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

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
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

Another example DFA

- Language?
  - \((ab|aba)^*\)

Relating REs to DFAs and NFAs

- Regular expressions, NFAs, and DFAs accept the same languages!
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
- A DFA accepts \(s\) if it stops at a final state on \(s\)

Formal Definition: Example

- \(\Sigma = \{0, 1\}\)
- \(Q = \{S0, S1\}\)
- \(q_0 = S0\)
- \(F = \{S1\}\)
- \(\delta\) is defined as:
  
<table>
<thead>
<tr>
<th>Input</th>
<th>State 0</th>
<th>State 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>S0</td>
<td>S0</td>
</tr>
<tr>
<td>1</td>
<td>S1</td>
<td>S1</td>
</tr>
</tbody>
</table>

or as \{(S0,0,S01),(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
  - \(\Sigma\) 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
  - \(\delta \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times Q\) specifies the NFA's transitions
    - Transitions on \(\epsilon\) are allowed—can optionally take these transitions without consuming any input
    - Can have more than one transition for a given state and symbol
  - \(\delta\) is a relation, not a function
- An NFA accepts \(s\) if there is at least one path from its start to final state on \(s\)

Reducing Regular Expressions to NFAs

- Goal: Given regular expression \(e\), construct NFA: \(\langle e \rangle = (\Sigma, Q, q_0, F, \delta)\)
  - Remember regular expressions are defined recursively from primitive RE languages
  - Invariant: \(|F| = 1\) in our NFAs
    - Recall \(F\) = set of final states
  - Base case: \(a\)
    - \(\langle a \rangle = \{(a), (S0, S1), S0, \{S1\}, \{(S0, a, S1)\}\}\)
Reduction (cont.)

- Base case: $\varepsilon$
  
  $$\langle \varepsilon \rangle = (\emptyset, \{S0\}, S0, \{S0\}, \emptyset)$$

- Base case: $\emptyset$
  
  $$\langle \emptyset \rangle = (\emptyset, \{S0, S1\}, S0, \{S1\}, \emptyset)$$

Reduction: Concatenation (cont.)

- Induction: $AB$
  
  $$\langle A \rangle \cdot \langle B \rangle = (\Sigma_A \cup \Sigma_B, Q_A \cup Q_B, q_A, \delta_A \cup \delta_B \cup \{(f_A, \varepsilon, q_B)\})$$

Reduction: Concatenation

- Induction: $AB$
  
  $\langle A \rangle \cdot \langle B \rangle$

Reduction: Union

- Induction: $(A \mid B)$
  
  $\langle A \rangle \mid \langle B \rangle$
**Reduction: Union (cont.)**

- Induction: $(A|B)$

\[
\begin{align*}
\langle A \rangle &= (\Sigma_A, Q_A, q_A, \{f_A\}, \delta_A) \\
\langle B \rangle &= (\Sigma_B, Q_B, q_B, \{f_B\}, \delta_B) \\
\langle (A|B) \rangle &= (\Sigma_A \cup \Sigma_B, Q_A \cup Q_B \cup \{S0,S1\}, S0, \{S1\}, \\
&\quad \delta_A \cup \delta_B \cup \{(S0,\epsilon,q_A), (S0,\epsilon,q_B), (f_A,\epsilon,S1), (f_B,\epsilon,S1)\})
\end{align*}
\]

**Reduction: Closure (cont.)**

- Induction: $A^*$

\[
\begin{align*}
\langle A \rangle &= (\Sigma_A, Q_A, q_A, \{f_A\}, \delta_A) \\
\langle A^* \rangle &= (\Sigma_A, Q_A \cup \{S0,S1\}, S0, \{S1\}, \\
&\quad \delta_A \cup \{(f_A,\epsilon,S1), (S0,\epsilon,q_A), (S0,\epsilon,S1), (S1,\epsilon,S0)\})
\end{align*}
\]

**Reduction Complexity**

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

- How many states does $\langle A \rangle$ have?
  - 2 added for each $\mid$, 2 added for each $^*$
  - $O(n)$
  - That’s pretty good!
Practice

- Draw NFAs for the following regular expressions and languages
  - \((0|1)^*110^*\)
  - \(101^*111\)
  - all binary strings ending in 1 (odd numbers)
  - all alphabetic strings which come after “hello” in alphabetic order
  - \((ab^*c|d*a|ab)d\)

Recap

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

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

Reducing RE to NFA
- Concatenation
- Union
- Closure

Next

- 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
    - S1
    - S2
    - S3
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

\[ \begin{array}{c}
\text{NFA}\\
S_1 \rightarrow S_2 \rightarrow \epsilon \rightarrow S_3
\end{array} \quad \begin{array}{c}
\text{DFA}\\
S_1 \rightarrow S_1, S_2, S_3
\end{array} \]

Reducing NFA to DFA (cont.)

- Reduction applied using the \textit{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 two subroutines
    - \(\varepsilon\)-\text{closure}(p)
    - move(p, a)

\(\varepsilon\)-transitions and \(\varepsilon\)-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
    - \(\{p, \varepsilon, p_1\} \in \delta\), \(\{p_1, \varepsilon, p_2\} \in \delta, \ldots, \{p_n, \varepsilon, q\} \in \delta\)
- \(\varepsilon\)-\text{closure}(p)
  - Set of states reachable from \(p\) using \(\varepsilon\)-transitions alone
    - Set of states \(q\) such that \(p \xrightarrow{\varepsilon} q\)
    - \(\varepsilon\)-\text{closure}(p) = \{q \mid p \xrightarrow{\varepsilon} q\}
  - Note
    - \(\varepsilon\)-\text{closure}(p) always includes \(p\)
    - \(\varepsilon\)-\text{closure}( ) may be applied to set of states (take union)

\(\varepsilon\)-closure: Example 1

- Following NFA contains
  - \(S_1 \xrightarrow{\varepsilon} S_2\)
  - \(S_2 \xrightarrow{\varepsilon} S_3\)
  - \(S_1 \xrightarrow{a} S_3\)
    - Since \(S_1 \xrightarrow{\varepsilon} S_2\) and \(S_2 \xrightarrow{\varepsilon} S_3\)
- \(\varepsilon\)-\text{closures}
  - \(\varepsilon\)-\text{closure}(S1) = \{ S1, S2, S3 \}
  - \(\varepsilon\)-\text{closure}(S2) = \{ S2, S3 \}
  - \(\varepsilon\)-\text{closure}(S3) = \{ S3 \}
  - \(\varepsilon\)-\text{closure}( \{ S1, S2 \} ) = \{ S1, S2, S3 \} \cup \{ S2, S3 \} \)
**ε-closure: Example 2**

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

Since $S_1 \xrightarrow{\varepsilon} S_3$ and $S_3 \xrightarrow{\varepsilon} S_2$,

**ε-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:

Find $\varepsilon$-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 $\{p, a, q\} \in \delta$
  - $\text{move}(p, a) = \{q | \{p, a, q\} \in \delta\}$

- 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:
- $\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 $(\Sigma, Q, q_0, F_n, \delta)$, 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 $r \in R$
    - Mark $r$ // each state visited once
    - 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 $e \notin R$
        - Let $R = R \cup \{e\}$ // if state $e$ is new
        - Let $F_d = \{r \mid \exists s \in r \& s \in F_n\}$ // final if include state in $F_n$
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

- \( R = \{ \{A\}, \{B, D\} \} \)

NFA → DFA Example 3

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

Analyzing the reduction

- Any string from \( \{A\} \) to either \( \{D\} \) or \( \{CD\} \)
  - Represents a path from \( A \) to \( D \) in the original NFA
Analyzing the reduction (cont’d)

- 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 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 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 \( P_2 \)
  - 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 \( \{X\}, \{Y\} \)
  - Need to split partition \( \{S,T,U\} \) into \( \{S,T\}, \{U\} \)

---

DFA Minimization Algorithm (1)

- Input DFA \( (\Sigma, Q, q_0, F_n, \delta) \), Output DFA \( (\Sigma, R, r_0, F_d, \delta) \)
- Algorithm
  
  Let \( p_0 = F_n, p_1 = Q - F \) \hspace{1cm} // initial partitions = final, nonfinal states
  Let \( R = \{ p | p \in \{p_0,p_1\} \text{ and } p \neq \emptyset \} \), \( P = \emptyset \) \hspace{1cm} // add \( p \) to \( R \) if nonempty

  While \( P \neq R \) do \hspace{1cm} // while partitions changed on prev iteration

  Let \( P = R \), \( R = \emptyset \) \hspace{1cm} // \( P \) = prev partitions, \( R \) = current partitions

  For each \( p \in P \) \hspace{1cm} // for each partition from previous iteration

  \( (p_0,p_1) = \text{split}(p,P) \) \hspace{1cm} // split partition, if necessary

  \( R = R \cup \{ p | p \in \{p_0,p_1\} \text{ and } p \neq \emptyset \} \) \hspace{1cm} // add \( p \) to \( R \) if nonempty

  \( r_0 = p \in R \text{ where } q_0 \in p \) \hspace{1cm} // partition w/ starting state

  \( F_d = \{ p | p \in R \text{ and exists } s \in p \text{ such that } s \in F_n \} \) \hspace{1cm} // partitions w/ final states

  \( \delta(p,c) = q \text{ when } \delta(s,c) = r \text{ where } s \in p \text{ and } r \in q \) \hspace{1cm} // add transitions

---

DFA Minimization Algorithm (2)

- Algorithm for \( \text{split}(p,P) \)

  Choose some \( r \in p \), let \( q = p - \{r\}, m = \{\} \) \hspace{1cm} // pick some state \( r \) in \( p \)

  For each \( s \in q \) \hspace{1cm} // for each state in \( p \) except for \( r \)

  For each \( c \in \Sigma \) \hspace{1cm} // for each symbol in alphabet

  If \( \delta(r,c) = q_0 \text{ and } \delta(s,c) = q_1 \text{ and } \) \hspace{1cm} // \( q_0 \text{ and } q_1 \) are states reached for \( c \)

  there is no \( p_1 \in P \) such that \( q_0 \in p_1 \) and \( q_1 \in p_1 \) then

  \( m = m \cup \{s\} \) \hspace{1cm} // add \( s \) to \( m \) if \( q_0 \text{ and } q_1 \) are not in same partition

  Return \( p - m, m \) \hspace{1cm} // \( m \) = 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 } = P1
  - Reject { S, T } = P2

- Split partition? → Not required, minimization done
  - move(S,a) = T ∈ P2
  - move(T,a) = T ∈ P2

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(T,a) = S ∈ P2

After cleanup

Minimizing DFA: Example 3

DFA

- Initial partitions
  - Accept { R } = P1 minimal
  - Reject { S, T } = P2

- Split partition? → Yes, different partitions for B
  - move(S,a) = T ∈ P2
  - move(T,a) = T ∈ P2

Complement of DFA

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

A

B

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 (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;
    }
}
```

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 (generic)

More generally, use generic table-driven DFA

```c
given components $(\Sigma, Q, q_A, F, \delta)$ of a DFA:
let $q = q_0$
while (there exists another symbol $s$ of the input string)
    $q := \delta(q, s)$;
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 $(\Sigma, Q_A, q_A, f_A, \delta_A)$, 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
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