CMSC 330: Organization of Programming Languages

DFAs, and NFAs, and Regexps
The story so far, and what’s next

- Goal: Develop an algorithm that determines whether a string \( s \) is matched by regex \( R \)
  - I.e., whether \( s \) is a member of \( R \)'s language

- Approach: Convert \( R \) to a finite automaton \( FA \) and see whether \( s \) is accepted by \( FA \)
  - Details: Convert \( R \) to a nondeterministic FA (NFA), which we then convert to a deterministic FA (DFA), which enjoys a fast acceptance algorithm
Two Types of Finite Automata

- **Deterministic** Finite Automata (DFA)
  - Exactly one sequence of steps for each string
    - Easy to implement acceptance check
  - All examples so far

- **Nondeterministic** Finite Automata (NFA)
  - May have many sequences of steps for each string
  - Accepts if any path ends in final state at end of string
  - More compact than DFA
    - But more expensive to test whether a string matches
Comparing DFAs and NFAs

- NFAs can have more than one transition leaving a state on the same symbol

- DFAs allow only one transition per symbol
  - I.e., transition function must be a valid function
  - DFA is a special case of NFA
Comparing DFAs and NFAs (cont.)

- NFAs may have transitions with empty string label
  - May move to new state without consuming character

- DFA transition must be labeled with symbol
  - DFA is a special case of NFA
DFA for $(a|b)^*abb$
NFA for \((a|b)^*abb\)

- ba
  - Has paths to either S0 or S1
  - Neither is final, so rejected

- babaabb
  - Has paths to different states
  - One path leads to S3, so accepts string
NFA for \((ab|aba)^*\)

- **aba**
  - Has paths to states \(S0, S1\)

- **ababa**
  - Has paths to \(S0, S1\)
  - Need to use \(\varepsilon\)-transition
Comparing NFA and DFA for \((ab|aba)^*\)
Quiz 1: Which DFA matches this regexp?

\[ b (b | a+b?) \]

A. 

B. 

C. 

D. None of the above
Quiz 1: Which DFA matches this regexp?

\[ b (b | a+b?) \]

A.

B.

C.

D. None of the above
A deterministic finite automaton (DFA) is a 5-tuple $(\Sigma, Q, q_0, F, \delta)$ where

- $\Sigma$ is an alphabet
- $Q$ is a nonempty set of states
- $q_0 \in Q$ is the start state
- $F \subseteq Q$ is the set of final states
- $\delta : Q \times \Sigma \rightarrow Q$ specifies the DFA's transitions

What's this definition saying that $\delta$ is?

A DFA accepts $s$ if it stops at a final state on $s$
Formal Definition: Example

- $\Sigma = \{0, 1\}$
- $Q = \{S0, S1\}$
- $q_0 = S0$
- $F = \{S1\}$

<table>
<thead>
<tr>
<th>$\delta$</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>S0</td>
<td>S1</td>
</tr>
<tr>
<td>S1</td>
<td>S0</td>
<td>S1</td>
</tr>
</tbody>
</table>

or as $\{(S0,0,S0),(S0,1,S1),(S1,0,S0),(S1,1,S1)\}$
Implementing DFAs (one-off)

It's easy to build a program which mimics a DFA

cur_state = 0;
while (1) {
    symbol = getchar();
    switch (cur_state) {
        case 0:
            switch (symbol) {
                case '0': cur_state = 0; break;
                case '1': cur_state = 1; break;
                case '\n': printf("rejected\n"); return 0;
                default: printf("rejected\n"); return 0;
            }
            break;
        case 1:
            switch (symbol) {
                case '0': cur_state = 0; break;
                case '1': cur_state = 1; break;
                case '\n': printf("accepted\n"); return 1;
                default: printf("rejected\n"); return 0;
            }
            break;
        default: printf("unknown state; I'm confused\n"); break;
    }
}

It's easy to build a program which mimics a DFA.
Implementing DFAs (generic)

More generally, use generic table-driven DFA

given components \((\Sigma, Q, q_0, F, \delta)\) of a DFA:

\[
\begin{align*}
\text{let } q &= q_0 \\
\text{while (there exists another symbol } \sigma \text{ of the input string)} &
\begin{align*}
q &:= \delta(q, \sigma) \\
\text{if } q \in F \text{ then } &\quad \text{accept} \\
\text{else } &\quad \text{reject}
\end{align*}
\end{align*}
\]

• \(q\) is just an integer
• Represent \(\delta\) using arrays or hash tables
• Represent \(F\) as a set
An NFA is a 5-tuple \((\Sigma, Q, q_0, F, \delta)\) where

- \(\Sigma, Q, q_0, F\) as with DFAs
- \(\delta \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times Q\) specifies the NFA's transitions

Example

- \(\Sigma = \{a\}\)
- \(Q = \{S1, S2, S3\}\)
- \(q_0 = S1\)
- \(F = \{S3\}\)
- \(\delta = \{(S1,a,S1), (S1,a,S2), (S2,\epsilon,S3)\}\)

An NFA accepts \(s\) if there is at least one path via \(s\) from the NFA's start state to a final state.
NFA Acceptance Algorithm (Sketch)

- When NFA processes a string $s$
  - NFA must keep track of several “current states”
    - Due to multiple transitions with same label, and $\varepsilon$-transitions
  - If any current state is final when done then accept $s$

- Example
  - After processing “a”
    - NFA may be in states $S1$, $S2$, $S3$
    - Since $S3$ is final, $s$ is accepted

- Algorithm is slow, space-inefficient; prefer DFAs!
Relating REs to DFAs and NFAs

- Regular expressions, NFAs, and DFAs accept the same languages! *Can convert between them*

NB. Both *transform* and *reduce* are historical terms; they mean “convert”
Goal: Given regular expression $A$, construct NFA: $<A> = (\Sigma, Q, q_0, F, \delta)$

- Remember regular expressions are defined recursively from primitive RE languages
- Invariant: $|F| = 1$ in our NFAs
  - Recall $F = \text{set of final states}$

Will define $<A>$ for base cases: $\sigma, \varepsilon, \emptyset$
- Where $\sigma$ is a symbol in $\Sigma$

And for inductive cases: $AB, A|B, A^*$
Reducing Regular Expressions to NFAs

Base case: $\sigma$

$<\sigma> = (\{\sigma\}, \{S0, S1\}, S0, \{S1\}, \{(S0, \sigma, S1)\})$
Reduction

- Base case: $\varepsilon$

$\langle \varepsilon \rangle = (\emptyset, \{S0\}, S0, \{S0\}, \emptyset)$

- Base case: $\emptyset$

$\langle \emptyset \rangle = (\emptyset, \{S0, S1\}, S0, \{S1\}, \emptyset)$
Reduction: Concatenation

Induction: $AB$

- $<A> = (\Sigma_A, Q_A, q_A, \{f_A\}, \delta_A)$
- $<B> = (\Sigma_B, Q_B, q_B, \{f_B\}, \delta_B)$
Reduction: Concatenation

Induction: \( AB \)

- \(<A> = (\Sigma_A, Q_A, q_A, \{f_A\}, \delta_A)\)
- \(<B> = (\Sigma_B, Q_B, q_B, \{f_B\}, \delta_B)\)
- \(<AB> = (\Sigma_A \cup \Sigma_B, Q_A \cup Q_B, q_A, \{f_B\}, \delta_A \cup \delta_B \cup \{(f_A, \epsilon, q_B)\})\)
Reduction: Union

Induction: $A|B$

- $<A> = (\Sigma_A, Q_A, q_A, \{f_A\}, \delta_A)$
- $<B> = (\Sigma_B, Q_B, q_B, \{f_B\}, \delta_B)$
Reduction: Union

Induction: $A|B$

- $<A> = (\Sigma_A, Q_A, q_A, \{f_A\}, \delta_A)$
- $<B> = (\Sigma_B, Q_B, q_B, \{f_B\}, \delta_B)$
- $<A|B> = (\Sigma_A \cup \Sigma_B, Q_A \cup Q_B \cup \{S0, S1\}, S0, \{S1\}, \delta_A \cup \delta_B \cup \{(S0, \varepsilon, q_A), (S0, \varepsilon, q_B), (f_A, \varepsilon, S1), (f_B, \varepsilon, S1)\})$
Reduction: Closure

Induction: $A^*$

• $<A> = (\Sigma_A, Q_A, q_A, \{f_A\}, \delta_A)$
Reduction: Closure

- Induction: $A^*$

- $<A> = (\Sigma_A, Q_A, q_A, \{f_A\}, \delta_A)$
- $<A^*> = (\Sigma_A, Q_A \cup \{S0,S1\}, S0, \{S1\},$
  $\delta_A \cup \{(f_A,\varepsilon,S1), (S0,\varepsilon,q_A), (S0,\varepsilon,S1), (S1,\varepsilon,S0)\})$
Quiz 2: Which NFA matches $a^*$?

A. 

B. 

C. 

D. 

CMSC 330 Spring 2020
Quiz 2: Which NFA matches $a^*$?

A.

B.

C.

D.
Quiz 3: Which NFA matches $a|b^*$?
Quiz 3: Which NFA matches $a|b^*$?
RE $\rightarrow$ NFA

Draw NFAs for the regular expression $(0|1)^*110^*$
Reduction Complexity

- Given a regular expression \( A \) of size \( n \)... 
  \[ \text{Size} = \# \text{ of symbols} + \# \text{ of operations} \]

- How many states does \(<A>\) have?
  - Two added for each |, two added for each *
  - \( O(n) \)
  - That’s pretty good!
Reducing NFA to DFA

can reduce

DFA ← NFA

can reduce

RE
Reducing NFA to DFA

- NFA may be reduced to DFA
  - By explicitly tracking the set of NFA states

- Intuition
  - Build DFA where
    - Each DFA state represents a set of NFA “current states”

- Example
Algorithm for Reducing NFA to DFA

- Reduction applied using the subset algorithm
  - DFA state is a subset of set of all NFA states

- Algorithm
  - Input
    - NFA ($\Sigma$, Q, $q_0$, $F_n$, $\delta$)
  - Output
    - DFA ($\Sigma$, R, $r_0$, $F_d$, $\delta$)
  - Using two subroutines
    - $\varepsilon$-closure($\delta$, p) (and $\varepsilon$-closure($\delta$, Q))
    - move($\delta$, p, $\sigma$) (and move($\delta$, Q, $\sigma$))
      - (where p is an NFA state)
\( \varepsilon \)-transitions and \( \varepsilon \)-closure

- We say \( p \xrightarrow{\varepsilon} q \)
  - If it is possible to go from state \( p \) to state \( q \) by taking only \( \varepsilon \)-transitions in \( \delta \)
  - If \( \exists p, p_1, p_2, \ldots p_n, q \in Q \) such that
    - \( \{p, \varepsilon, p_1\} \in \delta \), \( \{p_1, \varepsilon, p_2\} \in \delta \), \( \ldots \), \( \{p_n, \varepsilon, q\} \in \delta \)

- \( \varepsilon \)-closure(\( \delta \), \( p \))
  - Set of states reachable from \( p \) using \( \varepsilon \)-transitions alone
    - Set of states \( q \) such that \( p \xrightarrow{\varepsilon} q \) according to \( \delta \)
    - \( \varepsilon \)-closure(\( \delta \), \( p \)) = \{ \( q \mid p \xrightarrow{\varepsilon} q \) in \( \delta \) \}
    - \( \varepsilon \)-closure(\( \delta \), \( Q \)) = \{ \( q \mid p \in Q, p \xrightarrow{\varepsilon} q \) in \( \delta \) \}
  - Notes
    - \( \varepsilon \)-closure(\( \delta \), \( p \)) always includes \( p \)
    - We write \( \varepsilon \)-closure(\( p \)) or \( \varepsilon \)-closure(\( Q \)) when \( \delta \) is clear from context
ε-closure: Example 1

- Following NFA contains
  1. \( p_1 \xrightarrow{\varepsilon} p_2 \)
  2. \( p_2 \xrightarrow{\varepsilon} p_3 \)
  3. \( p_1 \xrightarrow{\varepsilon} p_3 \)

  ➤ Since \( p_1 \xrightarrow{\varepsilon} p_2 \) and \( p_2 \xrightarrow{\varepsilon} p_3 \)

- ε-closures
  1. \( \varepsilon\text{-closure}(p_1) = \{ p_1, p_2, p_3 \} \)
  2. \( \varepsilon\text{-closure}(p_2) = \{ p_2, p_3 \} \)
  3. \( \varepsilon\text{-closure}(p_3) = \{ p_3 \} \)
  4. \( \varepsilon\text{-closure}( \{ p_1, p_2 \} ) = \{ p_1, p_2, p_3 \} \cup \{ p_2, p_3 \} \)
**ε-closure: Example 2**

- Following NFA contains
  - p1 → p3
  - p3 → p2
  - p1 → p2
  - Since p1 → p3 and p3 → p2

- ε-closures
  - ε-closure(p1) = \{ p1, p2, p3 \}
  - ε-closure(p2) = \{ p2 \}
  - ε-closure(p3) = \{ p2, p3 \}
  - ε-closure( { p2, p3 } ) = \{ p2 \} \cup \{ p2, p3 \}
ε-closure Algorithm: Approach

- **Input:** NFA \((\Sigma, Q, q_0, F_n, \delta)\), State Set \(R\)
- **Output:** State Set \(R'\)
- **Algorithm**
  
  Let \(R' = R\)  
  
  Repeat
  
  Let \(R = R'\)  
  
  Let \(R' = R \cup \{q \mid p \in R, (p, \varepsilon, q) \in \delta\}\)  
  
  Until \(R = R'\)

This algorithm computes a **fixed point**
ε-closure Algorithm Example

Calculate $\varepsilon$-closure($\delta, \{p1\}$)

<table>
<thead>
<tr>
<th>R</th>
<th>R'</th>
</tr>
</thead>
<tbody>
<tr>
<td>${p1}$</td>
<td>${p1}$</td>
</tr>
<tr>
<td>${p1}$</td>
<td>${p1, p2}$</td>
</tr>
<tr>
<td>${p1, p2}$</td>
<td>${p1, p2, p3}$</td>
</tr>
<tr>
<td>${p1, p2, p3}$</td>
<td>${p1, p2, p3}$</td>
</tr>
</tbody>
</table>

Let $R' = R$
Repeat
  Let $R = R'$
  Let $R' = R \cup \{q \mid p \in R, (p, \varepsilon, q) \in \delta\}$
Until $R = R'$
Calculating $\text{move}(p, \sigma)$

- $\text{move}(\delta, p, \sigma)$
  
  - Set of states reachable from $p$ using exactly one transition on symbol $\sigma$
    
    - Set of states $q$ such that $\{p, \sigma, q\} \in \delta$
    
    - $\text{move}(\delta, p, \sigma) = \{ q \mid \{p, \sigma, q\} \in \delta \}$
    
    - $\text{move}(\delta, Q, \sigma) = \{ q \mid p \in Q, \{p, \sigma, q\} \in \delta \}$
      
      - i.e., can “lift” $\text{move}()$ to a set of states $Q$

- Notes:
  
  - $\text{move}(\delta, p, \sigma)$ is $\emptyset$ if no transition $(p, \sigma, q) \in \delta$, for any $q$
  
  - We write $\text{move}(p, \sigma)$ or $\text{move}(R, \sigma)$ when $\delta$ clear from context
move(p, σ) : Example 1

- Following NFA
  - \( \Sigma = \{ a, b \} \)

- Move
  - \( \text{move}(p_1, a) = \{ p_2, p_3 \} \)
  - \( \text{move}(p_1, b) = \emptyset \)
  - \( \text{move}(p_2, a) = \emptyset \)
  - \( \text{move}(p_2, b) = \{ p_3 \} \)
  - \( \text{move}(p_3, a) = \emptyset \)
  - \( \text{move}(p_3, b) = \emptyset \)

\( \text{move}(\{p_1, p_2\}, b) = \{ p_3 \} \)
move(p, σ) : Example 2

- Following NFA
  - $\Sigma = \{ a, b \}$

- Move
  - $\text{move}(p_1, a) = \{ p_2 \}$
  - $\text{move}(p_1, b) = \{ p_3 \}$
  - $\text{move}(p_2, a) = \{ p_3 \}$
  - $\text{move}(p_2, b) = \emptyset$
  - $\text{move}(p_3, a) = \emptyset$
  - $\text{move}(p_3, b) = \emptyset$
  - $\text{move}(\{p_1, p_2\}, a) = \{p_2, p_3\}$
NFA → DFA Reduction Algorithm ("subset")

- Input NFA (Σ, Q, q₀, Fₙ, δ), Output DFA (Σ, R, r₀, Fₚ, δ’)
- Algorithm

  Let r₀ = ε-closure(δ, q₀), add it to R // DFA start state

  While ∃ an unmarked state r ∈ R // process DFA state r
    Mark r // each state visited once
    For each σ ∈ Σ // for each symbol σ
      Let E = move(δ, r, σ) // states reached via σ
      Let e = ε-closure(δ, E) // states reached via ε
      If e ∉ R // if state e is new
        Let R = R ∪ {e} // add e to R (unmarked)
        Let δ’ = δ’ ∪ {r, σ, e} // add transition r→e on σ
    Let Fₚ = {r | ∃ s ∈ r with s ∈ Fₙ} // final if include state in Fₙ
NFA → DFA Example 1

• Start = $\varepsilon$-closure($\delta$,p1) = \{ p1,p3 \}
• R = \{ p1,p3 \}
• r $\in$ R = \{ p1,p3 \}
• move($\delta$,{p1,p3},a) = \{ p2 \}
  ➢ e = $\varepsilon$-closure($\delta$,\{p2\}) = \{ p2 \}
  ➢ R = R $\cup$ \{p2\} = \{ p1,p3, \{ p2 \} \}
  ➢ $\delta'$ = $\delta'$ $\cup$ \{p1,p3, a, \{ p2 \} \}
• move(\delta,\{p1,p3\},b) = \emptyset

![Diagram of NFA and DFA](image)
NFA → DFA Example 1 (cont.)

- $R = \{ \{p1,p3\}, \{p2\} \}$
- $r \in R = \{p2\}$
- $\text{move}(\delta, \{p2\}, a) = \emptyset$
- $\text{move}(\delta, \{p2\}, b) = \{p3\}$
  - $e = \varepsilon$-closure$(\delta, \{p3\}) = \{p3\}$
  - $R = R \cup \{\{p3\}\} = \{\{p1,p3\}, \{p2\}, \{p3\}\}$
  - $\delta' = \delta' \cup \{\{p2\}, b, \{p3\}\}$

![NFA Diagram]

![DFA Diagram]
NFA → DFA Example 1 (cont.)

• \( R = \{ \{p1,p3\}, \{p2\}, \{p3\} \} \)
• \( r \in R = \{p3\} \)
• \( \text{Move(\{p3\},a)} = \emptyset \)
• \( \text{Move(\{p3\},b)} = \emptyset \)
• \( \text{Mark \{p3\}, exit loop} \)
• \( F_d = \{\{p1,p3\}, \{p3\}\} \)
  ➢ Since \( p3 \in F_n \)
• Done!
NFA $\rightarrow$ DFA Example 2

- **NFA**

- **DFA**

$$ R = \{ \{A\}, \{B,D\}, \{C,D\} \} $$
Quiz 4: Which DFA is equivalent to this NFA?

NFA:

A. 

B. 

C. 

D. None of the above
Quiz 4: Which DFA is equivalent to this NFA?

NFA:

A.

B.

C.

D. None of the above
Actual Answer

NFA:
NFA → DFA Example 3

R = \{ {A, E}, {B, D, E}, {C, D}, {E} \}
NFA → DFA Example
NFA → DFA Practice
NFA → DFA Practice
Analyzing the Reduction

- Can reduce any NFA to a DFA using subset alg.
- How many states in the DFA?
  - Each DFA state is a subset of the set of NFA states
  - Given NFA with \( n \) states, DFA may have \( 2^n \) states
    - Since a set with \( n \) items may have \( 2^n \) subsets
  - Corollary
    - Reducing a NFA with \( n \) states may be \( O(2^n) \)
Recap: Matching a Regexp $R$

- Given $R$, construct NFA. Takes time $O(R)$
- Convert NFA to DFA. Takes time $O(2^{|R|})$
  - But usually not the worst case in practice
- Use DFA to accept/reject string $s$
  - Assume we can compute $\delta(q,\sigma)$ in constant time
  - Then time to process $s$ is $O(|s|)$
    - Can’t get much faster!
- Constructing the DFA is a one-time cost
  - But then processing strings is fast
Closing the Loop: Reducing DFA to RE

DFA

NFA

can transform

can reduce

can transform

RE
Reducing DFAs to REs

- **General idea**
  - Remove states one by one, labeling transitions with regular expressions
  - When two states are left (start and final), the transition label is the regular expression for the DFA
Minimizing DFAs

- Every regular language is recognizable by a unique minimum-state DFA
  - Ignoring the particular names of states

- In other words
  - For every DFA, there is a unique DFA with minimum number of states that accepts the same language
Minimizing DFA: Hopcroft Reduction

Intuition

- Look to distinguish states from each other
  - End up in different accept / non-accept state with identical input

Algorithm

- Construct initial partition
  - Accepting & non-accepting states
- Iteratively split partitions (until partitions remain fixed)
  - Split a partition if members in partition have transitions to different partitions for same input
    - Two states \( x, y \) belong in same partition if and only if for all symbols in \( \Sigma \) they transition to the same partition
- Update transitions & remove dead states
No need to split partition \{S,T,U,V\}

- All transitions on a lead to identical partition P2
- Even though transitions on a lead to different states
Splitting Partitions (cont.)

- Need to split partition \{S,T,U\} into \{S,T\}, \{U\}
  - Transitions on \(a\) from \(S,T\) lead to partition \(P_2\)
  - Transition on \(a\) from \(U\) lead to partition \(P_3\)
Resplitting Partitions

- Need to reexamine partitions after splits
  - Initially no need to split partition \{S,T,U\}
  - After splitting partition \{X,Y\} into \{X\}, \{Y\} we need to split partition \{S,T,U\} into \{S,T\}, \{U\}
Minimizing DFA: Example 1

- DFA

- Initial partitions

- Split partition
Minimizing DFA: Example 1

DFA

Initial partitions
• Accept \{ R \} = P1
• Reject \{ S, T \} = P2

Split partition? → Not required, minimization done
• move(S,a) = T ∈ P2
• move(T,a) = T ∈ P2
• move(S,b) = R ∈ P1
• move(T,b) = R ∈ P1
Minimizing DFA: Example 2
Minimizing DFA: Example 2

- **DFA**

- **Initial partitions**
  - Accept  \{ R \} = P1
  - Reject \{ S, T \} = P2

- **Split partition? → Yes, different partitions for B**
  - move(S,a) = T ∈ P2  –  move(S,b) = T ∈ P2
  - move(T,a) = T ∈ P2  –  move(T,b) = R ∈ P1

DFA already minimal
Complement of DFA

Given a DFA accepting language \( L \)

- How can we create a DFA accepting its complement?
- Example DFA
  - \( \Sigma = \{a, b\} \)
Complement of DFA

- Algorithm
  - Add explicit transitions to a dead state
  - Change every accepting state to a non-accepting state & every non-accepting state to an accepting state

- Note this only works with DFAs
  - Why not with NFAs?
Summary of Regular Expression Theory

- Finite automata
  - DFA, NFA

- Equivalence of RE, NFA, DFA
  - RE → NFA
    - Concatenation, union, closure
  - NFA → DFA
    - $\varepsilon$-closure & subset algorithm

- DFA
  - Minimization, complementation