Computability Models
Finite-State Machines

Robert M. Keller
Harvey Mudd College
March 2014
Reading

• Read in Sipser, “Introduction to the Theory of Computation”, chapters 0 and 1.
Types of Computability Models

• Machine-Like Models
  • Finite-State Machines
  • Pushdown Automata
  • Turing Machines

• Equational Models
  • Recursive Equations, rewriting
  • Language Equations

• Language Models
  • Grammars
  • Regular Expressions
Why investigate these models?

• Understand what is possible with models of different levels of complexity.

• Understand ultimate limitations of computation.

• Provide insights for algorithm development (for parsing, translation, other problems).
Strings

- Strings are used to represent input and output to machines.

- Strings can be used and produced incrementally.

- Strings also generalize natural numbers.
Alphabets

- Strings are sequences of letters drawn from an alphabet.

- Alphabets are usually finite.

- Σ and Δ are common symbols for alphabets.
**Σ**

- **Σ** is defined to be the set of all *finite* strings over alphabet **Σ**.

- Example Σ = {0, 1}
  \[ Σ^* = \{ε, 0, 1, 00, 01, 10, 11, 000, \ldots\} \]

- ε represents the **empty string** (string of no letters). λ or ∆ are also used for this purpose in some sources.
Inductive Definition of $\Sigma^*$

- **Basis:** $\varepsilon \in \Sigma^*$
- **Induction rule:**
  
  If $x \in \Sigma^*$ and $\sigma \in \Sigma$, then $x\sigma \in \Sigma^*$.

- The only members of are those obtainable by a finite number of rule applications.
String Concatenation

• Concatenation is the main operation on strings.

• Concatenation is indicated by juxtaposition (placing one string after another).

• If $x$ and $y$ are variables with strings as values, then $xy$ means the string consisting of the letters in $x$ followed by those in $y$.

• Example: $x = 011$, $y = 01$, $xy = 01101$. 
Identity Element

- The empty string $\epsilon$ is the identity element for concatenation.

- For any $x \in \Sigma^*$
  - $x\epsilon = x$
  - $\epsilon x = x$
Inductive Definition of Concatenation \( xy \)

- **Basis:** \( \forall x \ x \varepsilon = x \)

- **Induction rule:**
  \[ \forall x \ \forall y \ \forall \sigma \in \Sigma \ x(y\sigma) = (xy)\sigma \]
Strings vs. Natural Numbers

- Natural numbers N can be viewed as a special case of strings over a 1-letter alphabet.
- Let say the letter is just ‘1’.
- Then the connection is suggested by:

<table>
<thead>
<tr>
<th>N</th>
<th>{1}^*</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>ε</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>11</td>
</tr>
<tr>
<td>3</td>
<td>111</td>
</tr>
<tr>
<td>4</td>
<td>1111</td>
</tr>
<tr>
<td>n</td>
<td>1^n (n 1’s in a row)</td>
</tr>
</tbody>
</table>
String Axioms (similar to Peano axioms)

$\Sigma$ elements behave as multiple successors.

- **SA1**: $(\forall x \in \Sigma^*)(\forall \sigma \in \Sigma) x\sigma \neq \epsilon$
- **SA2**: $(\forall x, y \in \Sigma^*)(\forall \sigma, \tau \in \Sigma)
  \hspace{1cm} x\sigma = y\tau \text{ implies } \sigma = \tau \text{ and } x = y$
- **SA3(Induction)**: Let $P(x)$ be any formula with free variable $x$.

\[
(P(\epsilon) \land \forall x (P(x) \rightarrow \forall \sigma P(x\sigma))) \rightarrow \forall x P(x)
\]
Associativity

- String concatenation is obviously associative:
  - For any \( x, y, z \in \Sigma^* \)
    
    \[ x (y z) = (x y) z \]

- This could also be proved by induction, using the inductive definition, similar to proofs for the natural number theory.
Algebraic Structure

• A structure with an associative operation is called a **semigroup**.

• A semigroup with an identity is called a **monoid**.

• Thus $\Sigma^*$ with concatenation and identity $\varepsilon$ is a monoid, sometimes called the **free monoid** on $\Sigma$. 
Length of Strings

- The $| |$ operator on strings denotes the length of a string. This can be defined inductively:

  - **Basis:** $| \varepsilon | = 0$

  - **Induction rule:**
    $\forall x \ \forall \sigma \in \Sigma \ |x\sigma| = |x| + 1$
Length of a Concatenation

- \(|xy| = |x| + |y|\)

- This can be proved by induction.
Letter Count

- For any $\sigma \in \Sigma$
  for any $x \in \Sigma^*$

  $\#_{\sigma}(x) =$
  the number of times $\sigma$ occurs in $x$

- This can be defined by induction.

- Example:
  $\#_0(01010) = 3$, $\#_1(01010) = 2$
Replication

• for any $x \in \Sigma^*$, $x^n$ means the concatenation of $n$ copies of $x$.

• $x^0 = \varepsilon$

• $x^{n+1} = x^n x$
Machines Computing on Strings

- **On-line** models has the machine scan the string one letter at a time.

01101
Types of Output

- A **transducer** produces a string as output as well.
Types of Output

- A **classifier** identifies the input as being in one of several classes.
Types of Output

- An **acceptor** is a classifier with just two classes: accepting and rejecting
Types of Input

- **Tape** models potential can move back and forth on the string.

```
1 1 1 0 1 0 1 0 1 0
```

- Any of the previous types of output are possible, as well as tape output.
Types of Storage

- Some models, such as combinational logic circuits, have no internal storage.

- Finite-state models have a finite set of internal states for storage.

- Other models can have infinite storage, provided by a tape or other mechanisms.
Focus on Acceptors

• Computability theory largely focuses on acceptors.

• Why not much generality is lost:
  • We can convert transducers to classifiers.
  • We can convert classifiers to acceptors.
Transducer to Classifier

- A **transducer** produces a string as output in response to a string of input.

- Each **letter** in the string is produced based on a certain amount of input.

- Treat that letter as indicating one member of the output **class** for that much input.
Classifier to Acceptor

- A classifier identifies a member of a class as its output.

- Encode that class as a **bit vector**, using any of many possible encodings (2-ary, 2-adic, 1-hot, thermometer, etc.). Say this requires \( n \) bits.

- Operate \( n \) acceptors in parallel, each of which provides one bit of the encoding.
Example of an Acceptor (finite-state)

Accepted: 0 0 1 0 0 1

Not accepted: 0 0 1 0 0 1 0
Another Example of an Acceptor (infinite-state)

Accepted: $1^p$ where $p$ is prime

Not accepted: $1^q$ where $q$ is composite
Components of an Acceptor

- Q state set
- \( \Sigma \) input alphabet
- \( \delta : Q \times \Sigma \rightarrow Q \) state-transition function
- \( q_0 \in Q \) initial state
- \( F \subseteq Q \) accepting (aka “final”) state set

These are referenced as a 5-tuple: \((Q, \Sigma, \delta, q_0, F)\).
DFA

• An acceptor with a finite-state set is called a “DFA” (deterministic finite-state acceptor) in the Sipser text.
Behavior of an Acceptor

- There is a transition function $\delta: Q \times \Sigma \to Q$
- Machine starts in state $q_0$.
- From a current state $q$ it changes state to $q'$ with input $\sigma$ provided that
  \[ \delta(q, \sigma) = q' \]
- The machine accepts a string $x \in \Sigma^*$ provided that
  \[ \delta(q_0, x) \in F \]

where $\delta$ has been extended as defined on the next slide.
Extension of $\delta$ to domain $Q \times \Sigma^*$

- $\delta : Q \times \Sigma \rightarrow Q$ is given in the machine definition

- $\delta : Q \times \Sigma^* \rightarrow Q$ is defined inductively:
  - $\forall q \in Q \; \delta(q, \varepsilon) = q$
  - $\forall q \in Q \; \forall x \in \Sigma^* \; \forall \sigma \in \Sigma$
    
    $$\delta(q, x\sigma) = \delta(\delta(q, x), \sigma)$$

Above, the inner $\delta$ is the extended $\delta$, while the outer $\delta$ is the original one.
Presentation of $\delta$

- As a graph, as shown earlier
- By a table:

<table>
<thead>
<tr>
<th>$\delta$</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_0$ initial</td>
<td>$q_0$</td>
<td>$q_1$</td>
</tr>
<tr>
<td>$q_1$ accepting</td>
<td>$q_2$</td>
<td>$q_0$</td>
</tr>
<tr>
<td>$q_2$</td>
<td>$q_3$</td>
<td>$q_2$</td>
</tr>
<tr>
<td>$q_3$ accepting</td>
<td>$q_2$</td>
<td>$q_3$</td>
</tr>
</tbody>
</table>

- As a combination of simpler functions (not shown here)
Concatenation Lemma

- $\forall q \in Q \ \forall x \in \Sigma^* \ \forall y \in \Sigma^*$

  $$\delta(q, xy) = \delta(\delta(q, x), y)$$

- Proof by induction on $y$:
  - Basis: $\delta(q, x\epsilon) = \delta(\delta(q, x), \epsilon) = \delta(q, x)$
  - Induction step: Assume $\delta(q, xy) = \delta(\delta(q, x), y)$, show $\forall \sigma \in \Sigma \ \delta(q, x(y\sigma)) = \delta(\delta(q, x), y\sigma)$.
    By associativity of concatenation, $x(y\sigma) = (xy)\sigma$, so $\delta(q, x(y\sigma)) = \delta(q, (xy)\sigma)$ then, using the definition on the previous page,
    $$\delta(\delta(q, xy), \sigma) = \delta(\delta(q, x), y), \sigma) = \delta(\delta(q, x), y\sigma) .$$
Languages

- A **language** over an alphabet $\Sigma$ is a subset of $\Sigma^*$.  

- In other words, a language is a set of finite strings over a given alphabet.  

- (The subset is not necessarily proper.)
Examples of Languages

- English, over the alphabet \{a, b, \ldots, Z\}.
- Greek, over the alphabet \{\alpha, \beta, \ldots, \Omega\}.
- The language of all zip codes, over the alphabet \{0, 1, 2, \ldots, 9\}
- The language of all odd binary numerals, over the alphabet \{0, 1\}.
- The Python language, over the alphabet \{a, b, \ldots, _, #\}. 
Other Languages

- $\{0, 1\}^*$ the language of all strings of 0’s and 1’s
- $\emptyset$ the empty language
- $\{\varepsilon\}$ the language having one element, the empty string
- $\{0, 1\}$ the language having two strings, one a single-letter string 0, the other the single-letter string 1.
Still More Languages

- \( \{x \in \{0, 1\}^* \mid |x| < 64\} \), the language of all strings of 0’s and 1’s with fewer than 64 letters.

- \( \{x \in \{0, 1\}^* \mid \#_0(x) = \#_1(x)\} \), the language of all strings of 0’s and 1’s with the same number of 0’s as 1’s.

- \( \{0^n1^n \mid n \geq 0\} \), the language of strings of 0’s and 1’s with any number of 0’s followed by the same number of 1’s.
Language of an Acceptor

If $M$ is an acceptor $(Q, \Sigma, \delta, q_0, F)$ then

$$L(M) = \{ x \in \Sigma^* \mid \delta(q_0, x) \in F \}$$

is the language accepted by $M$. 
Example of a Language Accepted

$L = \text{the set of strings of 0's and 1's containing at most one 1.}$
Example of a Language Accepted

L = the set of strings of 0's and 1's with no two 1's in a row.
Example of a Language Accepted

$L = \text{set of binary numerals divisible by 3 MSB first}$
$= \{0, 11, 110, 1001, 1100, \ldots\}$
Construct Acceptors for These Languages

- The language of binary numerals that are divisible by 2, MSB first.
- The language of binary numerals that are divisible by 2, LSB first.
- The language of strings of 1’s and 0’s in which no two consecutive symbols are the same.
Operations on Languages

- As languages are sets, all set operations apply, with their usual meanings:
  - $L \cup M$
  - $L \cap M$
  - $L - M = \{x \in L \mid x \notin M\}$
  - $L \oplus M = (L - M) \cup (M - L)$
- (There is one operation corresponding to each binary propositional connective.)
Operations on Languages

- $LM = \{xy \mid x \in L \land y \in M\}$ is called the “concatenation” of L and M.

- $L^n = LL \ldots L$ (the n-fold concatenation) n times

- Note $L^0 = \{\varepsilon\}$, the language consisting of only the empty string.

- Note: $\emptyset^0 = \{\varepsilon\}$ by definition.
The Star Operation

- \( L^* = \bigcup \{L^n \mid n \in \mathbb{N} \} \) where \( \mathbb{N} \) is the set of natural numbers.

- \( L^* \) is the set of all strings formed by concatenating any number (including 0) of strings from \( L \).
The Plus Operation

- $L^+ = \cup \{L^n \mid n > 0\}$

- $L^+$ is the set of all strings formed by concatenating one or more strings from $L$.

- So $L^* = L^+ \cup \{\varepsilon\}$
Finite-State / Regular Languages

- A language is called `finite-state` if it is $L(M)$ for some finite-state acceptor $M$.

- Finite-State Languages are also called “regular languages”. 
Regular Operations

- The following language operations are called “regular” (Kleene, 1956):
  - concatenation: $LM$
  - union $L \cup M$
  - star $L^*$
Kleene’s Theorem

• A language is finite-state (or regular) iff it can be constructed from a set of finite languages using a finite-number of regular operations.
Example

• \{00, 11\}, \{100\} are two finite languages.

• \(L = \{00, 11\}^* \cup \{100\}\{100\}\) is a language constructed from those language using regular operations.

• Therefore \(L\) is regular according to the theorem.
½ Proof of Kleene’s Theorem

- Constructed from finite languages using regular operations implies there is a finite state acceptor.
Basis

- Every finite language is regular.
- Proof:
  - Construct a graph with nodes corresponding to the strings in the language being accepting states.
  - Direct other transitions to a “dead” state as necessary.
Basis Example

- Suppose the finite language is \(\{0, 00, 01, 111\}\)
- The graph is

![Graph Diagram]
Induction Step

• Suppose L and M are regular. Let A and B be acceptors accepting L and M, respectively.

• We need to show that LM, L ∪ M, and L* by constructing acceptors from A and B.

• It is not immediately obvious how to do this. We first generalize the definition of acceptor, then show how to convert the generalized form to a regular acceptor.
Non-Deterministic Acceptors

- A non-deterministic acceptor is one in which, for each node:
  
a. There can be 0, 1 or more transitions from the node with the same letter.

  
b. There can be “spontaneous” transitions from one node to another, without using a letter. These transitions are appropriately labeled $\varepsilon$, to indicate that no letter of the input is used in making the transition.
NFA transitions for a given state and letter

1 transition

2 transitions

no transitions
Spontaneous Transitions

Chained
Standard Acronyms

- DFA: Deterministic finite-state acceptor
- NFA: Non-deterministic finite-state acceptor
Transition Function for an NFA

- An NFA is represented \((Q, \Sigma, \delta, q_0, F)\) where everything is the same as in a DFA, except for \(\delta\).

- For an NFA:
  \[ \delta: Q \times \Sigma_\varepsilon \rightarrow 2^Q \]
  
  where

  - \(\Sigma_\varepsilon\) means \(\Sigma \cup \{\varepsilon\}\)
  - \(2^Q\) means the set of all subsets of \(Q\).
    (book uses \(P(Q)\)).
NFA Function Example

\[ \delta \]

<table>
<thead>
<tr>
<th>δ</th>
<th>ε</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>∅</td>
<td>{0, 1}</td>
<td>∅</td>
</tr>
<tr>
<td>1</td>
<td>{2}</td>
<td>∅</td>
<td>{0}</td>
</tr>
<tr>
<td>2</td>
<td>∅</td>
<td>{1}</td>
<td>{2}</td>
</tr>
</tbody>
</table>
DFA as a special case of NFA

- DFA has $\delta_{\text{DFA}} : Q \times \Sigma \rightarrow Q$

- NFA has $\delta_{\text{NFA}} : Q \times \Sigma_{\varepsilon} \rightarrow 2^Q$

- In viewing a DFA as an NFA, $\delta_{\text{NFA}}(q, x\sigma) = \{\delta_{\text{DFA}}(q, x\sigma)\}$ so that the set of next states of the NFA is identified with the singleton next state of the DFA.
Acceptance by an NFA

- Consider the graph representation of the NFA.

- A string $x \in \Sigma^*$ is accepted iff $x$ corresponds to a labeled path from the initial state to some accepting state.
NFA Acceptance Example

ε is not accepted
NFA Acceptance Example

0 is accepted
NFA Acceptance Example

1 is not accepted
NFA Acceptance Example

00 is accepted
NFA Acceptance Example

Another way in which 00 is accepted (with $\varepsilon$ used a second time)
NFA Acceptance Example

01 is accepted
NFA Acceptance Example

01 is accepted as on the previous slide, but it is **not** required that **all** paths labeled 01 go from initial to an accepting state.
NFA Acceptance Example

What else?
Unique Accepting State Assumption

- Without loss of generality, it can be assumed that an NFA has exactly one accepting state.

- (If not, introduce a new state and direct $\epsilon$ arcs from every accepting state to it. Then make those states non-accepting and the new state accepting.)

- Note: We cannot do this for a DFA in general.
Unique Accepting State Transformation

Transition diagram:

- State 0
  - Transition on 0 to State 1
  - Transition on ε to State 0
- State 1
  - Transition on ε to State 2
- State 2
  - Transition on 1 to State 1

Initial state: 0
Accepting states: 0, 1, 2
Constructing NFAs for Regular Operations: Union

- Let A and B be NFAs for L and M respectively, with initial states a and b.
- Then an NFA for $L \cup M$ is:
The “Guessing” Paradigm

• The NFA for union exemplifies what is sometimes called “guessing”.

• In order to accept $L \cup M$, the NFA makes a “guess” or free choice as to whether the input string is going to be in $L$ or in $M$. As long as it is in either, the string is accepted.
Constructing NFAs for Regular Operations: Concatenation

- Let A and B be NFAs for L and M respectively. Let a be the unique accepting state of A and let b be the initial state of B.
- Then an NFA for LM is:
Guessing for Concatenation

- Guessing is less obvious in the preceding construction, but it can also be thought to be present.

- When the NFA is in state a having read a portion of the input, it can guess to either stay within A or to make the spontaneous transition to B and continue.
Constructing NFAs for Regular Operations: Star

- Let $A$ be an NFA for $L$. Let $a$ be the initial state of $A$ and $b$ be the unique accepting state of $A$.
- Then an NFA for $L^*$ is:

![Diagram showing an NFA with states $a$, $b$, and $A$, connected with transitions labeled with $\epsilon$.]
Guessing for Star

- When the preceding NFA is started in its initial state, it can “guess” to go to a state to read the rest of the input or go directly to the accepting state.

- In the accepting state, it can choose to go back to the start for more input.
Example: Construct NFA for \((\{01\} \cup \{10\})^*)\)

\[
\begin{align*}
\{01\} & \xrightarrow{0} a \xrightarrow{1} b \xrightarrow{} c \\
\{10\} & \xrightarrow{} d \xrightarrow{1} e \xrightarrow{0} f \\
\end{align*}
\]
Example: NFA for \((\{01\} \cup \{10\})^*\)

\[\{01\} \cup \{10\}\]

\[\text{Unique Accepting State Transformation} \gg\]
Example: NFA for \((\{01\} \cup \{10\})^*)\)
Example: NFA for $\left(\{01\} \cup \{10\}\right)^*$

$\left(\{01\} \cup \{10\}\right)^*$
Shortcuts

In some cases, shortcuts are possible that eliminate states, but be careful, because it is easy to mess up. Some shortcuts for the previous NFA are shown here.
Further Shortcuts
Even More Shortcuts
The Ultimate in Shortcuts
Mini-Project

- Develop a set of graphical rules for shortcuts.
Two-Step Construction

- We now know how to construct NFAs for LM, $L \cup M$, and $L^*$. 

- The next step is to show how to construct, from any NFA, a DFA accepting the same language. 

- The latter is called the Subset Construction.
Subset Construction: NFA to DFA

For any state $q$ in an NFA, define the **closure** $c(q)$ of the state to be $\{q\}$ together with the set of states reachable from $q$ using only $\varepsilon$ transitions.

<table>
<thead>
<tr>
<th>$q$</th>
<th>$c(q)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>${a}$</td>
</tr>
<tr>
<td>b</td>
<td>${b}$</td>
</tr>
<tr>
<td>c</td>
<td>${a,c,d,g,h,i,j}$</td>
</tr>
<tr>
<td>d</td>
<td>${d}$</td>
</tr>
<tr>
<td>e</td>
<td>${e}$</td>
</tr>
<tr>
<td>f</td>
<td>${a,d,f,g,h,i,j}$</td>
</tr>
<tr>
<td>g</td>
<td>${a,d,g,h,j}$</td>
</tr>
<tr>
<td>h</td>
<td>${a,d,h}$</td>
</tr>
<tr>
<td>i</td>
<td>${a,d,g,h,i,j}$</td>
</tr>
</tbody>
</table>

The states of the new DFA will be among these subsets.
Closure of a set of states

For a set of states $S$, define

$$ c(S) = \bigcup \{ c(q) \mid q \in S \} $$

Note: Sipser uses $E(S)$ for this, p 56.
Subset Construction, continued

The initial state of the DFA will be the closure of the initial state.

$\{a,d,g,h,j\}$

Closure of initial state $g$. 
Defining $\delta$ for the DFA

- Each state of the DFA is a subset of the states of the NFA.

- For each $\sigma \in \Sigma$, define

\[
\delta(S, \sigma) = \bigcup \{c(\delta(q, \sigma)) \mid q \in S\}
\]

where inside the braces is the original NFA’s $\delta$. 
Example $\delta$ for the DFA

- $\{a,d,g,h,j\}$ is the initial state

- $\delta(S, \sigma) = \bigcup \{c(\delta(q, \sigma)) \mid q \in S\}$

- $\delta(\{a,d,g,h,j\}, 0) = \bigcup \{c(\delta(q, 0)) \mid q \in \{a,d,g,h,j\}\} = \bigcup \{c(\delta(a, 0)), c(\delta(d, 0)), c(\delta(g, 0)), c(\delta(h, 0)), c(\delta(j, 0))\} = \bigcup \{c(\emptyset), c(\emptyset), c(\emptyset), c(\emptyset), c(\emptyset)\} = \{\emptyset\}$

- similarly $\delta(\{a,d,g,h,j\}, 1) = \{e\}$
Subset Construction, continued

Transitions from the new initial state.

\[ \begin{array}{c}
\{a,d,g,h,j\} \\
\{b\} \\
\{e\}
\end{array} \]
Example $\delta$ for the DFA

- $\delta(S, \sigma) = \bigcup \{ c(\delta(q, \sigma)) \mid q \in S \}$

- $\delta(\{b\}, 0) = \bigcup \{ c(\delta(q, 0)) \mid q \in \{b\} \} = \bigcup \{ \emptyset \} = \emptyset$

- $\delta(\{b\}, 1) = \bigcup \{ c(\delta(q, 1)) \mid q \in \{b\} \} = \{a,c,d,g,h,i,j\}$

![DFA Diagram]
Subset Construction, continued

Transitions from other states

\[
\begin{align*}
\{b\} & \quad 1 \quad \{a,c,d,g,h,i,j\} \\
\{a,d,g,h,j\} & \quad 0 \quad 0 \quad \{a,c,d,g,h,i,j\} \\
\{e\} & \quad 1 \quad \emptyset
\end{align*}
\]
Example $\delta$ for the DFA

- $\delta(S, \sigma) = \cup \{c(\delta(q, \sigma)) \mid q \in S\}$

- $\delta(\{e\}, 0) = \cup \{c(\delta(q, 0)) \mid q \in \{f\}\} = \{a,d,f,g,h,i,j\}$

- $\delta(\{e\}, 0) = \emptyset$
Subset Construction, continued
Subset Construction, completed
Accepting States for the DFA

Any state of the DFA that contains an accepting state of the original NFA is accepting.
Accepting States Outlined

{a,d,g,h,j}  b  \{a,c,d,g,h,i,j\}  
\{a,d,g,h,j\}  e  \{a,d,f,g,h,i,j\}  
\emptyset  0, 1  
\{a,d,f,g,h,i,j\}  1  

\{a,d,g,h,j\}  0  b  \{a,c,d,g,h,i,j\}  
\{a,d,f,g,h,i,j\}  1  
\{a,d,f,g,h,i,j\}  0  

Notes on the Previous Example

• We intentionally used the version of NFA dictated by the regular operators for illustration, even though it would have been simpler to start with the “shortcut” version.

• The three accepting states of the DFA could be merged into one.
Minimized Version

State minimization in general is a separate topic saved for later discussion.
State Reachability

Given a DFA \((\Sigma, Q, q_0, f, F)\) a state \(q \in Q\) is said to be **reachable** provided

\[ \exists x \in \Sigma^* \quad q = \delta(q_0, x) \]

Normally states not reachable, and transitions from them, can be removed without affecting the language accepted.

In the subset construction, it is best to start with the initial state and only construct states reachable from it. This keeps the size of the machine smaller.
State Reachability for NFA

Given an NFA \((\Sigma, Q, q_0, f, F)\) a state \(q \in Q\) is said to be \textbf{reachable} provided

\[
\exists x \in \Sigma^* \quad q \in \delta(q_0, x)
\]

where we extend the given \(\delta:Q \times \Sigma_\varepsilon \to 2^Q\) to \(\Sigma^*\) as follows:

\[
\delta(q_0, \varepsilon) = c(q_0) \quad \text{(the closure of } q_0) \\
\delta(q_0, x\sigma) = \bigcup\{c(\delta(q, \sigma)) \mid q \in \delta(q_0, x)\}
\]

where the rightmost \(\delta\) is the extended version.
Applications of the Subset Construction

• Suppose we want to construct a DFA that will accept the language of strings ending with a given string, for example: 01011.

• The tricky part here is that if we get as input, for example, 01010, while this is not accepted, the last part 010 can possibly be used as the initial part of a string that is accepted, for example 0101011.

• By constructing an appropriate NFA and converting it, it is easy to get the DFA right.
An NFA for \( \{0, 1\}^* \ 01011 \)
Constructing a DFA for \(\{0, 1\}^*\) 01011
Notes on the Previous Example

• Not all subsets were reachable. Those that weren’t were not generated.

• The number of states is the same in the NFA and DFA. This is a coincidence; it may be more or fewer.

• The transition structure is more complex in the DFA. This is typical.

• A similar idea can be used to determine whether a string is contained in a given string.
Algorithmic Applications

- The principle underlying the illustrated method for text matching was the insight and basis for two text searching algorithms:
  - Knuth-Morris-Pratt: Search for a single string
  - Aho-Corasick: Search for a finite set of strings
Another Possibility for Search

- Rather than go through the DFA construction and simulate the result to do search a text, it is possible to simulate the NFA directly.

- In this case there would be a set of “current states” rather than a single one.

- Just use multiple state pointers for the simulation rather than a single one.