CMSC330
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*
  – \(\epsilon\)-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
  • \(\epsilon\)-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 \((\Sigma, Q, q_0, F, \delta)\)
  – Output
    • DFA \((\Sigma, R, r_0, F, \delta)\)
  – Using
    • \(\epsilon\)-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
    - \( (p, x, p_i) \in \delta \), \( (p_i, x, p_{i+1}) \in \delta \), \ldots, \( (p_n, x, q) \in \delta \)
- \( \varepsilon \)-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)

**ε-closure: Example 1**

- Following NFA contains
  - \( S_1 \xrightarrow{a} S_2 \)
  - \( S_2 \xrightarrow{b} S_3 \)
  - \( S_1 \xrightarrow{} S_3 \)
- \( \varepsilon \)-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
  - \( S_1 \xrightarrow{} S_3 \)
  - \( S_3 \xrightarrow{a} S_2 \)
  - \( S_1 \xrightarrow{b} S_2 \)
- \( \varepsilon \)-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( \( \{ S_2, S_3 \} \) ) = \( \{ S_2 \} \cup \{ S_2, S_3 \} \)

**ε-closure: Practice**

- Find \( \varepsilon \)-closures for following NFA
  - \( \varepsilon \)-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( \( \{ S_2, S_3 \} \) ) = \( \{ S_2 \} \cup \{ S_2, S_3 \} \)
    - The regular expression \( (0|1)^*111(0^*1) \)

**Calculating 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 \( a \)

**move(\( p, a \)) : 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(p,a) : Example 2

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

- Move
  - move(S1, a) = { S2 }
  - move(S1, b) = { S3 }
  - move(S2, a) = { S3 }
  - move(S2, b) = $\emptyset$
  - move(S3, a) = $\emptyset$
  - move(S3, 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 $S = \{ q \in r \land \text{move}(q, a) = s \}$ // states reached via $a$
      - Let $\delta = \varepsilon$-closure($S$) // states reached via $\varepsilon$
      - If $\delta \notin R$ // if state $\delta$ is new
        - Let $r = r \cup \delta$ // add $\delta$ to $R$ (unmarked)
        - Let $r = r \cup \{ r, a, \delta \}$ // add transition $r \rightarrow a$
      - Let $F_d = \{ r \mid \exists s \in r \land s \in F_n \}$ // final if include state in $F_n$

NFA $\rightarrow$ DFA Example 1

- Start = $\varepsilon$-closure(S1) = { [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 $\rightarrow$ DFA Example 1 (cont.)

- $R = [S1,S3], [S2]$
- $r \in R = [S2]$
- Move([S2],a) = $\emptyset$
- Move([S2],b) = [S3]
  - $e = \varepsilon$-closure([S3]) = [S3]
  - $R = R \cup [S3] = [S1,S3], [S2], [S3]$
  - $\delta = \delta \cup [S2], b, [S3]$

NFA $\rightarrow$ DFA Example 2

- NFA
  - DFA
NFA → DFA Example 3

- NFA
- DFA

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

Equivalence of DFAs and NFAs

- Any string from \([A]\) to either \([D]\) or \([CD]\)
  - Represents a path from \(A\) to \(D\) in the original

Minimizing DFA

- Result from CS DFA
  - 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 \([S, T, U, V]\)
  - 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 \( R \) 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 \( \{X\}, \{Y\} \)
  - Need to split partition \( \{S, T, U\} \) into \( \{S, T\}, \{U\} \)

Minimizing DFA: Example 1
- DFA
  - Initial partitions
    - Accept \( \{R\} \) → \( P_1 \)
    - Reject \( \{S, T\} \) → \( P_2 \)
  - Split partition? → Not required, minimization done
    - move(\( S, a \)) = \( T \) → \( P_2 \)
    - move(\( S, b \)) = \( R \) → \( P_1 \)
    - move(\( T, a \)) = \( T \) → \( P_2 \)
    - move(\( T, b \)) = \( R \) → \( P_1 \)

Minimizing DFA: Example 2
- DFA
  - Initial partitions
    - Accept \( \{R\} \) → \( P_1 \)
    - Reject \( \{S, T\} \) → \( P_2 \)
  - Split partition? → Yes, different partitions for \( B \)
    - move(\( S, a \)) = \( T \) → \( P_2 \)
    - move(\( S, b \)) = \( R \) → \( P_1 \)
    - move(\( T, a \)) = \( T \) → \( P_2 \)
    - move(\( T, b \)) = \( R \) → \( P_1 \)

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 (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
Practice

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

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

Alternatively, use generic table-driven DFA

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

- q is just an integer
- Represent δ using arrays or hash tables
- Represent F as a set

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 \( (\Sigma, Q, q_0, \delta, \delta_0) \), 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
  - \((0|1)^*1|0^*\)
  - Strings of alternating 0 and 1
  - \(aba^*|(ba|b)\)

Summary of Regular Expression Theory

- Finite automata
  - DFA, NFA

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

- DFA
  - Minimization, complement
  - Implementation

Discussion Tomorrow

- I will be in cville
- Going through \(\text{regexp->NFA->DFA->minimization}\) examples
- Quiz!
  - Your ego at work here…
  - Everything from last week (so no RE theory, no NFA/DFA)