MACM 300
Formal Languages and Automata

Anoop Sarkar

`http://www.cs.sfu.ca/~anoop`

Formal Languages: Recap

- Symbols: a, b, c
- Alphabet : finite set of symbols $\Sigma = \{a, b\}$
- String: sequence of symbols   bab
- Empty string: ε Define: $\Sigma^{\varepsilon} = \Sigma \cup \{\varepsilon\}$
- Set of all strings: Σ^* cf. *The Library of Babel*, Jorge Luis Borges
- (Formal) Language: a set of strings

  $\{ a^n b^n : n > 0 \}$

Regular Languages

- The set of regular languages: each element is a regular language
- Each regular language is an example of a (formal) language, i.e. a set of strings

  e.g. $\{ a^m b^n : m, n \text{ are +ve integers} \}$

Regular Languages

- Defining the set of all regular languages:
  - The empty set and {a} for all a in Σ^{ε} are regular languages
  - If L_1 and L_2 and L are regular languages, then:

    $L_1 \cdot L_2 = \{xy \mid x \in L_1 \text{ and } y \in L_2\}$ (concatenation)

    $L_1 \cup L_2$ (union)

    $L^* = \cup_{i=0}^{\infty} L^i$ (Kleene closure)

    are also regular languages
  - There are no other regular languages

Formal Grammars

- A formal grammar is a concise description of a formal language
- A formal grammar uses a specialized syntax
- For example, a **regular expression** is a concise description of a regular language

  *(a∪b)*abb* : is the set of all strings over the alphabet {*a*, *b*} which end in *abb*

Regular Expressions: Definition

- Every symbol of $\Sigma \cup \{ \varepsilon \}$ is a regular expression
- The empty language ϕ is a regular expression
  - Note that $1^*\phi = \phi$
- If r_1 and r_2 are regular expressions, so are
  - Concatenation: $r_1 r_2$
  - Alternation: $r_1 \cup r_2$
  - Repetition: r_1^*
- Nothing else is.
  - But grouping re's is allowed: e.g. aa∪bc vs. ((aa)∪b)c

Regular Expressions: Examples

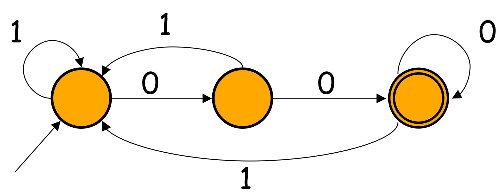
- Alphabet { 0, 1 }
- All strings that represent binary numbers divisible by 4 (but accept 0) ((0∪1)*00)|0
- All strings that do not contain "01" as a substring 1*0*

Finite Automata: Recap

- A set of states S
  - One start state q_0, zero or more final states F
- An alphabet ∑ of input symbols
- A transition function:
  - δ: S x $\Sigma \Rightarrow$ S
- Example: $\delta(1, a) = 2$

Finite Automata: Example

- What regular expression does this automaton accept?



Answer: $(0 \cup 1)^*00$

NFAs

- NFA: like a DFA, except
  - A transition can lead to more than one state, that is, $\delta: S \times \Sigma \Rightarrow 2^S$
  - One state is chosen non-deterministically
  - Transitions can be labeled with ε, meaning states can be reached without reading any input, that is,

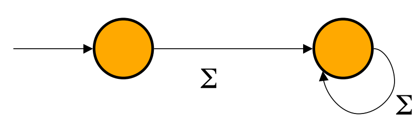
$$\delta: S \times \Sigma \cup \{ \varepsilon \} \Rightarrow 2^S$$

Thompson's construction

- Converts regexps to NFA
- Six simple rules
  - Empty language
  - Symbols
  - Empty String
  - Alternation (r_1 or r_2)
  - Concatenation (r_1 followed by r_2)
  - Repetition (r_1*)

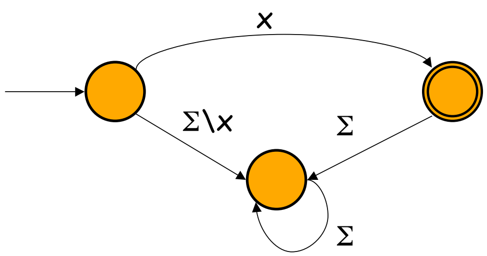
Thompson Rule 0

- For the empty language ϕ (optionally include a *sinkhole* state)



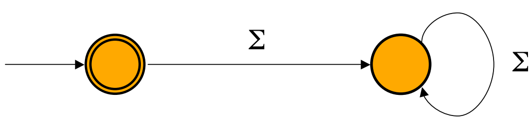
Thompson Rule 1

- For each symbol x of the alphabet, there is a NFA that accepts it (optionally include a *sinkhole* state)



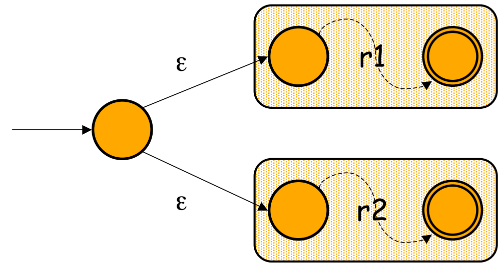
Thompson Rule 2

- There is an NFA that accepts only ε



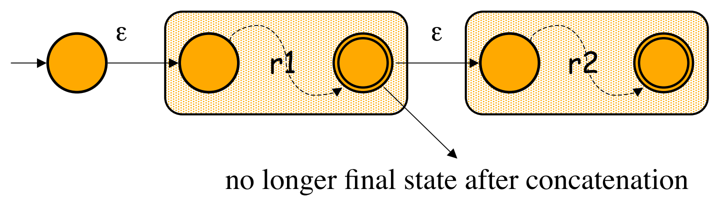
Thompson Rule 3

- Given two NFAs for r_1, r_2, there is a NFA that accepts $r_1 \cup r_2$



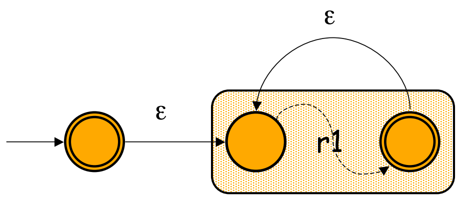
Thompson Rule 4

- Given two NFAs for r_1, r_2, there is a NFA that accepts $r_1 r_2$



Thompson Rule 5

- Given a NFA for r_1, there is an NFA that accepts r_1^*

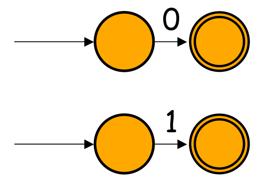


Example

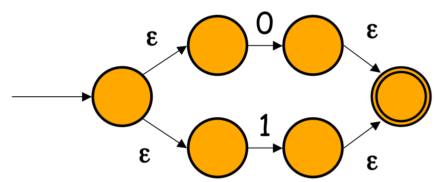
- Set of all binary strings that are divisible by four (include 0 in this set)
- Defined by the regexp: $((0\cup 1)^*00)\cup 0$
- Apply Thompson's Rules to create an NFA

Basic Blocks 0 and 1

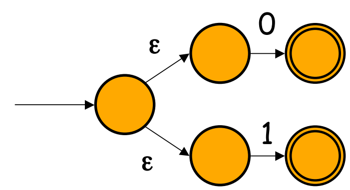
- 0
- 1

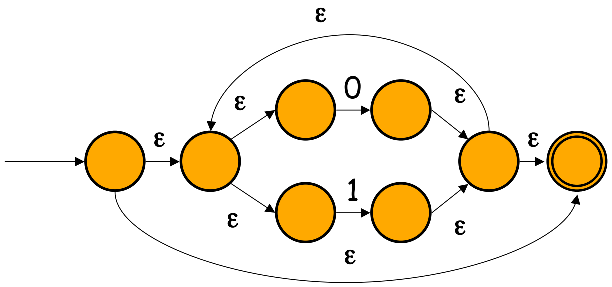


(this version does not report errors: no *sinkholes*)

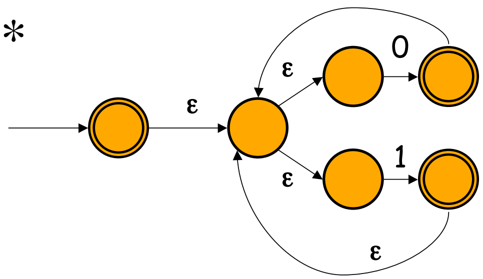


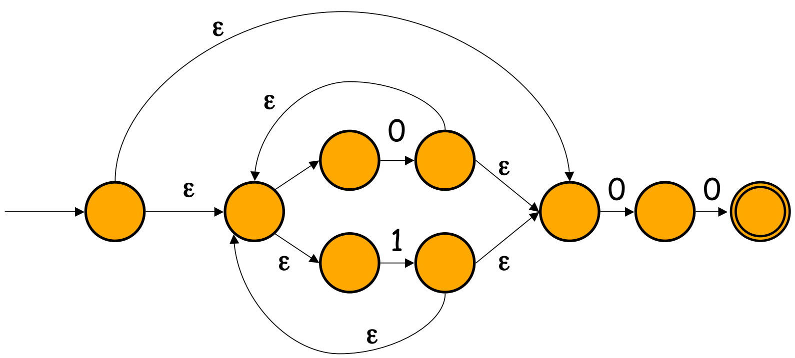
0|1



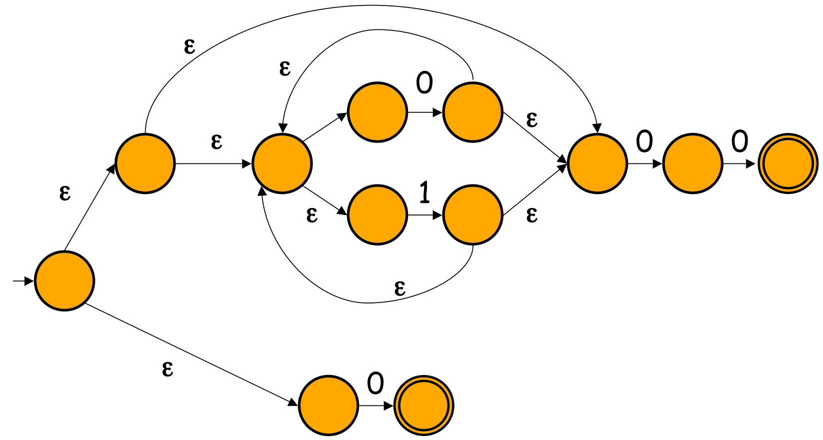


(0|1)*

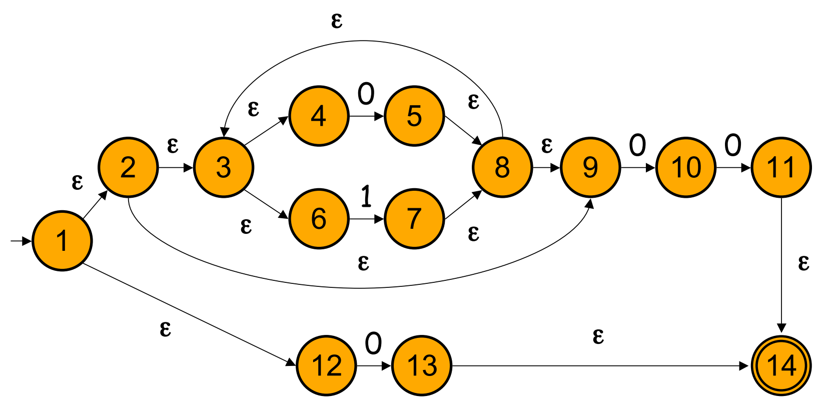




(0|1)*00



((0|1)*00)|0



((0|1)*00)|0

NFA to DFA Conversion

- Subset construction
- Idea: subsets of set of all NFA states are *equivalent* and become one DFA state
- Algorithm simulates movement through NFA
- Key problem: how to treat ε-transitions?

ε-Closure

- Start state: q_0
- ε-closure(S): S is a set of states

$$
\begin{aligned}
&\textbf{initialize: } S \leftarrow \{q_0\} \\
&T \leftarrow S \\
&\textbf{repeat } T' \leftarrow T \\
&\quad T \leftarrow T' \cup [\cup_{s \in T'} \textbf{move}(s, \epsilon)] \\
&\textbf{until } T = T'
\end{aligned}
$$

NFA Simulation

- After computing the ε-*closure* move, we get a set of states
- On some input extend all these states to get a new set of states

$$\textbf{DFAedge}(T, c) = \epsilon\textbf{-closure}\left(\cup_{q \in T} \textbf{move}(q, c)\right)$$

NFA Simulation

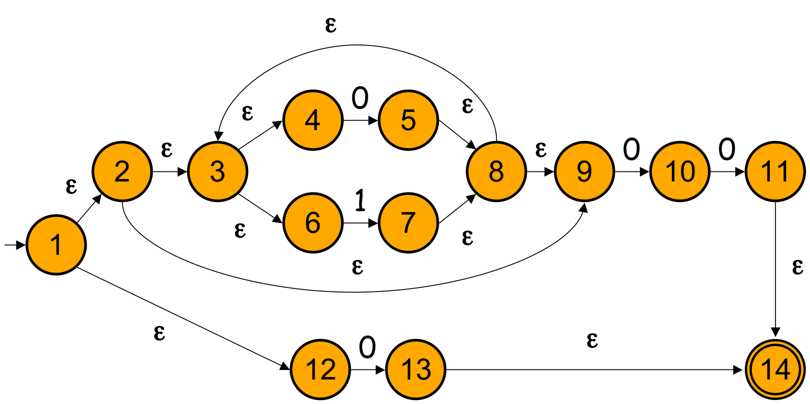
- Start state: q_0
- Input: $c_1, \ldots, c_k$

$$
\begin{aligned}
&T \leftarrow \epsilon\textbf{-closure}(\{q_0\}) \\
&\textbf{for } i \leftarrow 1 \textbf{ to } k \\
&\quad T \leftarrow \textbf{DFAedge}(T, c_i)
\end{aligned}
$$

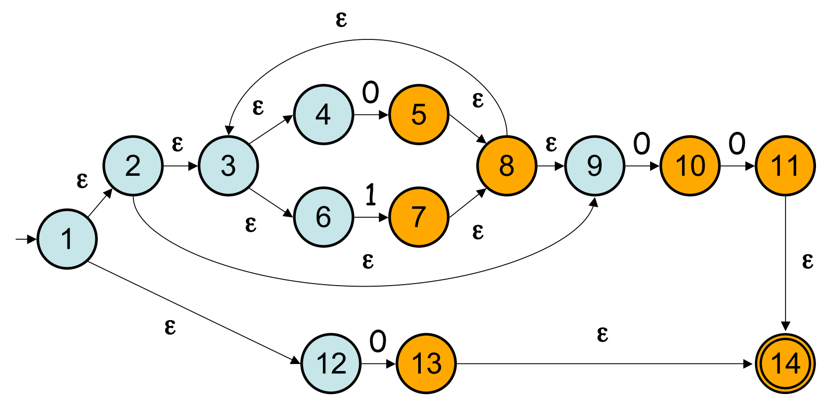
Conversion from NFA to DFA

- Conversion method closely follows the NFA simulation algorithm
- Instead of simulating, we can collect those NFA states that behave identically on the same input
- Group this set of states to form one state in the DFA

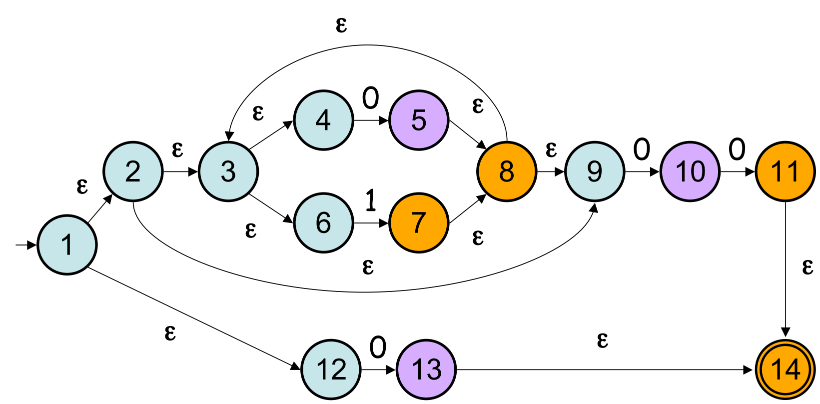
Example: subset construction



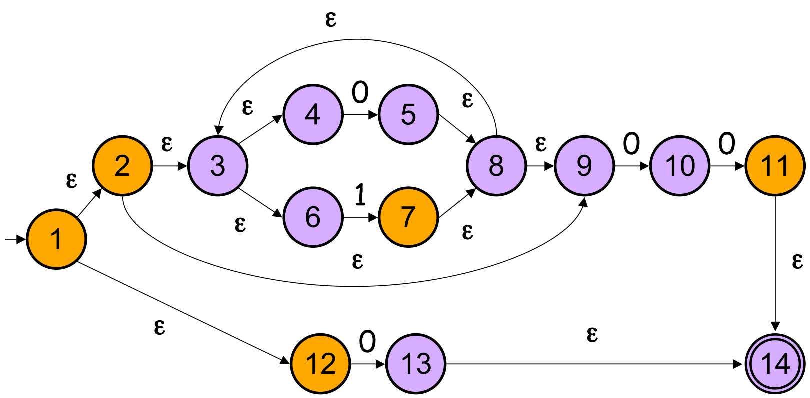
ε-*closure*(q_0)



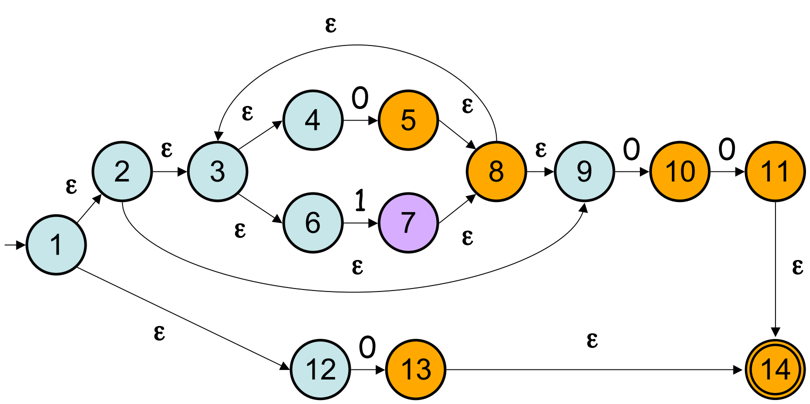
move(ε-*closure*(q_0), 0)



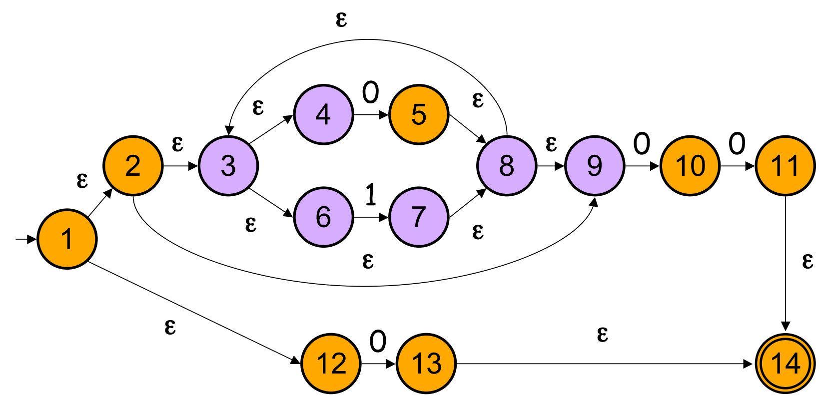
ε-*closure*(move(ε-*closure*(q_0), 0))



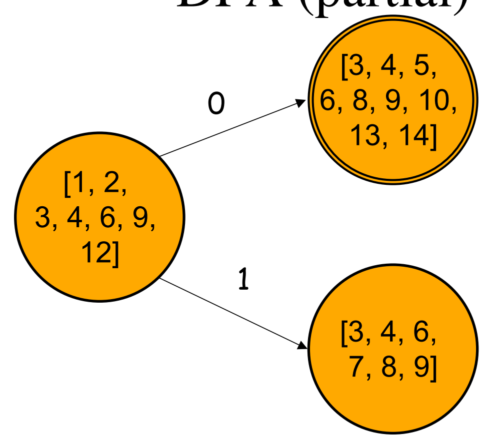
move(ε-*closure*(q_0), 1)



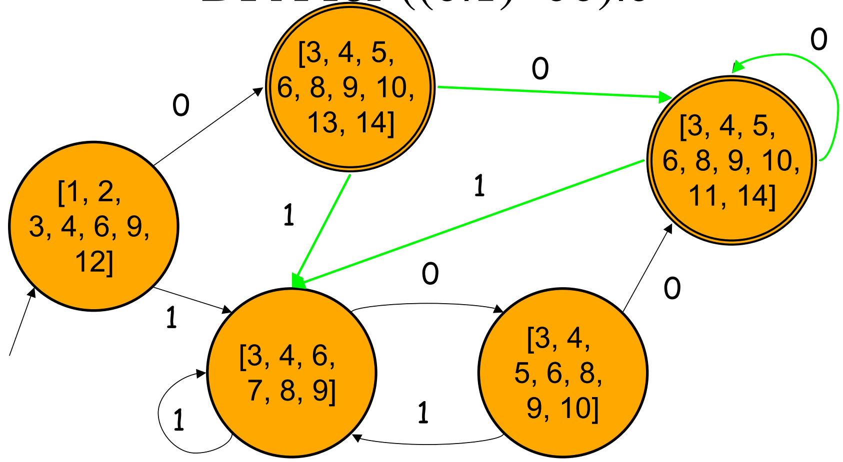
ε-*closure*(move(ε-*closure*(q_0), 1))



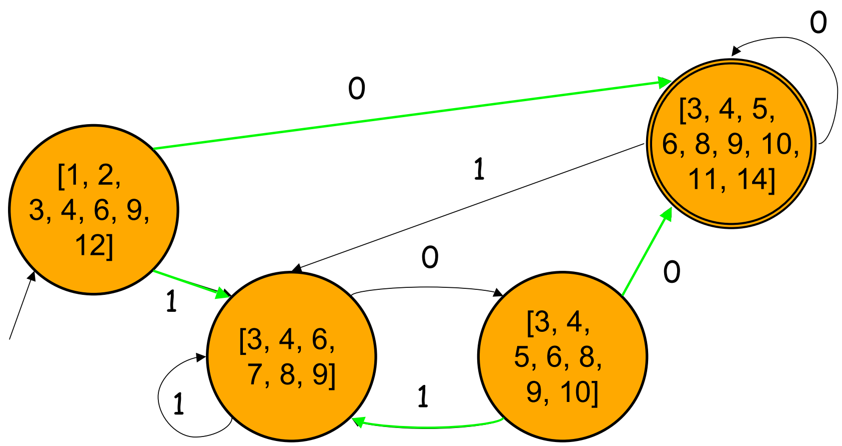
DFA (partial)



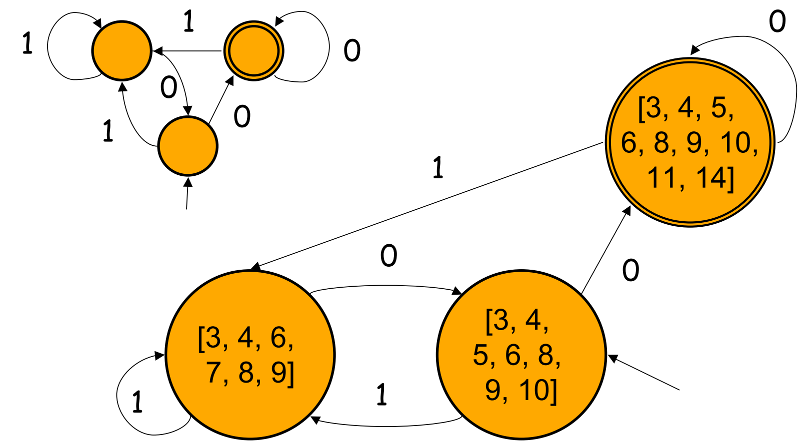
DFA for ((0|1)*00)|0



Minimization (I)

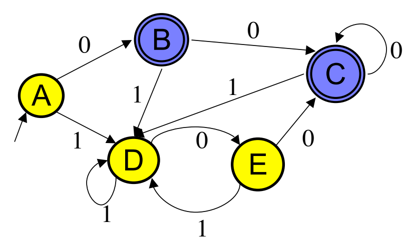


Minimization (II)



Minimization of DFAs

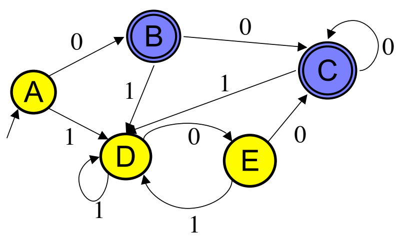
- Algorithm for minimizing the number of states in a DFA
- Step 1: partition states into 2 groups: accepting and non-accepting



Minimization of DFAs

- Step 2: in each group, find a sub-group of states having property P
- P: The states have transitions on each symbol (in the alphabet) to the *same* group

A, 0: blue
A, 1: yellow
E, 0: blue
E, 1: yellow
D, 0: yellow
D, 1: yellow

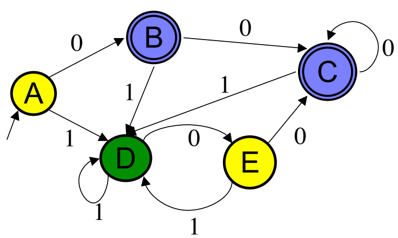


B, 0: blue
B, 1: yellow
C, 0: blue
C, 1: yellow

Minimization of DFAs

- Step 3: if a sub-group does not obey P split up the group into a separate group
- Go back to step 2. If no further sub-groups emerge then continue to step 4

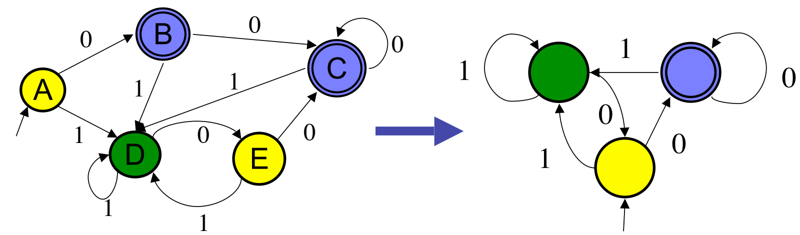
A, 0: blue
A, 1: green
E, 0: blue
E, 1: green
D, 0: yellow
D, 1: green



B, 0: blue
B, 1: green
C, 0: blue
C, 1: green

Minimization of DFAs

- Step 4: each group becomes a state in the minimized DFA
- Transitions to individual states are mapped to a single state representing the group of states



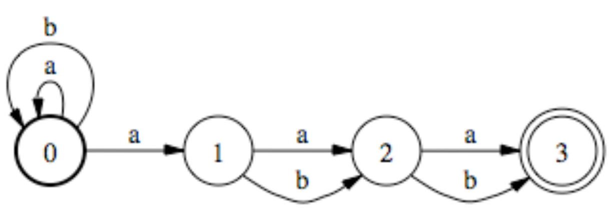
NFA to DFA

- Subset construction converts NFA to DFA
- Complexity:
  - in programs we measure time complexity in number of steps
  - For FSAs, we measure complexity in terms of the number of states

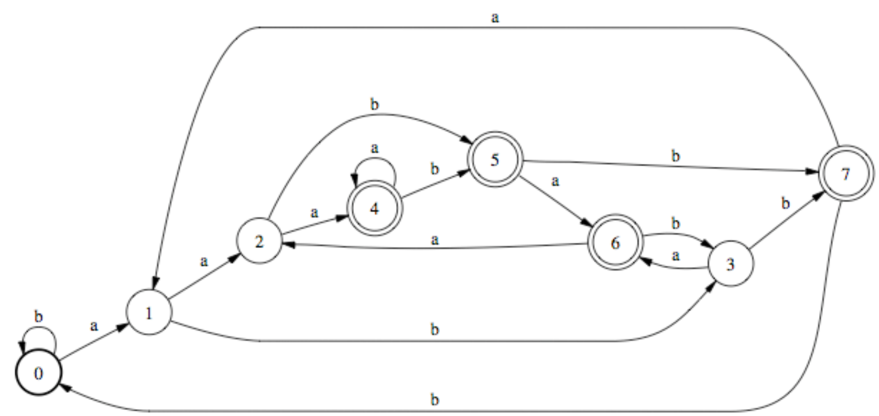
NFA to DFA

- Problem: An n state NFA can sometimes become a 2^n state DFA, an exponential increase in complexity
  - Try the subset construction on NFA built for the regexp $\mathbf{A^*aA^{n-1}}$ where **A** is the regexp **(a|b)**
- Minimization can reduce the number of states
- But minimization requires determinization

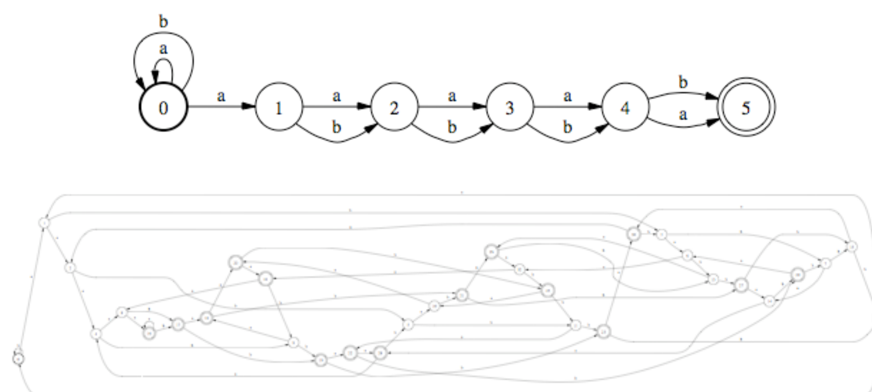
NFA to DFA



NFA to DFA



NFA to DFA



$2^5 = 32$ states

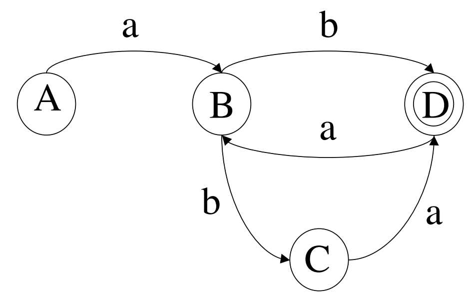
Equivalence of Regexps

- (R|S)|T == R|(S|T) == R|S|T
- (RS)T == R(ST)
- (R|S) == (S|R)
- R*R* == (R*)* == R* == RR*| ε
- R** == R*
- (R|S)T = RT|ST
- R(S|T) == RS | RT
- (R|S)* == (R*S*)* == (R*S)*R* == (R*|S*)*
- RR* == R*R
- (RS)*R == R(SR)*
- R = R|R = Rε = εR

Equivalence of Regexps

- 0(10)*1|(01)*
- (01)(01)*|(01)*
- (01)(01)*|(01)(01)*|ε
- (01)(01)*|ε
- (01)*
- (RS)*R == R(SR)*
- RS == (RS)
- R* == RR*|ε
- R == R|R
- R* == RR*| ε

NFA to RegExp



- A = a B
- B = b D | b C
- D = a B | ε
- C = a D

NFA to RegExp

- Three steps in the algorithm (apply in any order):

1. Substitution: for B = X pick every A = B | T and replace to get A = X | T
2. Factoring: (R S) | (R T) = R (S ∪ T) and (R T) | (S T) = (R ∪ S) T
3. Arden's Rule: For any set of strings S and T, the equation X = (S X) | T has X = (S*) T as a solution.

NFA to RegExp

- A = a B
  B = b D | b C
  D = a B | ε
  C = a D
- Substitute:
  A = a B
  B = b D | b a D
  D = a B | ε
- Factor:
  A = a B
  B = (b ∪ b a) D
  D = a B | ε
- Substitute:
  A = a (b ∪ b a) D
  D = a (b ∪ b a) D | ε

Summary

- Recognition of a string in a regular language: is a string accepted by an NFA?
- Conversion of regular expressions to NFAs
- Determinization: converting NFA to DFA
- Converting an NFA into a regular expression
- Other useful *closure* properties: union, concatenation, Kleene closure, intersection

NFA to RegExp

A = a (b ∪ b a) D
D = a (b ∪ b a) D | ε

- Factor:
  A = (a b ∪ a b a) D
  D = (a b ∪ a b a) D | ε
- Arden:
  A = (a b ∪ a b a) D
  D = (a b ∪ a b a)* ε
- Remove epsilon:
  A = (a b ∪ a b a) D
  D = (a b ∪ a b a)*
- Substitute:
  A = (a b ∪ a b a)
  (a b ∪ a b a)*
- Simplify:
  A = (a b ∪ a b a)+