**CMSC 330: Organization of Programming Languages**

**Finite Automata 2**

**This Lecture**
- Reducing NFA to DFA
  - ε-closure
  - Subset algorithm
- Minimizing DFA
  - Hopcroft reduction
- Complementing DFA
- Implementing 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 states
- Example

**Last Lecture**
- Finite automata
  - Alphabet, states…
  - \([\Sigma, Q, q_0, F, \delta]\)
- Types
  - Deterministic (DFA)
  - Non-deterministic (NFA)

**How NFA Works**
- When NFA processes a string
  - NFA may be in several possible states
    - Multiple transitions with same label
    - ε-transitions
- Example
  - After processing "a"
    - NFA may be in states
      - S1
      - S2
      - S3

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

**Reducing NFA to DFA (cont.)**
- Reduction applied using the subset algorithm
  - DFA state is a subset of set of all NFA states
- Algorithm
  - Input
    - NFA \([\Sigma, Q, q_0, F, \delta]\)
  - Output
    - DFA \([\Sigma, R, r_0, F, \delta]\)
  - Using
    - ε-closure(p)
    - move(p, a)
ε-transitions and ε-closure

- 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
    \( (p, \varepsilon, p_1) = \delta, (p_1, \varepsilon, p_2) = \delta, \ldots, (p_n, \varepsilon, q) = \delta \)

- \( \varepsilon \)-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(\( p \)) may be applied to set of states (take union)

ε-closure: Example 1

- Following NFA contains
  - \( S_1 \), \( S_2 \)
  - \( S_2 \), \( S_3 \)
  - \( S_1 \), \( S_3 \)

- \( \varepsilon \)-closures
  - \( \varepsilon \)-closure(\( S_1 \)) = \( \{ S_1, S_2, S_3 \} \)
  - \( \varepsilon \)-closure(\( S_2 \)) = \( \{ S_2, S_3 \} \)
  - \( \varepsilon \)-closure(\( S_3 \)) = \( \{ S_3 \} \)
  - \( \varepsilon \)-closure(\( \{ S_1, S_2 \} \)) = \( \{ S_1, S_2, S_3 \} \cup \{ S_2, S_3 \} \)

ε-closure: Example 2

- Following NFA contains
  - \( S_1 \), \( S_3 \)
  - \( S_3 \), \( S_2 \)
  - \( S_1 \), \( S_2 \)

- \( \varepsilon \)-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 \( \text{move}(p,a) \)

- \( \text{move}(p,a) \)
  - Set of states reachable from \( p \) using exactly one transition on \( a \)
    - Set of states \( q \) such that \( p, a, q \) \( \in \delta \)
    - \( \text{move}(p,a) = \{ q \mid p, a, q \in \delta \} \)
  - Note \( \text{move}(p,a) \) may be empty \( \emptyset \)
    - If no transition from \( p \) with label \( a \)

ε-closure: Practice

- Find \( \varepsilon \)-closures for following NFA

- Find \( \varepsilon \)-closures for the NFA you construct for
  - The regular expression \( (0|1*)111(0*|1) \)

move(\( a,p \)) : Example 1

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

- Move
  - \( \text{move}(S_1, a) = \{ S_2, S_3 \} \)
  - \( \text{move}(S_1, b) = \emptyset \)
  - \( \text{move}(S_2, a) = \emptyset \)
  - \( \text{move}(S_2, b) = \{ S_3 \} \)
  - \( \text{move}(S_3, a) = \emptyset \)
  - \( \text{move}(S_3, b) = \emptyset \)
**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 \rightarrow DFA Reduction Algorithm**

- Input NFA $(\Sigma, Q, q_0, F_0, \delta)$, Output DFA $(\Sigma, R, r_0, F_d, \delta)$

- Algorithm
  - Let $r_0 = \epsilon$-closure$(q_0)$, add it to $R$ // DFA start state
  - While there is an unmarked state $r \in R$
    - Mark $r$ // process DFA state $r$
    - For each $a \in \Sigma$
      - Let $S = \{s \mid s \in R \& \text{move}(q, a) = s\}$ // states reached via $a$
      - Let $e = \epsilon$-closure($S$) // states reached via $\epsilon$
      - If $e \in R$ // if state $e$ is new
        - Let $R = e \cup R$ // add $e$ to $R$ (unmarked)
        - Let $\delta = \delta \cup \{r, a, e\}$ // add transition $r \rightarrow e$
      - Let $F_d = \{r \mid \exists s \in R \text{ with } s \in F_0\}$ // final if include state in $F_0$
  - For each $a$
    - Mark $r$
    - $\delta$ \cup $s$
      - $R = \epsilon$-closure($\{S_1, S_3\}, \{S_2\}$)
      - $\delta = \delta \cup \{\{S_2\}, \{S_3\}\}$
      - $\delta = \delta \cup \{\{S_2, b, \{S_3\}\}$

**NFA \rightarrow DFA Example 1**

- Start = $\epsilon$-closure($S_1$) = $\{S_1, S_3\}$
- $R = \{\{S_1, S_3\}\}$
- $r \in R = \{S_1, S_3\}$
- $\text{Move}((S_1, S_3), a) = \{S_2\}$
  - $e = \epsilon$-closure($S_2$)
  - $R = R \cup \{S_2\} = \{S_1, S_3\}$
  - $\delta = \delta \cup \{S_1, S_3, a, \{S_2\}\}$
- $\text{Move}((S_1, S_3), b) = \emptyset$

**NFA \rightarrow DFA Example 1 (cont.)**

- $R = \{\{S_1, S_3\}, \{S_2\}\}$
- $r \in R = \{S_2\}$
- $\text{Move}((S_2), a) = \emptyset$
- $\text{Move}((S_2), b) = \{S_3\}$
  - $e = \epsilon$-closure($S_3$)
  - $R = R \cup \{S_3\} = \{S_1, S_3\}$
  - $\delta = \delta \cup \{S_2, \{S_3\}\}$

**NFA \rightarrow DFA Example 1 (cont.)**

- $R = \{\{S_1, S_3\}, \{S_2\}, \{S_3\}\}$
- $r \in R = \{S_3\}$
- $\text{Move}((S_3), a) = \emptyset$
- $\text{Move}((S_3), b) = \emptyset$
- $F_d = \{\{S_1, S_3\}, \{S_3\}\}$
  - Since $S_3 \in F_n$
- Done!

**NFA \rightarrow DFA Example 2**

- NFA
- DFA

**R** = \{ [A], [B, D], [C, D] \}
NFA → DFA Example 3

- NFA
  - A, B, C, D, E
  - R = \{ {A,E}, {B,D,E} \}

- DFA
  - A, B, C, D
  - R = \{ {A,E}, {B,D,E} \}

Equivalence of DFAs and NFAs

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

Equivalence of DFAs and NFAs (cont.)

- 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 \(O(2^n)\)

Minimizing DFA

- Result from CS theory
  - Every regular language is recognizable by a minimum-state DFA that is unique up to state names
- In other words
  - For every DFA, there is a unique DFA with minimum number of states that accepts the same language
  - Two minimum-state DFAs have same underlying shape

Minimizing DFA: Hopcroft Reduction

- Intuition
  - Look for states that can be distinguish from each other
    - End up in different accept / non-accept state with identical input
- Algorithm
  - Construct initial partition
    - Accepting & non-accepting states
  - Iteratively refine 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

\[ J. \text{ Hopcroft, "An n log n algorithm for minimizing states in a finite automaton," } 1971 \]
Splitting Partitions (cont.)

- Need to split partition \(\{S,T,U\}\) into \(\{S,T\}, \{U\}\)
  - Transitions on \(a\) from \(S,T\) lead to partition \(P2\)
  - Transition on \(a\) from \(R\) lead to partition \(P3\)

\[\text{DFA Minimization Algorithm (1)}\]

- **Input** DFA \((\Sigma, Q, q_0, F, \delta)\), Output DFA \((\Sigma, R, F_0, \delta)\)
- **Algorithm**
  - Let \(p_0 = F_0, p_1 = F\) // initial partitions = final, nonfinal states
  - Let \(R = \{ p \mid p \in (p_0,p_1) \text{ and } p \neq \emptyset \}, P = \emptyset \) // add p to R if nonempty
  - While \(P \neq R\) do // while partitions changed on prev iteration
    - Let \(P = R, R = \emptyset\) // P = prev partitions, R = current partitions
    - For each \(p \in P\) // for each partition from previous iteration
      - \(p_2 = p \) \(\text{split}(p, P)\) // split partition, if necessary
      - \(R = R \cup \{ p \mid p \in (p_0,p_1) \text{ and } p \neq \emptyset \}\) // add p to R if nonempty
      - \(r_0 = p \) \(\text{R where } q_0 \in p\) // partition w/ starting state
      - \(F_0 = \{ p \mid p \in \text{R and exists } s \in p \text{ such that } s \in F_0\}\) // parts w/ final states
      - \(\delta(p, c) = q\) when \(\delta(s, c) = r\) where \(s \in p\) and \(r \in q\) // add transitions

Resplitting Partitions

- Need to reexamine partitions after splits
  - Initially no need to split partition \(\{S,T,U\}\)
  - After splitting partition \(\{X,Y\}\) into \(\{X\}, \{Y\}\)
  - Need to split partition \(\{S,T,U\}\) into \(\{S,T\}, \{U\}\)

\[\text{DFA Minimization Algorithm (2)}\]

- **Algorithm for \(\text{split}(p, P)\)**
  - Choose some \(r \in p\), let \(q = p - \{r\}, m = \{\}\) // pick some state \(r\) in \(p\)
  - For each \(s \in q\) // for each state in \(p\) except \(r\)
    - For each \(c \in \Sigma\) // for each symbol in alphabet
      - If \(\delta(r, c) = q_1\) and \(\delta(s, c) = q_2\) and \(q_1 \neq q_2\) then
        - \(m = m \cup \{s\}\) // add \(s\) to \(m\) if \(q_1\)’s not in same partition
  - Return \(p = m, m\) // \(m\) = states that behave differently than \(r\)
    - \(m\) may be \(\emptyset\) if all states behave the same
    - \(p = m\) = states that behave the same as \(r\)

Minimizing DFA: Example 1

- **DFA**
  - Initial partitions
    - Accept \{R\} \(\rightarrow\) P1
    - Reject \{S, T\} \(\rightarrow\) P2
  - Split partition? \(\rightarrow\) Not required, minimization done
    - \(\text{move}(S, a) = T \rightarrow P2\)
    - \(\text{move}(T, a) = T \rightarrow P2\)
    - \(\text{move}(S, b) = R \rightarrow P1\)
    - \(\text{move}(T, b) = R \rightarrow P1\)

Minimizing DFA: Example 2

- **DFA**
  - Initial partitions
    - Accept \{R\} \(\rightarrow\) P1
    - Reject \{S, T\} \(\rightarrow\) P2
  - Split partition? \(\rightarrow\) Not required, minimization done
    - \(\text{move}(S, a) = T \rightarrow P2\)
    - \(\text{move}(T, a) = T \rightarrow P2\)
    - \(\text{move}(S, b) = R \rightarrow P1\)
    - \(\text{move}(T, b) = R \rightarrow P1\)
Minimizing DFA: Example 3

- DFA

  ![DFA Diagram]

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

- Split partition? \rightarrow Yes, different partitions for B
  - move(S, a) = T \rightarrow P2
  - move(S, b) = T \rightarrow P2
  - move(T, a) = T \rightarrow P2
  - move(T, b) = R \rightarrow P1

Complement of DFA

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

  ![Complement DFA Diagram]

Complement of DFA (cont.)

- 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?

Practice

Make the DFA which accepts the complement of the language accepted by the DFA below.

![Practice DFA Diagram]

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

Relating REs to DFAs and NFAs

- Why do we want to convert between these?
  - Can make it easier to express ideas
  - Can be easier to implement
Implementing DFAs

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

Run 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

Implementing DFAs (Alternative)

Alternatively, use generic table-driven DFA.

```c
given components ($\Sigma, Q, q_0, F, \delta$) of a DFA:
let $q = q_0$
while (there exists another symbol $s$ of the input string)
    $q \rightarrow \delta(q, s)$;
if $q \in F$ then
    accept
else reject

- $q$ is just an integer
- Represent $\delta$ using arrays or hash tables
- Represent $F$ as a set
```

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 $(\Sigma, Q_0, q_0, \delta_0, \delta)$, 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

Practice

- Convert to a DFA
- Convert to an NFA and then to a DFA
  - $(0|1)^*11|0^*$
  - Strings of alternating 0 and 1
  - aba" [(ba|b]

Summary of Regular Expression Theory

- Finite automata
  - DFA, NFA
- Equivalence of RE, NFA, DFA
  - RE $\rightarrow$ NFA
    -> Concatenation, union, closure
  - NFA $\rightarrow$ DFA
    -> $\epsilon$-closure & subset algorithm
- DFA
  - Minimization, complement
  - Implementation