CMSC 330: Organization of Programming Languages

DFAs, and NFAs, and Regexps
(Oh my!)
Types of Finite Automata

- Deterministic Finite Automata (DFA)
  - Exactly one sequence of steps for each string
  - 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?

$ba^+ (a | b)$

A.

B.

C.

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

\[ ba^+ (a \mid b) \]

A.

B.

C.

D. None of the above
How NFA Acceptance Works

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

- Example
  - After processing “a”
    - NFA may be in states
      - S1
      - S2
      - S3

Diagram: NFA states S1, S2, S3 with transitions on 'a' and $\epsilon$.
Relating REs to DFAs and NFAs

- Regular expressions, NFAs, and DFAs accept the same languages!

DFA  \longrightarrow  \text{can transform}  \longrightarrow  \text{NFA}

\text{RE}  \longrightarrow  \text{can transform}  \longrightarrow  \text{DFA}

\text{can transform}  \longrightarrow  \text{RE}  \longrightarrow  \text{can transform}
Formal Definition

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

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

or as
\[
\delta = \{(S0,0,S0),(S0,1,S1),(S1,0,S0),(S1,1,S1)\}
\]
Nondeterministic Finite Automata (NFA)

- 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

- An NFA accepts \(s\) if there is at least one path via \(s\) from the NFA’s start state to a final state
Reducing Regular Expressions to NFAs

- 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: $\varepsilon$**

  $<\varepsilon> = (\emptyset, \{S0\}, S0, \{S0\}, \emptyset)$

- **Base case: $\emptyset$**

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

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

- Induction: $A^*$

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

- Induction: $A^*$

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

A. 

B. 

C. 

D.
Quiz 3: Which NFA matches $a|b^*$?
Quiz 3: Which NFA matches \( a|b^* \) ?
Draw NFAs for the regular expression (0|1)*110*
Draw NFAs for the regular expression $(ab^*c|d^*a|ab)d$
Reduction Complexity

Given a regular expression $A$ of size $n$...
Size = # of symbols + # of operations

How many states does $<A>$ have?
• Two added for each $\mid$, two added for each $*$
• $O(n)$
• That’s pretty good!
Recap

- Finite automata
  - Alphabet, states…
  - $(\Sigma, Q, q_0, F, \delta)$

- Types
  - Deterministic (DFA)
  - Non-deterministic (NFA)

- Reducing RE to NFA
  - Concatenation
  - Union
  - Closure
Reducing NFA to DFA
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

```
NFA
S1  a  S2  ε  S3
S1, S2, S3
```

```
DFA
S1  a
S1, S2, S3
```
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 (Σ, Q, q₀, Fₙ, δ)
- Output
  - DFA (Σ, R, r₀, Fₜ, δ)
- Using two subroutines
  - ε-closure(p)
  - move(p, a)
We say $p \xrightarrow{\varepsilon} q$

- If it is possible to go from state $p$ to state $q$ by taking only $\varepsilon$-transitions
- 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($p$)

- Set of states reachable from $p$ using $\varepsilon$-transitions alone
  - Set of states $q$ such that $p \xrightarrow{\varepsilon} q$
  - $\varepsilon$-closure($p$) = $\{q \mid p \xrightarrow{\varepsilon} q\}$

Note

- $\varepsilon$-closure($p$) always includes $p$
- $\varepsilon$-closure($\ )$ may be applied to set of states (take union)
\( \varepsilon \)-closure: Example 1

- Following NFA contains
  - \( S_1 \xrightarrow{\varepsilon} S_2 \)
  - \( S_2 \xrightarrow{\varepsilon} S_3 \)
  - \( S_1 \xrightarrow{\varepsilon} S_3 \)
  
  - Since \( S_1 \xrightarrow{\varepsilon} S_2 \) and \( S_2 \xrightarrow{\varepsilon} S_3 \)

- \( \varepsilon \)-closures
  - \( \varepsilon \)-closure(\( S_1 \)) = \{ \text{S1, S2, S3} \}
  - \( \varepsilon \)-closure(\( S_2 \)) = \{ \text{S2, S3} \}
  - \( \varepsilon \)-closure(\( S_3 \)) = \{ \text{S3} \}
  - \( \varepsilon \)-closure( \{ \text{S1, S2} \} ) = \{ \text{S1, S2, S3} \} \cup \{ \text{S2, S3} \} \)
\( \varepsilon \)-closure: Example 2

Following NFA contains

- \( S_1 \xrightarrow{\varepsilon} S_3 \)
- \( S_3 \xrightarrow{\varepsilon} S_2 \)
- \( S_1 \xrightarrow{\varepsilon} S_2 \)

\( \Rightarrow \) Since \( S_1 \xrightarrow{\varepsilon} S_3 \) and \( S_3 \xrightarrow{\varepsilon} S_2 \)

\( \varepsilon \)-closures

- \( \varepsilon \)-closure(\( S_1 \)) = \{ \( S_1, S_2, S_3 \) \}
- \( \varepsilon \)-closure(\( S_2 \)) = \{ \( S_2 \) \}
- \( \varepsilon \)-closure(\( S_3 \)) = \{ \( S_2, S_3 \) \}
- \( \varepsilon \)-closure(\{ \( S_2, S_3 \) \}) = \{ \( S_2 \) \} \cup \{ \( S_2, S_3 \) \}
Calculating move(p,a)

- move(p,a)
  - Set of states reachable from p using exactly one transition on a
    - Set of states q such that {p, a, q} ∈ δ
    - move(p,a) = {q | {p, a, q} ∈ δ}
  - Note: move(p,a) may be empty ∅
    - If no transition from p with label a
move(a,p) : Example 1

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

Move
- $\text{move}(S1, a) = \{ S2, S3 \}$
- $\text{move}(S1, b) = \emptyset$
- $\text{move}(S2, a) = \emptyset$
- $\text{move}(S2, b) = \{ S3 \}$
- $\text{move}(S3, a) = \emptyset$
- $\text{move}(S3, b) = \emptyset$
move(a,p) : Example 2

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

Move
- $\text{move}(S1, a) = \{ S2 \}$
- $\text{move}(S1, b) = \{ S3 \}$
- $\text{move}(S2, a) = \{ S3 \}$
- $\text{move}(S2, b) = \emptyset$
- $\text{move}(S3, a) = \emptyset$
- $\text{move}(S3, b) = \emptyset$
NFA → DFA Reduction Algorithm (“subset”)

- **Input** NFA (Σ, Q, q₀, Fᵣ, δ), Output DFA (Σ, R, r₀, Fₛ, δ)
- **Algorithm**
  
  Let \( r₀ = \varepsilon\text{-closure}(q₀) \), add it to R  
  // DFA start state
  
  While \( \exists \) an unmarked state \( r \in R \)  
  // process DFA state \( r \)
  
  Mark \( r \)  
  // each state visited once
  
  For each \( a \in \Sigma \)  
  // for each letter \( a \)
    
    Let \( S = \{s \mid q \in r & \text{move}(q,a) = s\} \)  
    // states reached via \( a \)
    
    Let \( e = \varepsilon\text{-closure}(S) \)  
    // states reached via \( \varepsilon \)
    
    If \( e \notin R \)  
    // if state \( e \) is new
      
      Let \( R = R \cup \{e\} \)  
      // add \( e \) to \( R \) (unmarked)
      
      Let \( \delta = \delta \cup \{r, a, e\} \)  
      // add transition \( r \to e \)
    
    Let \( Fₛ = \{r \mid \exists s \in r \text{ with } s \in Fᵣ\} \)  
    // final if include state in \( Fᵣ \)
NFA → DFA Example 1

• Start = \( \varepsilon \)-closure(S1) = \{ S1,S3 \}
• \( R = \{ S1,S3 \} \)
• \( r \in R = \{ S1,S3 \} \)
• Move(S1,S3,a) = S2
  - \( e = \varepsilon \)-closure({S2}) = {S2}
  - \( R = R \cup \{ S2 \} = \{ S1,S3, S2 \} \)
  - \( \delta = \delta \cup \{ S1,S3, a, S2 \} \)
• Move(S1,S3,b) = \( \emptyset \)
NFA → DFA Example 1 (cont.)

- \( R = \{ \{S1,S3\}, \{S2\} \} \)
- \( r \in R = \{S2\} \)
- \( \text{Move}(\{S2\},a) = \emptyset \)
- \( \text{Move}(\{S2\},b) = \{S3\} \)
  - \( e = \varepsilon\)-closure(\(\{S3\}\)) = \{S3\}
  - \( R = R \cup \{S3\} = \{ \{S1,S3\}, \{S2\}, \{S3\} \} \)
  - \( \delta = \delta \cup \{S2\}, b, \{S3\} \)
NFA → DFA Example 1 (cont.)

- \( R = \{ \{S1,S3\}, \{S2\}, \{S3\} \} \)
- \( r \in R = \{S3\} \)
- \( \text{Move}(\{S3\},a) = \emptyset \)
- \( \text{Move}(\{S3\},b) = \emptyset \)
- Mark \( \{S3\} \), exit loop
- \( F_d = \{\{S1,S3\}, \{S3\}\} \)
  - Since \( S3 \in F_n \)
- Done!
NFA → DFA Example 2

NFA

\[ R = \{ \{A\}, \{B,D\}, \{C,D\} \} \]
Quiz 4: Which DFA is equiv 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
NFA:
NFA $\rightarrow$ DFA Example 3

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

- Any string from \{A\} to either \{D\} or \{CD\}
  - Represents a path from A to D in the original NFA

NFA

DFA
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 $O(2^n)$
Reducing DFA to RE

DFA ← can transform NFA

RE ← can transform DFA

RE ← can transform NFA
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
DFA to RE example

Language over $\Sigma = \{0,1\}$ such that every string is a multiple of 3 in binary
DFA to RE example

Language over $\Sigma = \{0,1\}$ such that every string is a multiple of 3 in binary

$$((0 + 1(0 1^* 0)1)^*)^*$$
Other Topics

- Minimizing DFA
  - Hopcroft reduction
- Complementing DFA
- Implementing 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
Splitting Partitions

- **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

![Diagram showing partitions P1 and P2 with transitions labeled a](image-url)
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(S,b) = R ∈ P1
  - move(T,a) = T ∈ P2  –  move(T,b) = R ∈ P1
Minimizing DFA: Example 2
Minimizing DFA: Example 2

- **DFA**

- **Initial partitions**
  - Accept  \( \{ R \} \)  = \( P_1 \)
  - Reject  \( \{ S, T \} \)  = \( P_2 \)

- **Split partition?** → Yes, different partitions for B
  - \( \text{move}(S,a) = T \in P_2 \)  – \( \text{move}(S,b) = T \in P_2 \)
  - \( \text{move}(T,a) = T \in P_2 \)  – \( \text{move}(T,b) = R \in P_1 \)

DFA already minimal
Minimizing DFA: Example 3
Minimizing DFA: Example 3
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?
Implementing DFAs (one-off)

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

```c
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:

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

- \(q\) is just an integer
- Represent \(\delta\) using arrays or hash tables
- Represent \(F\) as a set
Running Time of DFA

- How long for DFA to decide to accept/reject string $s$?
  - Assume we can compute $\delta(q, c)$ in constant time
  - Then time to process $s$ is $O(|s|)$
    - Can’t get much faster!
- Constructing DFA for RE $A$ may take $O(2^{|A|})$ time
  - But usually not the case in practice
- So there’s the initial overhead
  - But then processing strings is fast
Regular Expressions in Practice

- Regular expressions are typically “compiled” into tables for the generic algorithm
  - Can think of this as a simple byte code interpreter
  - But really just a representation of $(\Sigma, Q_A, q_A, \{f_A\}, \delta_A)$, the components of the DFA produced from the RE

- Regular expression implementations often have extra constructs that are non-regular
  - I.e., can accept more than the regular languages
  - Can be useful in certain cases
  - Disadvantages
    - Nonstandard, plus can have higher complexity
Summary of Regular Expression Theory

- Finite automata
  - DFA, NFA

- Equivalence of RE, NFA, DFA
  - \( RE \rightarrow NFA \)
    - Concatenation, union, closure
  - \( NFA \rightarrow DFA \)
    - \( \varepsilon \)-closure & subset algorithm

- DFA
  - Minimization, complement
  - Implementation