CS 3100 – Models of Computation – Fall 2011 – Notes for L9

September 27, 2011

1 Regular Expressions as a Formal Notation for Regular Languages

Each regular language can be compactly described using a *regular expression.* Each regular expression can be systematically translated into an NFA. Here is a table of REs and the languages they denote. Since you'll know how to convert languages to NFA in this sequel, I dispense with drawing the NFAs for these REs—but you are requested to do this as practice! Let us assume that the alphabet for the RE definition is Σ with $a, b, c, \ldots \in \Sigma$. Let R_1 and R_2 be regular expressions, in what follows. Let $\mathcal{L}(R)$ denote the language of an RE.

R	$\mathcal{L}(R)$
$\emptyset$	$\{\}$, or in fact the same as $\emptyset$
ε	$\{\varepsilon\}$
a	$\{a\}$
R_1R_2	$\mathcal{L}(R_1)\mathcal{L}(R_2)$
$R_1 + R_2$	$\mathcal{L}(R_1) \cup \mathcal{L}(R_2)$
$R*$	$(\mathcal{L}(R))^*$
(R)	$\mathcal{L}(R)$

QUESTION FOR YOU: Why is there no complement operation or intersection operation?

1.1 Example: RE (0*1*)

(write down your class notes)

1.2 What is not in the language of RE (0*1*)?

(write down your class notes)

1.3 Example: RE (0*1*)*

(write down your class notes)

1.4 How do (0*1*)* and (0+1)* relate?

(write down your class notes)

2 General Approach to Regular Language Handling

- Obtain informal language specification, say in English
- Converse with “customer,” refine language spec
- Check if regular (is a reg. language possible at all?)
- If so write an RE
- Convert RE to an NFA
- Convert NFA to a DFA
- Minimize DFA
- Code-up DFA as a table
- You have a recognizer for the RE!
- If in doubt, extract RE from final DFA, and walk around the same loop; you must re-obtain the same DFA (may be too arduous)
- Debug the DFA (or NFA, or RE) using queries—what is *not* in the language (ask the customer); then see if the intersection is empty or not. A whole set of putative queries checks out desired and undesired properties of the language in question.
- If really a critical task (most tasks are!), obtain the RE using two lines of thinking; obtain minimal DFAs from each RE; they must be *isomorphic*!

3 Kleene-Star Construction

We now illustrate how to perform Kleene-star of an NFA.

```
Q2 = {'S0', 'S1', 'S2', 'S3', 'S4', 'S5', 'S6'}

Sigma2 = {'0','1'}

Delta2 = { ('S0', '')  : { 'S1' },
           ('S0', '0') : { 'S0' },
           ('S0', '1') : { 'S0', 'S5' },
           ('S1', '0') : { 'S2' },
           ('S2', '0') : { 'S3' },
           ('S2', '1') : { 'S3' },
           ('S3', '0') : { 'S4' },
           ('S3', '1') : { 'S4' },
           ('S5', '0') : { 'S6' },
           ('S5', '1') : { 'S6' } }

q02 = 'S0'

F2 = {'S4', 'S6'}
```

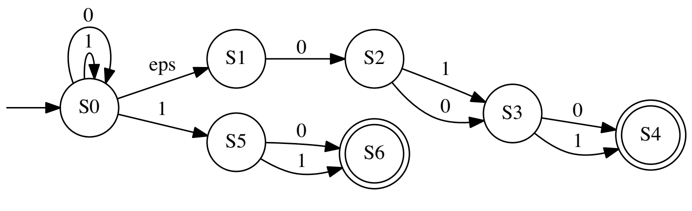

Figure 1: A simple NFA

```
NFA2 = mk_nfa(Q2, Sigma2, Delta2, q02, F2)

#--

def dot_nfa(N, fname):
    """Generate a dot file with the automaton in it. Run the dot file through
    dot and generate a ps file.
    """
    fl = open(fname, 'w')
    prDotHeader(fl)
    prNodeDefs(fl, N)
    prOrientation(fl)
    prNFAEdges(fl, N)  # <--- only difference with DFA!
    prClosing(fl)

def prNFAEdges(fl, N):
    """Suppress BH.
    """
    print(r'/* The graph itself */', file=fl)
    print(r'""  -> ', dot_san_str(N["q0"]), ";", file=fl)
    for QcQ in N["Delta"].items():
        for nxt_state in QcQ[1]:
            print(dot_san_str(QcQ[0][0]), r' -> ', dot_san_str(nxt_state),
                r'[label="', dot_san_str(ShowEps(QcQ[0][1])), r'"];', file=fl)
```

With the above definitions, we call `dot_nfa(NFA2,"NFA2.dot")` and print it, to obtain

- What is the language of this NFA?
- Test this NFA using the `accepts_nfa` or the `accepts_nfav` function on various strings.

4 Kleene-star of the NFA of Figure 1

To Kleene-star the NFA of Figure 1, we just need to do these steps:

- Introduce a new start state that is also a final state (we call this state `NewInit`)

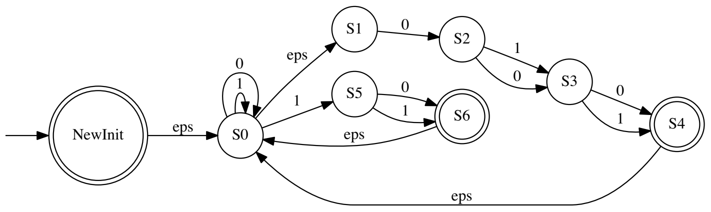

Figure 2: The Star of the simple NFA in Figure 1

- Jump via a ε transition to the *original start state*
- Connect all the final states of the NFA via ε transitions to the original start state of the NFA (in an earlier version of this figure, I had added transitions to the new new initial state `NewInit`; that is also OK; but let us stick with this construction which says "connect the final states back to the original start state").
- The original final states continue to be final states. The only additional final state that came in is `NewInit`, which happens also to be the new initial state.

Let us do the above steps in Python. Things can't be neater than this!

```
def starAnNFA(N):
    """Given an NFA N, return an NFA whose language is the Kleene-star
    of the NFA N. The construction is the standard textbook method of
    adding a fresh initial+final state that transitions to the original
    initial state, and a loop from every final state back to this new
    initial+final state. All these new transitions are epsilon transitions.
    """
    NQ = N["Q"] | set({"NewInit"})
    Nq0= "NewInit"
    NF = N["F"] | set({"NewInit"})
    #
    add_delta1 = { (q,"") : set({ N["q0"] }) for q in N["F"] }
    #
    add_delta1.update({ ("NewInit", "") : set({N["q0"]}) })
    #
    add_delta1.update(N["Delta"])
    #
    return(mk_nfa(NQ, N["Sigma"], add_delta1, Nq0, NF))
```

Then we do:

```
dot_nfa(starAnNFA(NFA2),"NFA2star.dot")
```

After converting the `.dot` file over, we obtain the starred NFA in Figure 2.

5 Algorithms for Various Operations on NFA and DFA

Please note that some algorithms do not apply to NFA. All algorithms work on DFA because DFA are a special case of NFA (but the result will be an NFA). Many of these sections are discussed in my textbook “Computational Engineering: Applied Automata Theory and Logic” published by Springer in 2006. What follows contain excerpts from my book’s draft, but with Python codes added.

5.1 NFA to DFA Conversion

```
              NFA                                  DFA
              ===                                  ===

     Input   0     1     e          Input   0       1
      --------------------            ------------------
States |                         States  |
  I    |  {I,A}  { I }  { }         {I}  |  {I,A}    { I }
       |                                 |
  A    |  { B }  { B }  { }       {I,A}  |  {I,A,B}  {I,B}
       |                                 |
  B    |  { F }  { F }  { }       {I,B}  |  {I,A,F}  {I,F}
       |                                 |
  F    |   { }    { }  { I }    {I,A,B}  | {I,A,B,F} {I,B,F}
                                         |
                              {I,A,B,F}  | {I,A,B,F} {I,B,F}
                                         |
                                {I,B,F}  |  {I,A,F}  {I,F}
                                         |
                                {I,A,F}  |  {I,A,B}  {I,B}
                                         |
                                  {I,F}  |   {I,A}    {I}
                                         |
```

Figure 3: NFA to DFA conversion illustrated on an example NFA

Given an NFA $N = (Q, \Sigma, \delta, q_0, F)$, a DFA that is language-equivalent to N is given by

$$D = (2^Q, \Sigma, \delta_D, Eclosure(q_0), \{f \mid f \in 2^Q \wedge f \cap F \neq \emptyset\}).$$

Note that **the concept of** $Eclosure(q_0)$ **is discussed in `notes7.pdf`.** This DFA D is designed to mimic the acceptance behavior of the NFA N as follows:

- N can have its tokens in (potentially) any combination of its states. Therefore, the states of D are chosen to be 2^Q, the powerset of the states of N.
- When we place a token in the initial state of N, it will spread (through *Eclosure*) to all states reachable through ε moves. To model this, D’s starting state is chosen to be $Eclosure(q_0)$.
- N accepts a string when one of its tokens reaches a state in F. To model this, the final states of D are chosen to be those subsets of 2^Q that include a state of F.

- The last (and crucial) detail is δ_D:

$$\delta_D(S, a) = \{y \mid \exists s \in S \, : \, y \in Eclosure(\delta(Eclosure(s), a))\}.$$

  In other words, δ_D takes a state S of D, first *Ecloses* all the states it contains, runs δ of the NFA from each resulting state, and finally *Ecloses* and unions the results.

Strictly speaking, once the algorithm starts, for the intermediate steps we don't need *Eclosure* to be performed before *and* after; it is obvious that doing an *Eclosure* from a state resulting from a previous *Eclosure* does not change things. However, it is convenient (and entirely equivalent) to pad each "step" with *Eclosure* performed both before and after. These "super-steps" are what the DFA engages in.

Another way to define δ_D is through the $\hat{\delta}$ (string transition) function of the NFA which, in effect, does the same thing as the above definition:

$$\delta_D(S, a) = \{y \mid \exists s \in S \, : \, y \in \hat{\delta}(s, a)\}.$$

(See the definition of $\hat{\delta}$ in `notes7.pdf`.)

By construction, whenever N accepts a string w, D also accepts w, and vice versa. Since NFAs can be converted to DFAs, the language of an NFA is also *regular*. Also, D can potentially have $2^{|Q|}$ states.

Illustration 5.1 Let us follow the illustration in Figure 3. We succinctly demonstrate the conversion by listing their state transition tables side by side as below. *We introduce only those states of 2^Q that are reachable.* A systematic way to proceed is now illustrated with respect to an example. Suppose the DFA is in state `{I,A,B}`. Suppose the input be `0`. Then we walk through the column `0` in the NFA table, union what we see against `I` with what we see against `A`, and what we see against `B`. These sets are, respectively, `{I,A}`, `{B}`, and `{F}`. The union of these sets is `{I,A,B,F}`. Now, we perform the *Eclosure* step, which applies only to state `F` which has an ε move to state `I`. But since `I` is already in `{I,A,B,F}`, the final result is `{I,A,B,F}`. We proceed to do the same for every other state of the DFA.

5.1.1 Python Code for Conversion

```
#-- nfa2dfa --
# FrontQ,     Qsofar,             Delta
# List of SS, List of SS so far,  Function
#
# Initially, Qsofar == set()
#
# For each c in Sigma,
#   move each item in FrontQ through c
#   see if any outside Qsofar; add those SS to FrontQ and fill Delta with move to SS
#   until all c generate fall into FrontQ
#   Export Q=Qsofar, Sigma, Delta, q0=Init, F = map over Qsofar picking out F in sets

def nfa2dfa(N):
    EC = eclosure({N["q0"]}, N)
    # Qsofar is set to [EC] because we have discovered the 'in's to EC; the outs will
    # be added while returning
    return n2d([EC], [EC], dict({}), N)

def n2d(FrontQ, Qsofar, Delta, N):
    # was the idea: unfortun. double call to ec_step_nfa; alt is using Qn but then iter over powerset?
    # NewMoves = [ ((Q, c), ec_step_nfa(Q, c, N)) for Q in FrontQ for c in N["Sigma"] if ec_step_nfa(Q, c, N) not in Qsofar ]
    AllMoves = [ ((Q, c), ec_step_nfa(Q, c, N)) for Q in FrontQ for c in N["Sigma"] ]
    # print("AllMoves")
    # print(AllMoves)
    NewMoves = list(filter(lambda QcQ: QcQ[1] not in Qsofar, AllMoves)) # [1] picks out c of ((a,b),c)
    # print("NewMoves")
    # print(NewMoves)
    if NewMoves == []:
        # print("Must still add last bit to Delta to discover last bit out OUTs")
        AllMovesDelta = dict([ ( (str(Qfrom), c), str(Qto) ) for ((Qfrom, c), Qto) in AllMoves ])
        Delta.update(AllMovesDelta)
        #
        DFA_Q = { str(Q) for Q in Qsofar }
        DFA_Sigma = N["Sigma"]
        DFA_Delta = Delta
        DFA_q0 = str(eclosure({N["q0"]}, N))
        DFA_F = set(map(lambda Q: str(Q), filter(lambda Q: (N["F"] & Q) != set({}), Qsofar)))
        #
        # print("DFA_Q")
        # print(DFA_Q)
        # print("DFA_Sigma")
        # print(DFA_Sigma)
        # print("DFA_Delta")
        # print(DFA_Delta)
        # print("DFA_q0")
        # print(DFA_q0)
        # print("DFA_F")
        # print(DFA_F)
        return mk_dfa(DFA_Q, DFA_Sigma, DFA_Delta, DFA_q0, DFA_F)
    else:
        newFrontQ = list(map(lambda QcQ: QcQ[1], NewMoves)) # picks out c of ((a,b),c)
        newQsofar = Qsofar + newFrontQ
        # was NewMoves; changed to AllMoves below because even the jumps back need to be added
        NewMovesDelta = dict([ ( (str(Qfrom), c), str(Qto) ) for ((Qfrom, c), Qto) in AllMoves ])
        Delta.update(NewMovesDelta)
        return n2d(newFrontQ, newQsofar, Delta, N)
```

The result of applying the NFA to DFA conversion for the NFA of Figure 1 is shown in Figure 4.

6 Other Automaton Conversions

We now take operations one at a time and illustrate then on regular machinery (regular machinery includes NFA and DFA, but later also *regular expressions*, or RE).

Under each operation, we comment on the relative difficulty of performing the operation directly on a certain machine type. More importantly, we point out those algorithms that are *incorrect* to carry over to the 'wrong' machine type. The operations will be illustrated on DFA D_1 and D_2, NFA N_1 and N_2, and REs R_1 and R_2, as the case may be. For the sake of notational uniformity, we employ alphabet names Σ_1 and Σ_2, where in fact $\Sigma_1 = \Sigma_2 = \Sigma$. We also assume that $Q_1 \cap Q_2 = \emptyset$. Specifically, we use

> $D_1 = (Q_1, \Sigma_1, \delta_1, q_0^1, F_1)$, and $D_2 = (Q_2, \Sigma_2, \delta_2, q_0^2, F_2)$, for binary operations, and D_1 alone for unary operations.

Similarly, we will employ NFA

> $N_1 = (Q_1, \Sigma_1, \delta_1, q_0^1, F_1)$ and $N_2 = (Q_2, \Sigma_2, \delta_2, q_0^2, F_2)$

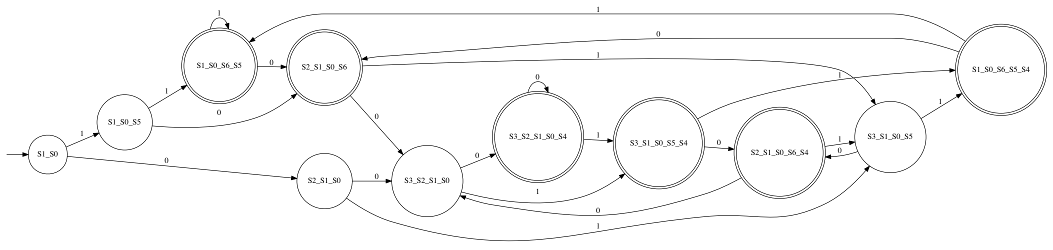

Figure 4: NFA to DFA conversion applied to the NFA of Figure 1

The results will be presented in terms of

$D = (Q, \Sigma, \delta, q_0, F)$ for DFA, $N = (Q, \Sigma, \delta, q_0, F)$ for NFA, and R for regular expressions.

In some cases, the operation changes the type of the machine, and if this happens we will explicitly point it out. For example, concatenation of two DFA can be directly accomplished if we treat the DFA as NFA and concatenate them; the result will then be an NFA. We show the results for REs along with NFAs. Through minor notational abuse, we also will call the resulting machines $\overline{D_1}$ for the complement of DFA D_1, $D_1 \cup D_2$ for the union of two DFAs, etc.

6.1 Complementing a DFA (does not work for an NFA)

This operation is most easily performed on DFA, resulting in

$$D = \overline{D_1} = (Q_1, \Sigma_1, \delta_1, q_0^1, Q_1 \setminus F_1).$$

As shown in class, this does not work work work for NFAs!

6.2 Union of two DFA (does not work for NFA)

Given DFAs D_1 and D_2, a DFA D that models the marching of D_1 and D_2 in lock-step, is constructed. Basically, the initial states of D_1 and D_2 are paired, and made the initial state of D. Next, for each symbol $a \in \Sigma$, the resulting next states of the individual DFAs are paired, and the pair is regarded as the next state of D. Any paired state where one of the DFAs is in one of its final states is considered a final state of D.

$D = D_1 \cup D_2 = (Q_1 \times Q_2,\ \Sigma,\ \delta,\ \langle q_0^1, q_0^2 \rangle,\ (Q_1 \times F_2 \cup F_1 \times Q_2))$, where
$\delta(\langle x, y \rangle, a) = \langle \delta_1(x, a), \delta_2(y, a) \rangle$.

Again, note that this algorithm is different from that for NFAs.

6.3 Intersection of two DFA (does not work for NFA)

The only difference with DFA union of § 6.2 is that the final set of states becomes $F_1 \times F_2$.

6.4 Intersection of two NFAs is hard (no direct algorithm)

If NFA intersection were easy, then NFA complementation would also be easy (because NFA union is easy, and using union and complementation we can do intersection, using the DeMorgan's Law). Thus, there can't be an easy algorithm for NFA intersection.

One has to convert an NFA to a DFA and then perform intersection. This can, of course, incur an exponential cost, as the number of states after such conversion may be exponential.

6.5 Union of two NFA (applies to DFA but results in NFA)

We have seen this in `notes7.pdf`. In symbols:

$N = N_1 \cup N_2 = (Q_1 \cup Q_2 \cup \{I_0\}, \Sigma, \delta, I_0, F_1 \cup F_2)$, where
$\delta = \delta_1 \cup \delta_2 \cup \{\langle I_0, \varepsilon, \{q_0^1, q_0^2\}\rangle\}$.

As can be seen, with NFA and RE, union is a linear-time operation, while with a DFA, it is an $O(N^2)$ operation.

6.6 Concatenation of two NFA (applies to DFA but results in NFA)

We have seen this in `notes7.pdf`. In symbols:

$N = N_1\ N_2 = (Q_1 \cup Q_2, \Sigma, \delta, q_0^1, F_2)$, where
$\delta = \delta_1 \cup \delta_2 \cup \{\langle x, \varepsilon, \{q_0^2\}\rangle \mid x \in F_1\}$.

6.7 Star of two NFA (applies to DFA but results in NFA)

We have seen this in `notes7.pdf` and also earlier in these notes. With NFA, we introduce a new start state I_0, regard this state also as a final state (thus introducing ε into the language of N^*), and introduce a transition on ε from I_0 to q_0. Finally, to introduce iteration, we introduce a transition from every state in F to q_0. In symbols:

$N = N_1^* = (Q_1 \cup \{I_0\}, \Sigma_1, \delta, I_0, F_1 \cup \{I_0\})$, where
$\delta = \delta_1 \cup \{\langle I_0, \varepsilon, \{q_0^1\}\rangle\} \cup \{\langle x, \varepsilon, \{q_0^1\}\rangle \mid x \in F_1\}$.

6.8 Reversal of an NFA (applies to DFA but results in NFA)

With NFA, to do reversal, we introduce a new state I_0, make that the initial state, and introduce a transition on ε from I_0 to the set of states F_1. We also reverse every edge in the NFA diagram, and regard $\{q_0^1\}$ as the set of final states. In symbols:

$N = rev(N_1) = (Q_1 \cup \{I_0\}, \Sigma_1, \delta, I_0, \{q_0^1\})$, where
$\delta = \{\langle I_0, \varepsilon, F_1\rangle\} \cup$
$\quad \{\langle r, a, \{q \mid q \in Q_1 \ \wedge\ r \in \delta(q, a)\}\rangle \mid r \in Q_1 \ \wedge\ a \in \Sigma_1 \cup \{\varepsilon\}\}$.

The reason why δ looks so complicated is because we try to reverse the transitions that, for an NFA, are specified via its delta function as moves from a state to a *set of next states*. We must also preserve the 'state to set of states' nature of the mapping for the δ of the reversed machine. In the δ for the reversed machine, we introduce a move from every state $r \in Q_1$ for every $a \in \Sigma_1 \cup \{\varepsilon\}$ to all those states q, such that those q states had at least one 'forward' transition through a to r.