CMSC 330: Organization of Programming Languages

Theory of Regular Expressions
NFAs → DFAs

Reminders
- Homework 1 due Sep. 20
- Project 1 due Sep. 24
- Exam 1 on Sep. 25
  - Study this weekend!
- Project 2 given out on Sep. 24.
  - Start soon!

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 ε-closure of a state?

Relating R.E.'s to DFAs and NFAs

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
  - Not so good, since DFAs are what we can implement easily

NFA → DFA reduction
Example:
NFA → DFA reduction Algorithm

- Let \( r_0 \) be the \( \varepsilon \)-closure of \( q_0 \), add it to \( R \)
- While there is an unmarked state \( r_i \) in \( R \)
  - Mark \( r_i \)
  - For each \( a \in \Sigma \)
    - Let \( S = \{ s \mid q \in r_i \text{ and for } (q, a, B) \in \delta, s \in B \} \)
    - Let \( E = \varepsilon\)-closure(\( S \))
    - If \( E \notin R \)
      - \( R = E \cup R \)
    - \( \delta = \delta \cup (r_i, a, E) \)
- Let \( r_f = \{ r_i \mid \exists s \in r_i \text{ with } s \in q_f \} \)

Notes: Let \( Q \) be the set of states for the NFA and \( R \) be the set of states for the DFA. All states are unmarked at creation.

NFA → DFA example

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

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

Regular expression? \( a^* b | a \)

Relating R.E.’s to DFAs and NFAs

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

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?
  - Can make it easier to express ideas
  - Can be easier to implement

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 '
': 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 '
': printf("accepted\n"); return 1;
        default: printf("rejected\n"); return 0;
    }
    break;
    default: printf("unknown state; I'm confused\n");
    break;
    }
}
```

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 '
': 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 '
': printf("accepted\n"); return 1;
        default: printf("rejected\n"); return 0;
    }
    break;
    default: printf("unknown state; I'm confused\n");
    break;
    }
}
```

Implementing DFAs (Alternative)

Alternatively, use generic table-driven DFA

```
given components (Σ, Q, q0, F, δ) of a DFA:
set q = q0
while (there exists another symbol s of the input string):
    q := δ(q, s)
    if q ∈ F then
        accept
    else
        reject
```

- q is just an integer
- Represent δ 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 δ(q0, c) in constant time
  - Then the 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 $(Σ, Q, q0, F, δ)$, the components of the DFA produced from the r.e.
- 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

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

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

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)]

Complement of DFA

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

Practice

Make the DFA which accepts all strings with a substring of 330
- Take the complement of this DFA