Announcements and Reminders

- Project 1 update due to CS dept. maintenance
  - On time projects due Saturday at 10pm!!
  - Late projects due Monday at midnight!
- Quiz 1 next Wednesday, June 16
  - Programming languages, Ruby, RE/FA
- RE/FA practice problems posted online
- Project 2 will be online tonight—due June 25
- Midterm 1 in 2 weeks—Wednesday, June 23
  - Programming languages, Ruby
  - Regular expressions, Finite automata
  - Context-free grammars (lectures next week)

CMSC 330: Organization of Programming Languages

Finite Automata 2

Last Lecture

- Finite automata
  - Alphabet, states...
  - (Σ, Q, q₀, F, δ)
- Types
  - Deterministic (DFA)
  - Non-deterministic (NFA)

This Lecture

- Reducing RE to NFA
  - Concatenation
  - Union
  - Closure
- Reducing NFA to DFA
  - ε-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 ($\Sigma$, $Q$, $q_0$, $F$, $\delta$)
  - Output
    - DFA ($\Sigma$, $R$, $r_0$, $F_d$, $\delta$)
  - Using
    - $\varepsilon$-closure($p$)
    - move($p$, $a$)

$\varepsilon$-transitions and $\varepsilon$-closure

- We say $p \overset{\varepsilon}{\Rightarrow} q$
  - 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
    - $(p, x, p_1) \in \delta$, $(p_1, x, p_2) \in \delta$, $\ldots$, $(p_n, x, q) \in \delta$
- $\varepsilon$-closure($p$)
  - Set of states reachable from $p$ using $\varepsilon$-transitions alone
    - Set of states $q$ such that $p \overset{\varepsilon}{\Rightarrow} q$
    - $\varepsilon$-closure($p$) = {$q$ | $p \overset{\varepsilon}{\Rightarrow} q$}
- 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:
- S1 ⇨ S2
- S2 ⇨ S3
- S1 ⇨ S3

ε-closures:
- ε-closure(S1) = {S1, S2, S3}
- ε-closure(S2) = {S2, S3}
- ε-closure(S3) = {S3}
- ε-closure({S1, S2}) = {S1, S2, S3} ∪ {S2, S3}

ε-closure: Example 2

Following NFA contains:
- S1 ⇨ S3
- S3 ⇨ S2
- S1 ⇨ S2

ε-closures:
- ε-closure(S1) = {S1, S2, S3}
- ε-closure(S2) = {S2}
- ε-closure(S3) = {S2, S3}
- ε-closure({S2, S3}) = {S2} ∪ {S2, S3}

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] ∈ δ
    - move(p,a) = {q | [p, a, q] ∈ δ}
  - Note move(p,a) may be empty Ø
    - If no transition from p with label a
move(a,p) : Example 1

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

Move
- move(S1, a) = \{ S2, S3 \}
- move(S1, b) = \emptyset
- move(S2, a) = \emptyset
- move(S2, b) = \{ S3 \}
- move(S3, a) = \emptyset
- move(S3, b) = \emptyset

move(a,p) : 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 → 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

While \( R \) is unmarked state \( r \) in R
- Mark \( r \)
- For each \( a \in \Sigma \)
  - 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 \( e \in R \)
    - If \( e \) is new
      - Add \( e \) to \( R \) (unmarked)
    - Add transition \( r \rightarrow e \)
  - Let \( \delta = \delta \cup \{ r, a, e \} \) // add transition \( r \rightarrow e \)
- Let \( F_d = \{ r \mid 3 s \in r \text{ with } s \in F_n \} \) // final if include state in \( F_n \)

NFA → DFA Example 1

Start = \( \varepsilon\)-closure(S1) = \{ S1, S3 \}

R = \{ S1, S3 \}

Move(S1, a) = S2

Move(S1, b) = S3

Move(S2, a) = S3

Move(S2, b) = S3

Move(S3, a) = S3

Move(S3, b) = S3

Start

(1,3) a

(2) b

S1

S2

S3

\( \varepsilon \)
NFA → DFA Example 1 (cont.)

• \( R = \{ \{S1,S3\}, \{S2\} \} \)
• \( r \in R = \{S2\} \)
• \( \text{Move}(\{S2\},a) = \emptyset \)
• \( \text{Move}(\{S2\},b) = \{S3\} \)
  \( \text{e} = \varepsilon \text{-closure}(\{S3\}) = \{S3\} \)
  \( R = R \cup \{S3\} = \{ \{S1,S3\}, \{S2\}, \{S3\} \} \)
• \( \delta = \delta \cup \{\{S2\}, b, \{S3\}\} \)

NFA

S1 \[ a \rightarrow \] S2 \[ b \rightarrow \] S3

DFA

(1,3) \[ a \rightarrow ] (2) \[ b \rightarrow ] (3)

NFA → DFA Example 2

NFA

```
\[ A \rightarrow B \]
\[ B \rightarrow C \]
\[ C \rightarrow D \]
```

R = \{ \{A\}, \{B,D\}, \{C,D\} \}

DFA

```
\[ A \rightarrow B, D \]
```

NFA → DFA Example 3

NFA

```
\[ A \rightarrow B \]
\[ B \rightarrow C \]
\[ C \rightarrow D \]
```

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

DFA

```
\[ A \rightarrow B, D, E \]
```

Since S3 \( \in F_n \)
• Done!
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

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 we can 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

J. Hopcroft, "An n log n algorithm for minimizing states in a finite automaton," 1971
Splitting Partitions

- No need to split partition \( \{S,T,U,V\} \)
  - All transitions on \( a \) lead to identical partition \( P_2 \)
  - Even though transitions on \( a \) lead to different states

\[
\begin{array}{c}
S \\
T \\
U \\
V
\end{array}
\rightarrow
\begin{array}{c}
X \\
Y \\
Z
\end{array}
\]

\( P_1 \)
\( P_2 \)

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 \( P_3 \)

\[
\begin{array}{c}
S \\
T \\
U \\
V
\end{array}
\rightarrow
\begin{array}{c}
X \\
Y \\
Z
\end{array}
\]

\( P_1 \)
\( P_2 \)
\( P_3 \)

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 \( \{S,T\}, \{U\} \)

\[
\begin{array}{c}
S \\
T \\
U \\
V
\end{array}
\rightarrow
\begin{array}{c}
P_4 \\
X \\
Y
\end{array}
\]

\( P_1 \)
\( P_2 \)
\( P_3 \)

DFA Minimization Algorithm (1)

- Input DFA \( (\Sigma, Q, q_0, F_n, \delta) \), Output DFA \( (\Sigma, R, r_0, F_d, \delta) \)
- Algorithm
  - Let \( p_0 = F_n \), \( p_1 = Q - F \), \( R = \{ p | p \in (p_0,p_1) \text{ and } p \neq \emptyset \} \), \( P = \emptyset \)
  - While \( P \neq R \) do
    - // iteratively break up partition
      - Let \( P = R \), \( R = \emptyset \)
      - For each \( p \in P \) \( \{p_0,p_1\} = \text{split}(p,P) \) \( \text{// split, if necessary} \)
      - \( R = R \cup \{ p | p \in (p_0,p_1) \text{ and } p \neq \emptyset \} \)
  - \( r_0 = p \in R \text{ where } q_0 \in p \)
  - \( F_d = \{ p | p \in R \text{ and exists } s \in p \text{ such that } s \in F_n \} \)
  - \( \lambda(p,c) = q \text{ when } \lambda(s,c) = r \text{ where } s \in p \text{ and } r \in q \)
**DFA Minimization Algorithm (2)**

- Algorithm for $\text{split}(p, P)$
  - Choose some $r \in p$, let $q = p - \{r\}$, $m = \{\}$
  - For each $s \in q$
    - For each $c \in \Sigma$
      - If $\delta(r, c) = q$ and $\delta(s, c) = q_1$ and
        - there is no $p_1 \in P$ such that $q_0 \in p_1$ and $q_1 \in p_1$
          - $m = m \cup \{s\}$
  - Return $p - m, m$

**Minimizing DFA: Example 1**

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

**Minimizing DFA: Example 2**

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

**Minimizing DFA: Example 3**

- DFA
  - Initial partitions
    - Accept $\{R\} \rightarrow P_1$
    - Reject $\{S, T\} \rightarrow P_2$
  - Split partition? → Yes, different partitions for $B$
    - $\text{move}(S, a) = T \rightarrow P_2$ → $\text{move}(S, b) = T \rightarrow P_2$
    - $\text{move}(T, a) = T \rightarrow P_2$ → $\text{move}(T, b) = R \rightarrow P_1$
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
  - Why not with NFAs?

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

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 '
':     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 '
':     printf("accepted\n"); return 1;
            default:   printf("rejected\n"); return 0;
            }
            break;
    default: printf("unknown state; I'm confused\n");
            break;
    }

Implementing DFAs (Alternative)

Alternatively, use generic table-driven DFA

- 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 $(\Sigma, Q, q_0, (A_1, A_2)$, 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
  - $aba^*|b(ab)b$

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