CMSC 330: Organization of Programming Languages

DFAs, and NFAs, and Regexps
Goal: Develop an algorithm that determines whether a string \( s \) is matched by regex \( R \)
- I.e., whether \( s \) is a member of \( R \)’s language

Approach: Convert \( R \) to a finite automaton \( FA \) and see whether \( s \) is accepted by \( FA \)
- Details: Convert \( R \) to a nondeterministic FA (NFA), which we then convert to a deterministic FA (DFA),
  - which enjoys a fast acceptance algorithm
Two Types of Finite Automata

- **Deterministic** Finite Automata (DFA)
  - Exactly one sequence of steps for each string
    - Easy to implement acceptance check
  - 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
    - But more expensive to test whether a string matches
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

\[ \varepsilon \rightarrow \text{\varepsilon-transition} \]
DFA for (a|b)*abb
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
Comparing NFA and DFA for (ab|aba)*

DFA

NFA
Quiz 1: Which DFA matches this regexp?

\[ b (b | a+b?) \]

A. 

B. 

C. 

D. None of the above

CMSC 330 Fall 2019
Quiz 1: Which DFA matches this regexp?

\[ b (b | a+b?) \]

A.

B.

C.

D. None of the above
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

  Ð What's this definition saying that \(\delta\) is?

- A DFA accepts \(s\) if it stops at a final state on \(s\)
Formal Definition: Example

• $\Sigma = \{0, 1\}$
• $Q = \{S_0, S_1\}$
• $q_0 = S_0$
• $F = \{S_1\}$

or as $\{(S_0,0,S_0),(S_0,1,S_1),(S_1,0,S_0),(S_1,1,S_1)\}$
Implementing DFAs (one-off)

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

```
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");
    }
}
```
Implementing DFAs (generic)

More generally, use generic table-driven DFA

\[
\text{given components } (\Sigma, Q, q_0, F, \delta) \text{ of a DFA:}
\]

\[
\text{let } q = q_0
\]

\[
\text{while (there exists another symbol } \sigma \text{ of the input string)}
\]

\[
q := \delta(q, \sigma);
\]

\[
\text{if } q \in F \text{ then}
\]

\[
\text{accept}
\]

\[
\text{else reject}
\]

- \text{q is just an integer}
- \text{Represent } \delta \text{ using arrays or hash tables}
- \text{Represent } F \text{ as a set}
Nondeterministic Finite Automata (NFA)

- An NFA is a 5-tuple \((\Sigma, Q, q_0, F, \delta)\) where
  - \(\Sigma, Q, q_0, F\) as with DFAs
  - \(\delta \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times Q\) specifies the NFA's transitions

Example

- \(\Sigma = \{a\}\)
- \(Q = \{S1, S2, S3\}\)
- \(q_0 = S1\)
- \(F = \{S3\}\)
- \(\delta = \{ (S1,a,S1), (S1,a,S2), (S2,\varepsilon,S3) \}\)

An NFA accepts \(s\) if there is at least one path via \(s\) from the NFA’s start state to a final state

CMSC 330 Fall 2019
NFA Acceptance Algorithm (Sketch)

- When NFA processes a string $s$
  - NFA must keep track of several “current states”
    - Due to multiple transitions with same label, and $\varepsilon$-transitions
  - If any current state is final when done then accept $s$

- Example
  - After processing “a”
    - NFA may be in states $S1$, $S2$, $S3$
    - Since $S3$ is final, $s$ is accepted

- Algorithm is slow, space-inefficient; prefer DFAs!
Relating REs to DFAs and NFAs

- Regular expressions, NFAs, and DFAs accept the same languages! *Can convert between them*

NB. Both *transform* and *reduce* are historical terms; they mean “convert”
Reducing Regular Expressions to NFAs

- Goal: Given regular expression $A$, construct NFA: $<A> = (\Sigma, Q, q_0, F, \delta)$
  - Remember regular expressions are defined recursively from primitive RE languages
  - Invariant: $|F| = 1$ in our NFAs
    - Recall $F =$ set of final states

- Will define $<A>$ for base cases: $\sigma$, $\varepsilon$, $\emptyset$
  - Where $\sigma$ is a symbol in $\Sigma$
- And for inductive cases: $AB$, $A|B$, $A^*$
Reducing Regular Expressions to NFAs

- Base case: $\sigma$

Recall: NFA is $(\Sigma, Q, q_0, F, \delta)$ where
  - $\Sigma$ is the alphabet
  - $Q$ is set of states
  - $q_0$ is starting state
  - $F$ is set of final states
  - $\delta$ is transition relation

$<\sigma> = (\{\sigma\}, \{S0, S1\}, S0, \{S1\}, \{(S0, \sigma, S1)\})$
Reduction

- **Base case:** $\varepsilon$

  \[<\varepsilon> = (\emptyset, \{S_0\}, S_0, \{S_0\}, \emptyset)\]

- **Base case:** $\emptyset$

  \[<\emptyset> = (\emptyset, \{S_0, S_1\}, S_0, \{S_1\}, \emptyset)\]
Reduction: Concatenation

Induction: $AB$

- $<A> = (\Sigma_A, Q_A, q_A, \{f_A\}, \delta_A)$
- $<B> = (\Sigma_B, Q_B, q_B, \{f_B\}, \delta_B)$
Reduction: Concatenation

- Induction: $AB$

\[<A> = (\Sigma_A, Q_A, q_A, \{f_A\}, \delta_A)\]
\[<B> = (\Sigma_B, Q_B, q_B, \{f_B\}, \delta_B)\]
\[<AB> = (\Sigma_A \cup \Sigma_B, Q_A \cup Q_B, q_A, \{f_B\}, \delta_A \cup \delta_B \cup \{(f_A, \varepsilon, q_B)\})\]
Reduction: Union

Induction: $A|B$

- $\langle A \rangle = (\Sigma_A, Q_A, q_A, \{f_A\}, \delta_A)$
- $\langle B \rangle = (\Sigma_B, Q_B, q_B, \{f_B\}, \delta_B)$
Reduction: Union

Induction: $A|B$

- $<A> = (\Sigma_A, Q_A, q_A, \{f_A\}, \delta_A)$
- $<B> = (\Sigma_B, Q_B, q_B, \{f_B\}, \delta_B)$
- $<A|B> = (\Sigma_A \cup \Sigma_B, Q_A \cup Q_B \cup \{S0,S1\}, S0, \{S1\}, \delta_A \cup \delta_B \cup \{(S0,\varepsilon,q_A), (S0,\varepsilon,q_B), (f_A,\varepsilon,S1), (f_B,\varepsilon,S1)\})$
Reduction: Closure

- Induction: $A^*$

- $<A> = (\Sigma_A, Q_A, q_A, \{f_A\}, \delta_A)$
Reduction: Closure

Induction: $A^*$

- $<A> = (\Sigma_A, Q_A, q_A, \{f_A\}, \delta_A)$
- $<A^*> = (\Sigma_A, Q_A \cup \{S0, S1\}, S0, \{S1\},$
  \[\delta_A \cup \{(f_A, \varepsilon, S1), (S0, \varepsilon, q_A), (S0, \varepsilon, S1), (S1, \varepsilon, S0)\})\]
Quiz 2: Which NFA matches $a^*$?
Quiz 2: Which NFA matches $a^*$?
Quiz 3: Which NFA matches $a|b^*$?
Quiz 3: Which NFA matches $a|b^*$?
Reduction Complexity

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

- How many states does $<A>$ have?
  
  • Two added for each $\mid$, two added for each $*$
  • $O(n)$
  • That’s pretty good!
Reducing NFA to DFA
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 “current states”

- Example
Algorithm for Reducing NFA to DFA

- 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 two subroutines
  - $\varepsilon$-closure($\delta$, $p$) (and $\varepsilon$-closure($\delta$, $Q$))
  - move($\delta$, $p$, $\sigma$) (and move($\delta$, $Q$, $\sigma$))
    - (where $p$ is an NFA state)
**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 in \( \delta \)
  - 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, \( \{p_n,\varepsilon,q\} \in \delta \)

- **ε-closure(\( \delta \), \( p \))**
  - Set of states reachable from \( p \) using \( \varepsilon \)-transitions alone
    - Set of states \( q \) such that \( p \xrightarrow{\varepsilon} q \) according to \( \delta \)
    - \( \varepsilon \)-closure(\( \delta \), \( p \)) = \{ \( q \mid p \xrightarrow{\varepsilon} q \) in \( \delta \) \}
    - \( \varepsilon \)-closure(\( \delta \), \( Q \)) = \{ \( q \mid p \in Q, p \xrightarrow{\varepsilon} q \) in \( \delta \) \}
  - **Notes**
    - \( \varepsilon \)-closure(\( \delta \), \( p \)) always includes \( p \)
    - We write \( \varepsilon \)-closure(\( p \)) or \( \varepsilon \)-closure(\( Q \)) when \( \delta \) is clear from context
**ε-closure: Example 1**

- Following NFA contains
  - $p_1 \xrightarrow{\varepsilon} p_2$
  - $p_2 \xrightarrow{\varepsilon} p_3$
  - $p_1 \xrightarrow{\varepsilon} p_3$

- Since $p_1 \xrightarrow{\varepsilon} p_2$ and $p_2 \xrightarrow{\varepsilon} p_3$

- **ε-closures**
  - $\varepsilon$-closure($p_1$) = \{ $p_1$, $p_2$, $p_3$ \}
  - $\varepsilon$-closure($p_2$) = \{ $p_2$, $p_3$ \}
  - $\varepsilon$-closure($p_3$) = \{ $p_3$ \}
  - $\varepsilon$-closure( \{ $p_1$, $p_2$ \} ) = \{ $p_1$, $p_2$, $p_3$ \} $\cup$ \{ $p_2$, $p_3$ \}
ε-closure: Example 2

Following NFA contains

- p1 $\xrightarrow{\varepsilon}$ p3
- p3 $\xrightarrow{\varepsilon}$ p2
- p1 $\xrightarrow{\varepsilon}$ p2

Since p1 $\xrightarrow{\varepsilon}$ p3 and p3 $\xrightarrow{\varepsilon}$ p2

ε-closures

- $\varepsilon$-closure(p1) = \{ p1, p2, p3 \}
- $\varepsilon$-closure(p2) = \{ p2 \}
- $\varepsilon$-closure(p3) = \{ p2, p3 \}
- $\varepsilon$-closure( \{ p2, p3 \} ) = \{ p2 \} $\cup$ \{ p2, p3 \}
ε-closure Algorithm: Approach

Input: NFA ($\Sigma$, $Q$, $q_0$, $F_n$, $\delta$), State Set $R$
Output: State Set $R'$

Algorithm

Let $R' = R$ // start states
Repeat
    Let $R = R'$ // continue from previous
    Let $R' = R \cup \{ q \mid p \in R, (p, \epsilon, q) \in \delta \}$ // new $\epsilon$-reachable states
Until $R = R'$ // stop when no new states

This algorithm computes a fixed point
ε-closure Algorithm Example

Calculate $\varepsilon$-closure($\delta$, {p1})

<table>
<thead>
<tr>
<th></th>
<th>R</th>
<th>R'</th>
</tr>
</thead>
<tbody>
<tr>
<td>{p1}</td>
<td>{p1}</td>
<td></td>
</tr>
<tr>
<td>{p1}</td>
<td>{p1, p2}</td>
<td></td>
</tr>
<tr>
<td>{p1, p2}</td>
<td>{p1, p2, p3}</td>
<td></td>
</tr>
<tr>
<td>{p1, p2, p3}</td>
<td>{p1, p2, p3}</td>
<td></td>
</tr>
</tbody>
</table>

Let $R' = R$
Repeat
Let $R = R'$
Let $R' = R \cup \{q \mid p \in R, (p, \varepsilon, q) \in \delta\}$
Until $R = R'$
Calculating move(p,σ)

move(δ,p,σ)

- **Set of states** reachable from p using exactly one transition on symbol σ
  - Set of states q such that \{p, σ, q\} ∈ δ
  - \(move(δ,p,σ) = \{ q | \{p, σ, q\} ∈ δ \}\)
  - \(move(δ,Q,σ) = \{ q | p ∈ Q, \{p, σ, q\} ∈ δ \}\)
    - i.e., can “lift” move() to a set of states Q

**Notes:**
- \(move(δ,p,σ)\) is ∅ if no transition \(p,σ,q\) ∈ δ, for any q
- We write move(p,σ) or move(R,σ) when δ clear from context
move(p, σ) : Example 1

- Following NFA
  - Σ = \{ a, b \}

- Move
  - move(p1, a) = \{ p2, p3 \}
  - move(p1, b) = ∅
  - move(p2, a) = ∅
  - move(p2, b) = \{ p3 \}
  - move(p3, a) = ∅
  - move(p3, b) = ∅

move(\{p1, p2\}, b) = \{ p3 \}
move(p, σ) : Example 2

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

- Move
  - \( \text{move}(p_1, a) = \{ p_2 \} \)
  - \( \text{move}(p_1, b) = \{ p_3 \} \)
  - \( \text{move}(p_2, a) = \{ p_3 \} \)
  - \( \text{move}(p_2, b) = \emptyset \)
  - \( \text{move}(p_3, a) = \emptyset \)
  - \( \text{move}(p_3, b) = \emptyset \)
  - \( \text{move}\{p_1, p_2\}, a\) = \{p_2, p_3\}
NFA → DFA Reduction Algorithm ("subset")

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

- **Algorithm**
  
  Let $r_0 = \varepsilon$-closure($\delta$, $q_0$), 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 $\sigma \in \Sigma$  
  // for each symbol $\sigma$
  
  Let $E = \text{move}(\delta, r, \sigma)$  
  // states reached via $\sigma$
  
  Let $e = \varepsilon$-closure($\delta$, $E$)  
  // states reached via $\varepsilon$
  
  If $e \notin R$  
  // if state $e$ is new
  
  Let $R = R \cup \{e\}$  
  // add $e$ to $R$ (unmarked)
  
  Let $\delta' = \delta' \cup \{r, \sigma, e\}$  
  // add transition $r \rightarrow e$ on $\sigma$
  
  Let $F_d = \{r \mid \exists s \in r \text{ with } s \in F_n\}$  
  // final if include state in $F_n$
NFA $\rightarrow$ DFA Example 1

- Start = $\varepsilon$-closure($\delta$, p1) = { {p1, p3} }
- $R = \{ \{p1, p3\} \}$
- $r \in R = \{p1, p3\}$
- move($\delta$, {p1, p3}, a) = {p2}
  - $e = \varepsilon$-closure($\delta$, {p2}) = {p2}
  - $R = R \cup \{\{p2\}\} = \{\{p1, p3\}, \{p2\}\}$
  - $\delta' = \delta' \cup \{\{p1, p3\}, a, \{p2\}\}$
- move($\delta$, {p1, p3}, b) = $\emptyset$
NFA → DFA Example 1 (cont.)

• \( R = \{ \{p1,p3\}, \{p2\} \} \)
• \( r \in R = \{p2\} \)
• \( \text{move}(\delta,\{p2\},a) = \emptyset \)
• \( \text{move}(\delta,\{p2\},b) = \{p3\} \)
  - \( e = \varepsilon\text{-closure}(\delta,\{p3\}) = \{p3\} \)
  - \( R = R \cup \{\{p3\}\} = \{ \{p1,p3\}, \{p2\}, \{p3\} \} \)
  - \( \delta' = \delta' \cup \{\{p2\}, b, \{p3\}\} \)
• $R = \{ \{p_1,p_3\}, \{p_2\}, \{p_3\} \}$
• $r \in R = \{p_3\}$
• $\text{Move}(\{p_3\},a) = \emptyset$
• $\text{Move}(\{p_3\},b) = \emptyset$
• Mark $\{p_3\}$, exit loop
• $F_d = \{\{p_1,p_3\}, \{p_3\}\}$
  > Since $p_3 \in F_n$
• Done!

DFA

NFA
NFA → DFA Example 2

NFA

DFA

R = \{ \{A\}, \{B,D\}, \{C,D\} \}
Quiz 4: Which DFA is equivalent to this NFA?

NFA:

A.

B.

C.

D. None of the above
Quiz 4: Which DFA is equivalent to this NFA?

NFA:

DFA:

A.

B.

C.

D. None of the above
Actual Answer

NFA:
NFA → DFA Example 3

R = \{ \{A, E\}, \{B, D, E\}, \{C, D\}, \{E\} \}
NFA → DFA Practice

Diagram: A non-deterministic finite automaton (NFA) with transitions labeled as follows:
- From state 0:
  - Transition on 'a' to state 1
  - Transition on 'b' to state 3
- From state 1:
  - Transition on 'a' to state 2
- From state 2:
  - Transition on 'a' to state 0
  - Transition on ε to state 2
- From state 3:
  - Transition on 'a' to state 1
- From state 0 to state 3:
  - Transition labeled 'b'

The DFA should be constructed to accept the same language as the NFA.
NFA $\rightarrow$ DFA Practice
Analyzing the Reduction

- 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 $O(2^n)$.

![NFA and DFA diagrams](image-url)
Recap: Matching a Regexp $R$

- Given $R$, construct NFA. Takes time $O(R)$
- Convert NFA to DFA. Takes time $O(2^{|R|})$
  - But usually not the worst case in practice
- Use DFA to accept/reject string $s$
  - Assume we can compute $\delta(q, \sigma)$ in constant time
  - Then time to process $s$ is $O(|s|)$
    - Can’t get much faster!

- Constructing the DFA is a one-time cost
  - But then processing strings is fast
Closing the Loop: Reducing DFA to RE
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
Minimizing DFAs

- Every regular language is recognizable by a unique minimum-state DFA
  - Ignoring the particular names of states

- In other words
  - For every DFA, there is a unique DFA with minimum number of states that accepts the same language
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 split 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 \textit{a} lead to identical partition P2
  - Even though transitions on \textit{a} lead to different states

\begin{tikzpicture}
  \node[state, fill=cyan!50] (s) at (0,0) {S};
  \node[state, fill=cyan!50] (t) at (1,0) {T};
  \node[state, fill=cyan!50] (u) at (0,-1) {U};
  \node[state, fill=cyan!50] (v) at (1,-1) {V};
  \node[state, fill=black] (x) at (2,0) {X};
  \node[state, fill=black] (y) at (2,-1) {Y};
  \node[state, fill=black] (z) at (3,-1) {Z};

  \draw[->, thick] (s) -- (x) node [midway, above] {a};
  \draw[->, thick] (s) -- (y) node [midway, above] {a};
  \draw[->, thick] (s) -- (z) node [midway, above] {a};
  \draw[->, thick] (t) -- (x) node [midway, above] {a};
  \draw[->, thick] (t) -- (y) node [midway, above] {a};
  \draw[->, thick] (t) -- (z) node [midway, above] {a};
  \draw[->, thick] (u) -- (x) node [midway, above] {a};
  \draw[->, thick] (u) -- (y) node [midway, above] {a};
  \draw[->, thick] (u) -- (z) node [midway, above] {a};
  \draw[->, thick] (v) -- (x) node [midway, above] {a};
  \draw[->, thick] (v) -- (y) node [midway, above] {a};
  \draw[->, thick] (v) -- (z) node [midway, above] {a};

  \node[draw=red, thick, rounded corners=5, fill=red!50, fit=(s) (t) (u) (v)] (P1) {};\node[anchor=north west] at (P1.north west) {P1};;
  \node[draw=red, thick, rounded corners=5, fill=red!50, fit=(x) (y) (z)] (P2) {};\node[anchor=north west] at (P2.north west) {P2};
\end{tikzpicture}
Splitting Partitions (cont.)

- Need to split partition \{S,T,U\} into \{S,T\}, \{U\}
  - Transitions on \textit{a} from S,T lead to partition \textit{P2}
  - Transition on \textit{a} from U lead to partition \textit{P3}
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\} we need to split partition \{S,T,U\} into \{S,T\}, \{U\}
Minimizing DFA: Example 1

- DFA

- Initial partitions

- Split partition
Minimizing DFA: Example 1

- **DFA**

  ![DFA Diagram]

  - **Initial partitions**
    - Accept: \( \{ R \} = P_1 \)
    - Reject: \( \{ S, T \} = P_2 \)

  - **Split partition?** → Not required, minimization done
    - \( \text{move}(S,a) = T \in P_2 \) → \( \text{move}(S,b) = R \in P_1 \)
    - \( \text{move}(T,a) = T \in P_2 \) → \( \text{move}(T,b) = R \in P_1 \)
Minimizing DFA: Example 2
Minimizing DFA: Example 2

- **DFA**

  ![Diagram of DFA](image)

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

- **Split partition? → Yes, different partitions for B**
  - move(S,a) = T ∈ P2 – move(S,b) = T ∈ P2
  - move(T,a) = T ∈ P2 – move (T,b) = R ∈ P1

**DFA already minimal**
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

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?
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, complementation