as cases in a giant switch statement





Building scanners

- The point
 - Theory lets us automate construction
 - Implementer writes down regular expressions
 - Generator does: RE \rightarrow NFA \rightarrow DFA \rightarrow code
 - Reliably produces fast, robust scanners
- Works for most modern languages
 - Think twice about language features that defeat the DFA-based scanners*





Next time...

- Grammars and parsing
- Project 1:
 - Extend a scanner for “mini Pascal”
 - I will email and post project description
- Next lecture will be Sept. 26
I'm going to a conference – PACT

