CS 3100 – Models of Computation – Fall 2011 – Notes for L5

September 15, 2011

1 On DFA

A *deterministic finite-state automaton* (DFA) models finite-state devices. There are many real-life systems that can be modeled using DFA; some examples are:

- C programs operating with finite memory,
- Scanners that check the syntax of tokens in a language (*e.g.*, strings, floating-point numbers, telephone numbers, etc.)
- Almost all board-games
- Binary decision diagrams (we will study them much later)

We will mostly study DFAs at an abstract level. We will then study how to express DFAs using regular expressions, and automatically build them using tools such as `ply`.

DFA are specified through a transition graph such as in Figure 1. It starts its operation in an initial state (in this case `I`). When a DFA is situated in one of its reachable states s and is fed an input string w, it advances to a *unique* state as per its transition graph. Formally, a *deterministic finite-state automaton* D is described by five quantities presented as a tuple, $(Q, \Sigma, \delta, q_0, F)$, where:

- Q, a *finite nonempty* set of states;
- Σ, a *finite nonempty* alphabet;
- $\delta : Q \times \Sigma \rightarrow Q$, a *total* transition function;
- $q_0 \in Q$, an initial state; and
- $F \subseteq Q$, a *finite, possibly empty* set of final (or *accepting*) states.

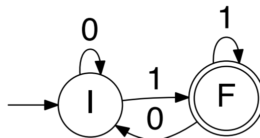

Figure 1: A DFA that recognizes strings over $\{0, 1\}$ ending with 1

```
/* Encoding of a DFA that accepts the language (0+1)*1 */
main()
{ bool state=0;
  char ch;
  while(1)
  { ch = getch();
    switch (state)
    { case I: switch-off the green light;
              switch (ch)
              { case 0 : break;
                case 1 : state=F;
              }
      case F: switch-on the green light;
              switch (ch)
              { case 0 : state=I;
                case 1 : break;
              }
    }
  }
}
```

Figure 2: Pseudocode for the DFA of Figure 1

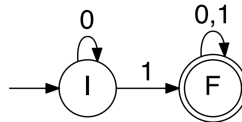

Figure 3: Multiple symbols labeling a transition in lieu of multiple transitions

For example, in Figure 1, $Q = \{I, F\}$, $\Sigma = \{0, 1\}$, $q_0 = I$, $F = \{F\}$, and δ is a total function that maps transitions as per this figure, *i.e.*,

```
{('I',0) : 'I', ('I',1) : 'F', ('F',0) : 'I', ('F',1) : 'F'}
```

Sometimes, it is tedious to draw out δ as a total function because most of the moves are to the "black hole" (BH) state: states into which transitions can check in, but not check out[1] In that case, please add documentation that "moves to the BH state are not shown" or something to that effect.

One could have a DFA with multiple symbols labeling a single transition in lieu of separate transitions that bear these symbols, as shown in Figure 3. One could have a DFA with truly unreachable states, as shown in Figure 4. These states (states X and Y in this example) may be removed without any loss of meaning.

The DFA in Figure 1 is really encoding programs similar to the one in Figure 2. All "DFA programs" are `while(1)` loop controlled `switch` structures. While in a certain state, a DFA decodes the current input symbol and decides to either update its state or keep its current state (a DFA always moves; it may decide to move back to its current state). The next input symbol is decoded in this (possibly) updated state. Whenever *any*

[1] Also called RM or Roach Motel states.

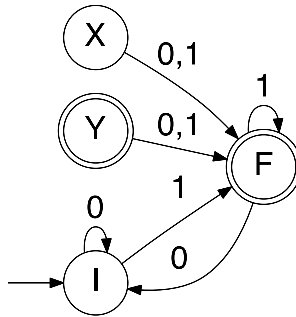

Figure 4: A DFA with unreachable (redundant) states

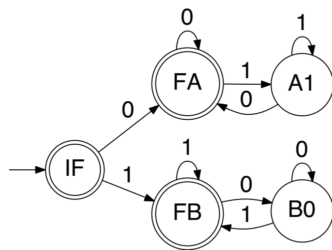

Figure 5: Another example DFA

final state (including the IF state) is entered, the DFA turns on its *green light.* The green light stays on so long as the DFA stays in one of the final state, and is turned off when it exits it. A DFA program is not allowed to use *any* other variable than the single variable called `state`. It cannot produce *any* other output than shine the green light. It appears to be programming at its simplest - yet, such humble DFAs are enormously powerful and versatile!

The δ function of a DFA is also commonly presented as a table, with rows being states, columns being inputs, with the final state(s) starred. For our example, we have

```
       input
       -----
state| 0   1
------------
 I   | I   F
*F   | I   F
```

A DFA is said to accept a string s if its green-light is 'on' after processing that string; all other strings are *rejected* by the DFA. However, note that a DFA "never stops working"; after processing any string s, it will be in a position to process any one of the input symbols a, and may (depending on the DFA) accept or reject that string, sa. DFAs can be thought of as *string classifiers*, assigning a status (accepted/rejected) for *every* string within Σ^*. The set of strings accepted by a DFA is said to be its *language.* A DFA is said to *recognize* its language (not *accept* its language - the term 'accept' is used for individual strings). Formally speaking, a DFA is said to *accept* an input $x \in \Sigma^*$ if x takes it from q_0 to one of its final states. These ideas are captured by the following Python programs:

```
# Some helper functions, and Dfa operations

def fst(p):
    """ First of a pair."""
    return p[0]

def snd(p):
    """ Second of a pair."""
    return p[1]

def fn_dom(F):
    """ For functions represented as hash-maps (dicts), return their domain as a set.
    """
    return {k for k in F.keys()}

def fn_range(F):
    """ For functions represented as hash-maps (dicts), return their range as a set.
    """
    return {v for v in F.values()}

# More checks are possible perhaps... but here is most
def mk_dfa(Q, Sigma, Delta, q0, F):
    """Make a DFA with the given traits. Delta is supplied as a hash-map (dict).
    """
    assert(Sigma != {})
    #
    # Sigma is a set of strings of length 1. This check does it in one line
    assert(not(False in list(map(lambda x: len(x)==1, Sigma))))
    #
    # Delta must be a total function
    dom = fn_dom(Delta)
```

```
    states_dom = set(map(fst,dom))
    input_dom = set(map(snd,dom))
    state_targ = set(fn_range(Delta))
    #-- num state and input entries to match
    assert(states_dom == Q)
    assert(input_dom == Sigma)
    #
    #-- Mapping for every pair must be present
    assert(len(Delta)==len(Q)*len(Sigma))
    #
    # Targets must be in Q and non-empty
    assert((state_targ <= Q)&(state_targ != {}))
    # Initial state in Q
    assert(q0 in Q)
    # Final states subset of Q (could be empty, could be Q)
    assert(set(F) <= Q)
    # If all OK, return DFA as a dict
    return({"Q":Q, "Sigma":Sigma, "Delta":Delta, "q0":q0, "F":F})

# An example DFA being made

Q1 = {'S0','S1'}

Sigma1 = {'a','b'}

Delta1 = {('S0', 'a'): 'S0', ('S1', 'a'): 'S0', ('S1', 'b'): 'S1', ('S0', 'b'): 'S1'}

q01 = 'S0'

F1 = {'S1'}

DFA1 = mk_dfa(Q1,Sigma1,Delta1,q01,F1)

# Check that it's made properly

assert(DFA1 == {'Q': {'S1', 'S0'}, 'q0': 'S0', 'F': {'S1'}, 'Sigma': {'a', 'b'}, 'Delta': {('S0', 'a'): 'S0', ('S1', 'a'): 'S0',

def mktot(D):
    """ Given a partially specified DFA, make it total by transitioning to state BH wherever undefined.
    """
    add_delta = { (q,c) : "BH" for q in D["Q"] for c in D["Sigma"] if (q,c) not in D["Delta"] }
    #
    # print("<add_delta")
    # print(add_delta)
    # print("add_delta>")
    #
    bh_moves =  { ("BH", c): "BH" for c in D["Sigma"] }
    #
    add_delta.update(bh_moves)
    #
    # print(add_delta)
    #
    add_delta.update(D["Delta"])
    #
    return {"Q": D["Q"] | { "BH" }, "Sigma": D["Sigma"], "q0": D["q0"], "F": D["F"], "Delta": add_delta}

assert(mktot(DFA1) == {'Q': {'S1', 'S0', 'BH'}, 'q0': 'S0', 'Delta': {('S0', 'a'): 'S0', ('BH', 'a'): 'BH', ('S1', 'a'): 'S0', (
```

```
assert(mktot(mktot(DFA1)) == mktot(DFA1))


def prdfa(D):
        """Prints the DFA neatly.
        """
        Dt = mktot(D)
        # print(totdfa)
        print("")
        print("Q:", Dt["Q"])
        print("Sigma:", Dt["Sigma"])
        print("q0:", Dt["q0"])
        print("F:", Dt["F"])
        print("Delta:")
        print("\t".join(map(str,Dt["Q"])))
        print("------------------------------------------------------------------------------------------")
        for c in (Dt["Sigma"]):
            nxt_qs = [Dt["Delta"][(q, c)] for q in Dt["Q"]]
            print("\t".join(map(str, nxt_qs)) + "\t" + c)
            print("")


def prdfa_nomktot(D):
        """Prints the DFA neatly. Don't make it total, as we suspect a total one is given...
        """
        print("")
        print("Q:", D["Q"])
        print("Sigma:", D["Sigma"])
        print("q0:", D["q0"])
        print("F:", D["F"])
        print("Delta:")
        print("\t".join(map(str,D["Q"])))
        print("------------------------------------------------------------------------------------------")
        for c in (D["Sigma"]):
            nxt_qs = [D["Delta"][(q, c)] for q in D["Q"]]
            print("\t".join(map(str, nxt_qs)) + "\t" + c)
            print("")

# Some ideas for printing obtained from Andrew Badr's pretty printer in his DFA package
# I've made it so that we can pretty-print NFA also (so undefined delta moves are OK)
#
def pr_nobh(D):
        """Prints the DFA nicely - don't list transitions to/from black-holes, i.e. BH
        """
        Dt = mktot(D)
        # print(totdfa)
        print("")
        print("Q:", Dt["Q"])
        print("Sigma:", Dt["Sigma"])
        print("q0:", Dt["q0"])
        print("F:", Dt["F"])
        print("Delta:")
        print("\t".join(map(str,Dt["Q"])))
        print("------------------------------------------------------------------------------------------")
        for c in (Dt["Sigma"]):
            nxt_qs = [Dt["Delta"][(q, c)] for q in Dt["Q"] ]
```

```
            print("\t".join(map(str, nxt_qs)) + "\t" + c)
            print("")


def step_dfa(D, q, c):
    """Run DFA D from state q on character c. Return the next state.
    """
    assert(c in D["Sigma"])
    assert(q in D["Q"])
    return D["Delta"][(q,c)]

def run_dfa(D, q, s):
    """Run DFA D from state q on string S. Return the next state. We run DFAs to run on "" also (empty).
    """
    # Don't run DFAs on empty strings
    return q if s=="" else run_dfa(D, step_dfa(D, q, s[0]), s[1:])

def accepts(D, q, s):
    """ Checks for DFA acceptance.
    """
    return run_dfa(D, q, s) in D["F"]
```

Drill Problem: Define the Python function `recognizes(D, n)` that returns all strings of length $\leq$ `n` recognized by the DFA `D`. *Hint:* Can you use `nthnumeric` to test candidate strings? How?

Drill Problem: What is wrong with trying to define a different function `recognizes(D)` that returns all strings of all lengths recognized by the DFA `D`? What is a test you can subject `D` to such that you can define such a function for all the `D` that passes the test?

> **REGULAR LANGUAGE:** A language L is defined to be *regular* if there is a DFA D such that $L = \mathcal{L}(D)$.

If L is the language of a DFA D, then when the DFA is run on any string $s \in L$, it leaves the DFA in one of its accepting states.

Any string **not** in the language of the DFA D is a language in the complement of the language L. Thus, this string is still over the same alphabet but leaves the DFA in a non-final state.

2 Examples of DFA

Unless you design many DFAs "by hand," you will not learn to appreciate how to design them. That is precisely what we are about to embark on.

1. Define, using set comprehension, the regular language recognized by the DFA of Figure 5.
2. For all languages defined in `notes3.pdf` that are regular languages, design a DFA.
3. Define a DFA that accepts all strings over {0,1} fed LSB-first such that these strings when interpreted according to standard binary conventions defines numbers which are evenly divisible by 3.
4. Define a DFA that accepts all strings over {0,1} such that every block of five consecutive positions contains at least two 0s.
5. Define a DFA for the language defined by the concatenation of the languages denoted by DFA of Figures 1 and 5.