CHAPTER 7: KLEENE’S THEOREM *

Peter Cappello
Department of Computer Science
University of California, Santa Barbara
Santa Barbara, CA 93106
cappello@cs.ucsb.edu

• The corresponding textbook chapter should be read before attending this lecture.

• These notes are not intended to be complete. They are supplemented with figures, and other material that arises during the lecture period in response to questions.

*Based on Theory of Computing, 2nd Ed., D. Cohen, John Wiley & Sons, Inc.
Unification

Theorem: Any language that can be defined by
- a regular expression, or
- a finite automaton, or
- a transition graph

can be defined by all 3 methods.

This is the fundamental theorem of finite automata.
Proof Architecture

Part 1 Every language that can be defined by an FA can be defined by a TG.

Part 2 Every language that can be defined by a TG can be defined by a regular expression.

Part 3 Every language that can be defined by a regular expression can be defined by an FA.

Proof of Part 1

Since every FA can be viewed as a TG that accepts the same language, Part 1 is true.
TURNING TGs INTO REGULAR EXPRESSIONS

Proof of Part 2

We give an algorithm whose input is a TG and whose output is a regular expression that denotes the same language accepted by the TG.
Transform the TG into $TG'$ with a unique start state

1. $TG'$ starts as a copy of TG.

2. Add a new start state, $s'$ to $TG'$.

3. Add $\Lambda$-transitions to $TG'$ from $s'$ to the pre-existing start states.

4. Remove each pre-existing start state from $S$, the set of start states.
   - Illustrate.
   - $TG'$ has only 1 start state.
   - If word $w$ is accepted by the original TG, it is accepted by $TG'$.
   - If word $w$ is accepted by $TG'$, it is accepted by the original TG; every start state of the original TG is reachable from $s'$ without consuming any input.
Transform the \( TG' \) into \( TG'' \) with a unique final state

1. \( TG'' \) starts as a copy of \( TG' \).

2. Add a new \textit{final} state, \( f'' \) to \( TG'' \). We require \( s' \neq f'' \).

3. Add \( \Lambda \)-transitions to \( TG' \) from each pre-existing final state to \( f'' \).

4. Remove each pre-existing final state from \( F \), the set of final states.
   - Illustrate.
   - \( TG'' \) has only 1 final state.
   - If word \( w \) is accepted by \( TG' \), it is accepted by \( TG'' \).
   - If word \( w \) is accepted by \( TG'' \), it is accepted by \( TG' \); if there is a successful path for \( w \) in \( TG'' \), it must go through a final state in \( TG' \), and transits without consuming any more input to \( f'' \).
Some Transformation Methods

`eliminateMultipleSelfLoops()` {
    replace multiple self-loops with 1 self-loop labelled with the sum of the labels.
}

Illustrate.

`eliminateMultipleTransitions()` {
    if there are multiple transitions between states $p$ and $q$,
    replace them with 1 transition, labelled with the sum of the labels.
}

Illustrate.
bypassAState(p, q, t) {
    if (p, e1, q) and (q, e2, t) are transitions,
    where e1 and e2 are strings or regular expressions,
    replace them with transition (p, e1 e2, t).
}

Illustrate.

bypassASelfLoopState(p, q, t) {
    if (p, e1, q) and (q, e2, t) are transitions,
    where e1 and e2 are strings or regular expressions,
    and q has a self-loop labelled e3,
    replace them with transition (p, e1 e3* e2, t).
}

Illustrate.
omitAState() {
    if a state has transitions neither into nor out of it, 
    omit it.
}

Illustrate.

eliminateMultipleTransitions(s, f) {
    if the GTG has only a start and a final state with multiple 
    transitions, replace the transitions with their sum.
}

Illustrate.
Algorithm

1. Transform the TG so that it has a unique source start state, and a unique sink final state.

2. while ( |Q| > 2 ) eliminate any internal state:
   
   If state q’s inDegree = i and its outDegree = o, perform \( i \times o \) bypass operations. Delete q.

   Eliminate multiple transitions between 2 states, in favor of a single transition, labelled with the sum of the labels.

3. When only the start and final state remain, output the sum of the regular expressions on the arcs connecting them.

   Illustrate.
WHERE WE ARE IN THE PROOF

**Part 1** Every language that can be defined by an FA can be defined by a TG.

**Part 2** Every language that can be defined by a TG can be defined by a regular expression.

**Part 3** Every language that can be defined by a regular expression can be defined by an FA.
Proof of Part 3

• The proof exploits the recursive nature of regular expressions.
• For each of the 3 rules for building a regular expression, we give a construction of an FA.
• Thus, as we build up the regular expression, we build up an FA that accepts the language denoted by the regular expression.
Proof of Part 3, Rule 1

Rule 1

• There is an FA that accepts the empty language.
• There is an FA that accepts only the word \( \Lambda \).
• For each letter of the alphabet, there is an FA that accepts the 1 word consisting of only that letter.

Illustrate.
Rule 2 Given:

- FA $E$ that accepts the language denoted by regular expression $e$
- FA $F$ that accepts the language denoted by regular expression $f$

There exists an FA that accepts $L(e + f)$.

Illustrate with:

- $(a + b)^*aa(a + b)^*$, all strings with a double $a$
- $(a + b)^*b$, all strings that end in $b$. 
Proof of Part 3, Rule 2: Construction

Construct $FA \ Z = (Q_Z, \Sigma, z_0, F_Z, \delta_Z)$, given

- $FA \ X = (Q_X, \Sigma, x_0, F_X, \delta_X)$
- $FA \ Y = (Q_Y, \Sigma, y_0, F_Y, \delta_Y)$.

1. $Q_Z = Q_X \times Q_Y$.
2. $z_0 = (x_0, y_0)$.
3. $(x, y) \in F_Z$ if $x \in F_X$ or $y \in F_Y$.
4. $\delta_Z((x, y), a) = (x', y')$, where
   - $\delta_X(x, a) = x'$, for $x, x' \in Q_X$ and $a \in \Sigma$.
   - $\delta_Y(y, a) = y'$, for $y, y' \in Q_Y$ and $a \in \Sigma$. 


**Part 3, Rule 2: Another illustration**

Illustrate with:

- $(b + ab^*a)^*$, all strings with an even number of $a$’s
- $(a + ba^*b)^*$, all strings with an even number of $b$’s.
Proof of Part 3, Rule 3

Rule 3 Given:

- FA $E$ that accepts the language denoted by regular expression $e$
- FA $F$ that accepts the language denoted by regular expression $f$

There exists an FA that accepts $L(ef)$.

Illustrate with:

- $(a + b)^*aa(a + b)^*$, all strings with a double $a$
- $(a + b)^*b$, all strings that end in $b$. 
Proof of Part 3, Rule 3: Construction

Construct FA $Z = (Q_Z, \Sigma, z_0, F_Z, \delta_Z)$, given

- FA $X = (Q_X, \Sigma, x_0, F_X, \delta_X)$
- FA $Y = (Q_Y, \Sigma, y_0, F_Y, \delta_Y)$.

1. $Q_Z = Q_X \times 2^{Q_Y}$.

2. Start state:
   $$z_0 = \begin{cases} (x_0, \{\} ), & x_0 \notin Q_X; \\ (x_0, \{y_0\}), & \text{otherwise}. \end{cases}$$

3. Final states: $(x, \{y_{i_1}, y_{i_2}, \ldots, y_{i_j}\}) \in F_Z$ if any $y_i \in F_Y$. 

18
4. $\delta$ function. Let

- $(x, \{y_{i_1}, y_{i_2}, \ldots, y_{i_j}\}) \in Q_Z$
- $\delta_X(x, a) = x'$, for $x, x' \in Q_X$ and $a \in \Sigma$.
- $\delta_Y(y_{i_k}, a) = y'_{i_k}$, for $y_{i_k}, y'_{i_k} \in Q_Y$ and $a \in \Sigma$.

Then,

$$\delta_Z((x, \{y_{i_1}, y_{i_2}, \ldots, y_{i_j}\}), a) =\begin{cases} 
(x', \{y'_{i_1}, y'_{i_2}, \ldots, y'_{i_j}\}), & x' \notin F_X; \\
(x', \{y'_{i_1}, y'_{i_2}, \ldots, y'_{i_j}\} \cup \{y_0\}), & \text{otherwise}
\end{cases}$$

Illustrate with:

- All strings without a double $a$.
- All strings of odd length.
Proof of Part 3, Rule 4

- **Rule 4:** Let FA $X$ accept the language denoted by regular expression $e$.

- There exists an FA $Z$ that accepts $L(e^*)$.

- The ideas:
  
  - When $X$ is in a final state, the simulation continues *and* jumps to $x_0$.
  
  - Accept $\Lambda$: $Z^*$’s start state is a final state.
  
  - Be careful: The start state of $X$ may:
    
    * not be final
    
    * have incoming arcs.
Why is this a potential problem?

- Illustrate with $a^* + aa^*b$. 
Proof of Part 3, Rule 4: Construction

Construct FA \( Z = (Q_Z, \Sigma, z_0, F_Z, \delta_Z) \), given

\( \bullet \) FA \( X = (Q_X, \Sigma, x_0, F_X, \delta_X) \).

1. \( Q_Z = 2^{Q_X} - \) any subset that contains a final state, but not the start state.

2. Start state: \( z_0 = \{ \} \).
   \( \bullet \) \( \delta_Z(z_0, a) = \delta(x_0, a) \);
   \( \bullet \) \( \delta_Z(z_0, b) = \delta(x_0, b) \);

3. Final states: \( \{ z_0 \} \cup \{ \text{Every } z \in Q_Z \text{ that contains an } x \in F_X \} \).

4. \( \delta \) function. Let
\begin{itemize}
  \item \( \{ x_{i_1}, x_{i_2}, \ldots, x_{i_j} \} \in Q_Z \)
  \item \( \delta_X(x_{i_k}, a) = x'_{i_k}, \) for \( x_{i_k}, x'_{i_k} \in Q_X \) and \( a \in \Sigma. \)
\end{itemize}

Then,

\[
\delta_Z(\{ x_{i_1}, x_{i_2}, \ldots, x_{i_j} \}, a) = \begin{cases} 
\{ x'_{i_1}, x'_{i_2}, \ldots, x'_{i_j} \}, & \text{no } x' \in F_X; \\
\{ x_{i_1}', x_{i_2}', \ldots, x_{i_j}' \} \cup \{ x_0 \}, & \text{otherwise}
\end{cases}
\]

Illustrate with an FA that accepts all strings with an odd number of \( b \)'s.
Nondeterministic Finite Automata

Definition

A nondeterministic finite automaton (NFA) is a TG:

• with a unique start state

• whose transition function labels each arc with a letter in its input alphabet $\Sigma$.

Illustrate.
Theorem

For every NFA, there is an FA that accepts the same language.

Proof

1. Convert the NFA to a regular expression, using Kleene’s theorem.
2. Convert the regular expression to an FA, using Kleene’s theorem.
NFA: Theorem

Theorem
For every NFA, there is an FA that accepts the same language.

Alternate Proof

1. Given NFA $N = (Q_N, q_0_N, F_N, \delta_N)$.
2. Construct FA $A = (Q_A, q_0_A, F_A, \delta_A)$ as follows:
   
   \begin{itemize}
   
   \item $Q_A = 2^{Q_N}$.
   \item $q_0_A = q_0_N$.
   \item Let $q_A \in Q_A$. If $q_A$ contains a $q \in F_N$, then $q_A \in F_A$.

   \end{itemize}
- Let \( q_a = \{q_1, q_2, \ldots, q_j\} \in Q_N \).
- Define \( \delta_A(\{q_1, q_2, \ldots, q_j\}, a) = \{q'_1, q'_2, \ldots, q'_j\} \), where \( \delta_N(q_i, a) = q'_i \).
- \( \delta_A(\{\}, a) = \{\} \).