CMSC 330: Organization of Programming Languages

Finite Automata 2

Last Lecture

- Finite automata
  - Alphabet, states...
  - ($\Sigma, Q, q_0, F, \delta$)
- Types
  - Deterministic (DFA)
  - Non-deterministic (NFA)

Reducing RE to NFA

- Concatenation
- Union
- Closure

This Lecture

- Reducing NFA to DFA
  - $\varepsilon$-closure
  - Subset algorithm
- Minimizing DFA
  - Hopcroft reduction
- Complementing DFA
- Implementing DFA

How NFA Works

- When NFA processes a string
  - NFA may be in several possible states
    - Multiple transitions with same label
    - $\varepsilon$-transitions
- Example
  - After processing "a"
    - NFA may be in states
      - $S_1$
      - $S_2$
      - $S_3$

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

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 ($\Sigma, Q, q_0, F_n, \delta$)
  - Output
    - DFA ($\Sigma, R, r_0, F_d, \delta$)
  - Using
    - $\varepsilon$-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 ε-transitions
  - If $\exists \ p, p_1, p_2, \ldots, p_n, q \in Q \text{ such that } (p, \varepsilon, p_1) \in \delta, (p_1, \varepsilon, p_2) \in \delta, \ldots, (p_n, \varepsilon, q) \in \delta$

ε-closure($p$)

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

Note

- ε-closure($p$) always includes $p$
- ε-closure() may be applied to set of states (take union)

ε-closure: Example 1

- Following NFA contains
  - $S_1 \xrightarrow{\varepsilon} S_2$
  - $S_2 \xrightarrow{\varepsilon} S_3$
  - $S_1 \xrightarrow{a} S_3$

ε-closures

- ε-closure($S_1$) = $\{S_1, S_2, S_3\}$
- ε-closure($S_2$) = $\{S_2, S_3\}$
- ε-closure($S_3$) = $\{S_3\}$
- ε-closure($\{S_1, S_2\}$) = $\{S_1, S_2, S_3\} \cup \{S_2, S_3\}$

ε-closure: Example 2

- Following NFA contains
  - $S_1 \xrightarrow{\varepsilon} S_3$
  - $S_3 \xrightarrow{a} S_2$
  - $S_1 \xrightarrow{a} S_2$

ε-closures

- ε-closure($S_1$) = $\{S_1, S_2, S_3\}$
- ε-closure($S_2$) = $\{S_2\}$
- ε-closure($S_3$) = $\{S_2, S_3\}$
- ε-closure($\{S_2, S_3\}$) = $\{S_2\} \cup \{S_2, S_3\}$

ε-closure: Practice

- Find ε-closures for following NFA
- Find ε-closures for the NFA you construct for
  - The regular expression $(0|1^*)111(0^*|1)$

ε-closure: Practice

- Find ε-closures for following NFA
- Find ε-closures for the NFA you construct for
  - The regular expression $(0|1^*)111(0^*|1)$

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 $(p, a, q) \in \delta$
    - move($p,a$) = $\{q \mid (p, a, q) \in \delta\}$
  - Note move($p,a$) may be empty $\emptyset$
    - If no transition from $p$ with label $a$

move($a,p$): Example 1

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

Move

- move($S_1, a$) = $\{S_2, S_3\}$
- move($S_1, b$) = $\emptyset$
- move($S_2, a$) = $\emptyset$
- move($S_2, b$) = $\{S_3\}$
- move($S_3, a$) = $\emptyset$
- move($S_3, b$) = $\emptyset$
move(a,p) : Example 2

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

Move
- $\text{move}(S_1, a) = \{S_2\}$
- $\text{move}(S_1, b) = \{S_3\}$
- $\text{move}(S_2, a) = \{S_3\}$
- $\text{move}(S_2, b) = \emptyset$
- $\text{move}(S_3, a) = \emptyset$
- $\text{move}(S_3, b) = \emptyset$

NFA $\rightarrow$ DFA Reduction Algorithm

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

Algorithm
- Let $r_0 = \varepsilon$-closure$(q_0)$, 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 $e = \varepsilon$-closure$(S)$ // states reached via $\varepsilon$
    - If $e \not\in R$ // 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 $F_d = \{r \mid \exists s \in r \& s \in F_n\}$ // final if include state in $F_n$

NFA $\rightarrow$ DFA Example 1

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

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

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

NFA $\rightarrow$ DFA Example 2

- $R = \{[A], [B, D], [C, D]\}$

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 → DFA Example 3

\[ R = \{(A,E), (B,D,E), (C,D), (E)\} \]

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

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
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

DFA Minimization Algorithm (1)
- Input DFA (∑, Q, q0, F, δ)
- Output DFA (∑, R, r0, F, δ)
- Algorithm
  - Let p0 = F, p1 = Q - F, R = \{ p | p \in \{p0,p1\} and p \neq ∅ \}, P = ∅
  - While P \neq R do
    - Let P = R, R = ∅
    - For each p \in P
      - For each s \in p
      - For each c \in ∑
      - If δ(r,c) = q0 and δ(s,c) = q1 and there is no p1 \in P such that q0 \in p1 and q1 \in p1
      - Then
        - m = m \cup \{s\}
    - Return p - m, m

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
- After cleanup

Minimizing DFA: Example 2
- 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) = S → P2
  - move(T,b) = R → P1
- After cleanup
Minimizing DFA: Example 3

- **DFA**

  ![DFA Diagram](image)

  - Initial partitions
    - Accept: \( \{ R \} \) → P1
    - Reject: \( \{ S, T \} \) → P2

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

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](image)

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](image)

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

![Reducing DFAs](image)

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

![Relating REs](image)
Implementing DFAs

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

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

Implementing DFAs (Alternative)

Alternatively, use generic table-driven DFA

```c
given components (Σ, Q, q, f, δ) of a DFA:
let q = q0;
while (there exists another symbol s of the input string)
    q := δ(q, s);
if q ∈ F then accept
else reject
```

Run Time of DFA

- How long for DFA to decide to accept/reject string s?
  - Assume we can compute δ(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 (Σ, Q, qA, fA, δA), 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 → NFA
    - Concatenation, union, closure
  - NFA → DFA
    - ε-closure & subset algorithm
- DFA
  - Minimization, complement
  - Implementation