CMSC 330: Organization of Programming Languages

Theory of Regular Expressions
NFAs $\rightarrow$ DFAs
Review

• How are DFAs and NFAs different?

• When does an NFA accept a string?

• How do we convert from a regular expression to an NFA?

• What is the $\varepsilon$-closure of a state?
Relating R.E.'s to DFAs and NFAs

DFA can transform NFA can transform r.e.
(can transform we’ll discuss this next)
Reduction Complexity

- Regular expression to NFA reduction:
  - $O(n)$

- NFA to DFA reduction
  - Intuition: Build DFA where each DFA state represents a set of NFA states
  - How many states could there be in the DFA?
  - Given NFA with $n$ states, DFA may have $2^n$ states
  - This is not so good, but we can implement DFAs more easily
NFA → DFA reduction

Example:
NFA → DFA reduction Algorithm

• Let $r_0 = \varepsilon$-closure($q_0$), $R = \{r_0\}$, $\delta_R = \emptyset$
• While there is an unmarked state $r_i$ in $R$
  – Mark $r_i$
  – For each $a \in \Sigma$
    • Let $S = \{s \mid \{q, a, s\} \in \delta$ and $q \in r_i\}$
    • Let $E = \varepsilon$-closure($S$)
    • If $E \not\in R$, then $R = R \cup E$
    • $\delta_R = \delta_R \cup \{r_i, a, E\}$
• Let $F_R = \{r_i \mid \exists s \in r_i \text{ with } s \in q_f\}$

Notes: Let $(\Sigma, Q, q_0, F_Q, \delta)$ be the NFA and $(\Sigma, R, r_0, F_R, \delta_R)$ be the DFA. All new states in $R$ are unmarked at creation.
NFA → DFA example

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

Language?
All strings that have exactly 1 b and end in b or the string a

Regular expression?
\[ a^*b | a \]
Practice

Convert the NFA to a DFA:
Equivalence of DFAs and NFAs

- Any string from A to either D or CD in the DFA represents a path from A to D in the original NFA.
Relating R.E.'s to DFAs and NFAs

- Regular expressions, NFAs, and DFAs accept the same languages!

High-level idea next
Converting from 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 R.E’s to DFAs and NFAs

• Why do we want to convert between these?
  – Easier to express ideas (regexes/NFAs)
  – Easier to implement (DFAs)
Implementing DFAs

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");
    }
}

It's easy to build a program which mimics a DFA
Implementing DFAs (Alternative)

Alternatively, use generic table-driven DFA

given components \((\Sigma, Q, q_0, F, \delta)\) 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
Run Time of Algorithm

• Given a string $s$, how long does algorithm take to decide whether $s$ is accepted?
  – Assume we can compute $\delta(q_0, c)$ in constant time
  – Time per string $s$ to determine acceptance is $O(|s|)$
  – Can’t get much faster!

• But recall that constructing the DFA from the regular expression $A$ may take $O(2^{|A|})$ time
  – But this is usually not the case in practice

• So there’s the initial overhead, but then accepting 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 \((\Sigma, Q_A, q_A, \{f_A\}, \delta_A)\), the components of the DFA produced from the r.e.
Complement of DFA

Given a DFA accepting language L, how can we create a DFA accepting its complement?
(the alphabet = \{a,b\})
Complement Steps

- Add implicit transitions to a dead state
- Change every accepting state to a non-accepting state and every non-accepting state to an accepting state
- Note: this *only* works with DFAs - Why?
Practice

Make the DFA which accepts the complement of the language accepted by the DFA below.
Practice

Convert to a DFA:

Convert to an NFA and then to a DFA:
• $(0|1)^*11|0^*$
• strings of alternating 0 and 1
• $aba^*|(ba|b)$
Practice

• Make the DFA which accepts all strings with a substring of 330
• Take the complement of this DFA
Considering Ruby Again

- Interpreted
- Implicit declarations
- Dynamically typed
  - These three make it quick to write small programs
- Built-in regular expressions and easy string manipulation
  - This and the three above are the hallmark of scripting languages
- Object-oriented
  - Everything (!) is an object
- Code blocks
  - Easy higher-order programming!
  - Get ready for a lot more of this...
Other Scripting Languages

• Perl and Python are also popular scripting languages
  – Also are interpreted, use implicit declarations and dynamic typing, have easy string manipulation
  – Both include optional “compilation” for speed of loading/execution

• Will look fairly familiar to you after Ruby
  – Lots of the same core ideas
  – All three have their proponents and detractors
  – Use whichever one you like best