CS 3100 – Models of Computation – Fall 2011 – Notes for L11

September 29, 2011

1 RE Mobius

You are a 2D ant forced to live on a mobius strip. You can walk along the strip, making round after round after round. First you see "NFA" written; then you see a fork on the road:

- One path says `nfa2dfa` and then "DFA" written. You know how this path works.
- The other path on the fork says `nfa2re`. You are yet to learn this, but see § 3.

You take the `nfa2dfa` path and then immediately see a marker saying `mindfa`. You don't know how DFA minimization works (you are yet to learn it, but see § 2).

Then you see the path `re2nfa`. Unfortunately, parsing REs requires context-free parsing (the syntax of legal REs is *not regular*). You are yet to learn how to use `ply` to do it. This will be taught much later. So now we have two algorithms to learn, described in § 3 and § 2. Then we will finish up `notes9.pdf` and then do problem after problem.

2 Minimizing DFA

The most important result with regard to DFA minimization is the *Myhill-Nerode Theorem.*

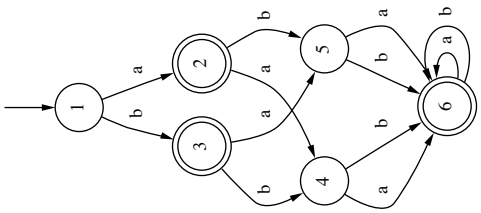
> The result of minimizing DFAs is unique, up to isomorphism.[1]

The theorem says that given two DFAs over the same alphabet that are language-equivalent, they will result in identical DFAs when minimized, up to the renaming of states.

The basic idea behind DFA state minimization is to consider all pairs of states systematically by constructing a table. For each pair of states, we consider all strings of length zero and up, and see if they can distinguish any pair of states. Initially, we distinguish all pairs of states $\langle p, q\rangle$ such that p is a final state and q is nonfinal. We enter an `x` in the table to record that these states are ε-distinguishable. In essence, at the beginning of the algorithm we are treating all final states as belonging to one equivalence class, and all non-final states as belonging to another.

Thereafter, in the ith iteration of table filling, we see if any of the state pairs $\langle p, q\rangle$ that are not yet distinguished have a move on some $a \in \Sigma$ such that they go to states $\langle p', q'\rangle$ that are distinguishable (at the end of the $i-1$st iteration); if so, distinguish $\langle p, q\rangle$. The algorithm stops when two iterations k and $k+1$ result in the same table.

[1] The concept of isomorphism comes from graph theory. Two directed graphs G_1 and G_2 are isomorphic if there is a bijection b between their nodes that preserves the graph connectivity structure. In other words, if n_1 and n_2 are nodes of G_1 and G_2, respectively, and if the bijection relates n_1 and n_2, then the list of successors of n_1 in G_1 are also bijective with the list of successors of n_2 in G_2.




```
      2 .
      3 .  .
      4 .  .  .
      5 .  .  .  .
      6 .  .  .  .  .
        1  2  3  4  5
```

```
2 x            2 x            2 x            2 x
3 x  .         3 x  .         3 x  .         3 x  .
4 .  x  x      4 .  x  x      4 x  x  x      4 x  x  x
5 .  x  x  .   5 .  x  x  .   5 x  x  x  .   5 x  x  x  .
6 x  .  .  x  x 6 x  x  x  x  x 6 x  x  x  x  x 6 x  x  x  x  x
  1  2  3  4  5   1  2  3  4  5   1  2  3  4  5   1  2  3  4  5

    0-dist          1-dist          2-dist      3-dist? No change!
                                                So, done.
```

Figure 1: Example for DFA minimization

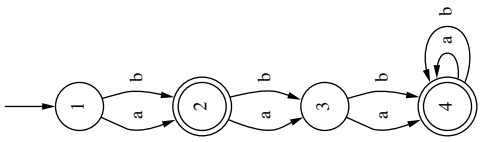


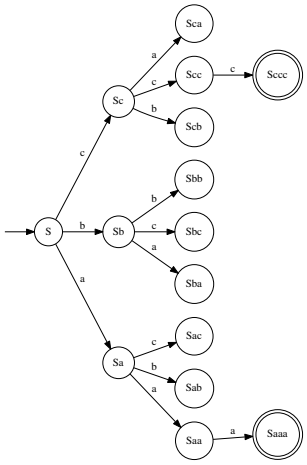
Illustration: Consider the DFA in Figure 1 (adapted from Kozen's book on Automata Theory) where the final states are 2, 3, 6, and $\Sigma = \{a, b\}$.

1. The initial blank table that permits all pairs of states to be compared is in Figure 1.
2. All 0-length string distinguishable states are all pairs of states that consist of *exactly* one accept state (see below). All subsequent steps identifying i-distinguishable states for all i are also in Figure 1.
3. Let us understand how one `x` was added. In Figure 1, we put an `x` (to distinguish between) states `2` and `6`. Why is this so? This is because `2` has a move upon input `a` to state `4`, while state `6` moves upon `a` to state `6` itself. From the `0-dist` table, we know that states `4` and `6` are distinguishable: one being a final and the other being a nonfinal state.
4. At the `3-dist` step, there was no change from the previous table. At this point, state pairs `3,2` and `5,4` still have a '.' connecting them (they could not be distinguished). Therefore, we merge these states, resulting in the minimized DFA of Figure 1.

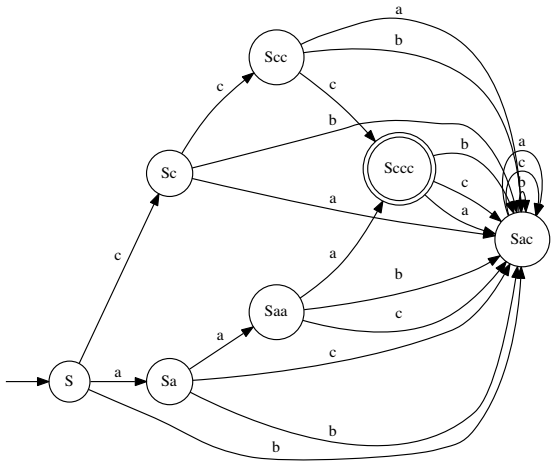
2.1 Example: Tree3

I've written the code for this conversion and produced these machines.

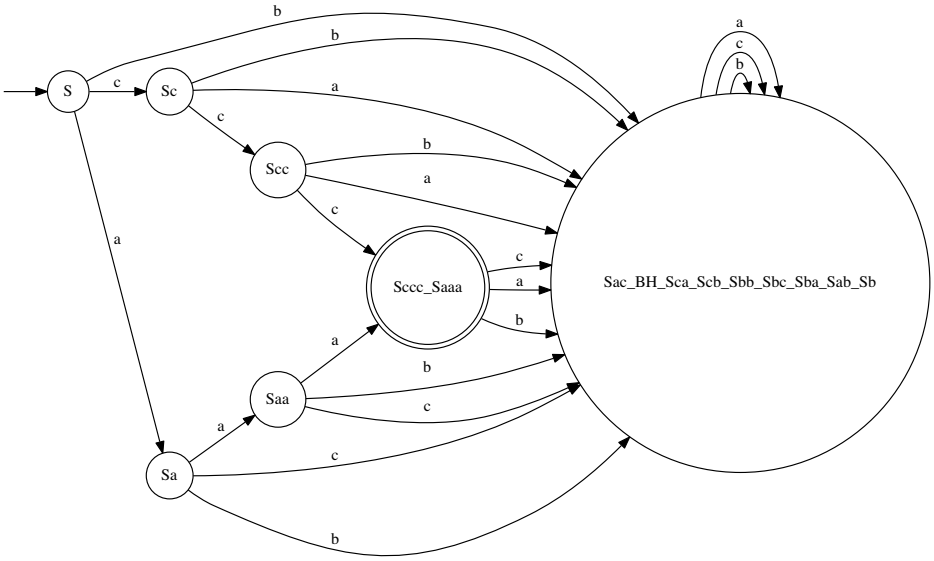
2.1.1 Initial DFA without BH states shown



2.1.2 Minimized DFA Showing Representative States

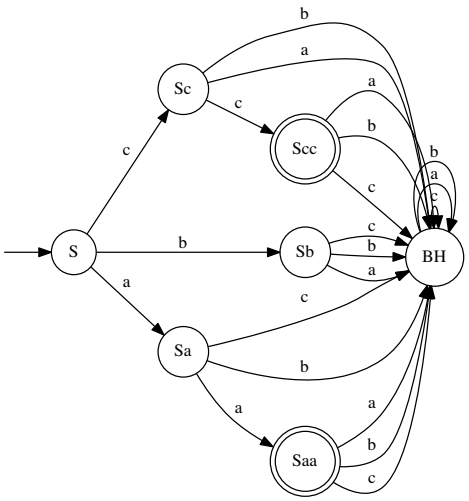


2.1.3 Minimized DFA Showing State Equivalence-class Named States



2.2 Example: Tree2

2.2.1 Initial DFA with BH states shown



2.2.2 Minimized DFA Showing Representative States

(let's do this in class)

2.2.3 Minimized DFA Showing State Equivalence-class Named States

(let's do this in class)

3 NFA to REs

We introduce algorithms that operate on 'regular machinery,' meaning various representations of regular sets.

3.1 NFA to RE

Given NFA $N = (Q, \Sigma, \delta, q_0, F)$, we can convert it to a regular expression by successively eliminating its states. This is often called the *generalized NFA* or GNFA approach, where we build GNFA whose transitions are labeled using regular expressions, instead of members of Σ_ε.

- Preprocess N by adding one new initial state B and one new final state E. In the following steps, we will eliminate every state of N, leaving only B and E behind. The transition connecting B and E will be labeled with the desired RE upon termination of our algorithm.

- Make all states F nonfinal, and introduce a transition from every state in F to E via ε.
- Introduce an ε transition from B to q_0.
- Repeatedly eliminate a state from Q. In all the steps below, whenever any pair of states p and q has two transitions going from p to q, labeled with regular expressions R_1 and R_2, replace them by a single transition labeled $R_1 + R_2$.
- Suppose
  - state p has a transition into s labeled with RE R_{ps},
  - state s has a transition to itself labeled R_{ss}, and
  - state s has a transition out of it to state q labeled R_{sq}.

  Then, we can eliminate the ps and sq transitions, and introduce a direct transition from p to q labeled $R_{ps}(R_{ss})^*R_{sq}$. We keep repeating these steps until state s is disconnected from the rest of the graph, at which point, it can be eliminated.

We now illustrate the NFA to RE conversion algorithm. Many excerpts from my textbook "Computation Engineering: Applied Automata Theory and Logic," Springer 2006 now follow.

10.2.9 Prefix-closure

This operation is straightforward to perform either on an NFA or on a DFA. We simply turn every state along the way from the start state towards one of the final states into a final state.

10.3 More Conversions

In this section, we define additional interconversions between REs, NFAs, and DFAs, thus establishing the equivalence of their expressive power.

10.3.1 RE to NFA

We first specify how to convert the *basic* regular expressions $\emptyset$, ε, and $a \in \Sigma$ into an NFA. For all other REs, we can recursively convert their constituent basic REs into NFA and then apply the corresponding NFA-building operator. The conversion of basic REs goes as follows:

- $\emptyset$ is an RE denoting $\emptyset$ The corresponding NFA is
$$N = (\{q^{\emptyset}\}, \Sigma, \emptyset, q^{\emptyset}, \emptyset),$$
where $q^{\emptyset}$ is the only state of the NFA. The transition function is $\emptyset$, i.e., has no moves.
- ε is an RE denoting $\{\varepsilon\}$. The corresponding NFA is
$$N = (\{q^{\varepsilon}\}, \Sigma, \emptyset, q^{\varepsilon}, \{q^{\varepsilon}\}),$$
where q^{ε} is the only state of the NFA that also happens to be a final state.
- $a \in \Sigma$ is an RE denoting $\{a\}$. The corresponding NFA is
$$N = (\{q_0^a, q_F^a\}, \Sigma, \delta^a, q_0^a, \{q_F^a\}),$$
where δ is $\{\langle q_0^a, a, \{q_F^a\}\rangle\}$.
- The NFA for $r_1\ r_2$, $r_1 + r_2$, and r_1^* are obtained by first obtaining the NFA for r_1 and r_2, and applying, respectively, the algorithms for NFA concatenation, union, and star.

Illustration 10.3.1 For the RE of Figure 10.14, an NFA can be obtained in a very straightforward manner. The result is very easy to imagine (although left out due to space considerations). It will consist of an NFA for the embedding head sequence, branches labeled by ε to the various cases of errors, finally converging on an NFA fragment for the embedding tail sequence. The cases of errors will also have NFA fragments that directly track the RE syntax. □

10.3.2 NFA to RE

Given NFA $N = (Q, \Sigma, \delta, q_0, F)$, we can convert it to a regular expression by successively eliminating its states. This is often called the *generalized NFA* or GNFA approach, where we build GNFA whose transitions are labeled using regular expressions, instead of members of Σ_ε.

- Preprocess N by adding one new initial state B and one new final state E. In the following steps, we will eliminate every state of N, leaving only B and E behind. The transition connecting B and E will be labeled with the desired RE upon termination of our algorithm.
- Make all states F nonfinal, and introduce a transition from every state in F to E via ε.
- Introduce an ε transition from B to q_0.
- Repeatedly eliminate a state from Q. In all the steps below, whenever any pair of states p and q has two transitions going from p to q, labeled with regular expressions R_1 and R_2, replace them by a single transition labeled $R_1 + R_2$.
- Suppose
  - state p has a transition into s labeled with RE R_{ps},
  - state s has a transition to itself labeled R_{ss}, and
  - state s has a transition out of it to state q labeled R_{sq}.

  Then, we can eliminate the ps and sq transitions, and introduce a direct transition from p to q labeled $R_{ps}(R_{ss})^*R_{sq}$. We keep repeating these steps until state s is disconnected from the rest of the graph, at which point, it can be eliminated.

We now illustrate the NFA to RE conversion algorithm on the example NFA given in Figure 10.6. The result of the preprocessing step is in Figure 10.7. This machine is represented as a GNFA.

Notice that state B0 reaches state A1 via `(1+e)`. Therefore, we replace the two edges going from B0 to A1 by a single edge labeled with `(1+e)`. No other immediate simplifications are possible.

Now, let us eliminate state A1. Introduce the following paths:

`FA` to `FB` labeled by `(11*)` (this is to be understood as `(1(1*))` or as `1(1*)`, as `*` binds tighter than concatenation).
`FB` to `FA` labeled by `(1*)`.
`B0` to `FA` labeled by `(1+e)(1*)`.
`B0` to `FB` labeled by `(1+e)(1*)`.

The result appears in Figure 10.8.
Now, let us eliminate state B0. Introduce the following paths:

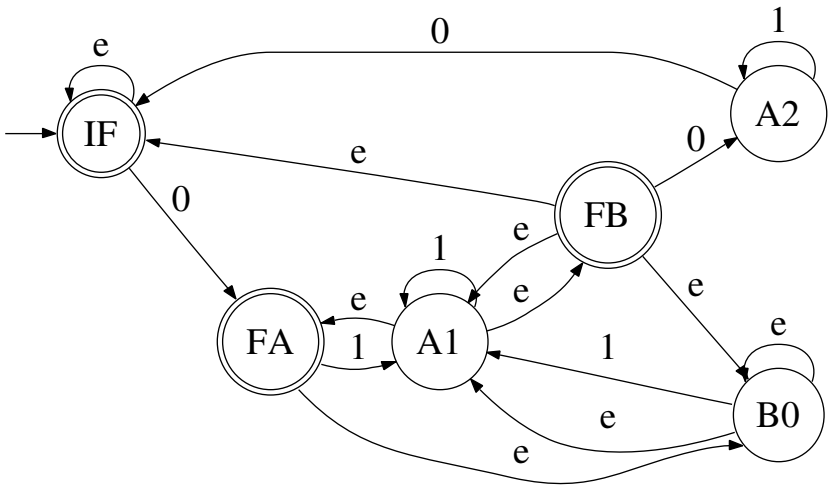


Fig. 10.6. An example NFA to be converted to a regular expression

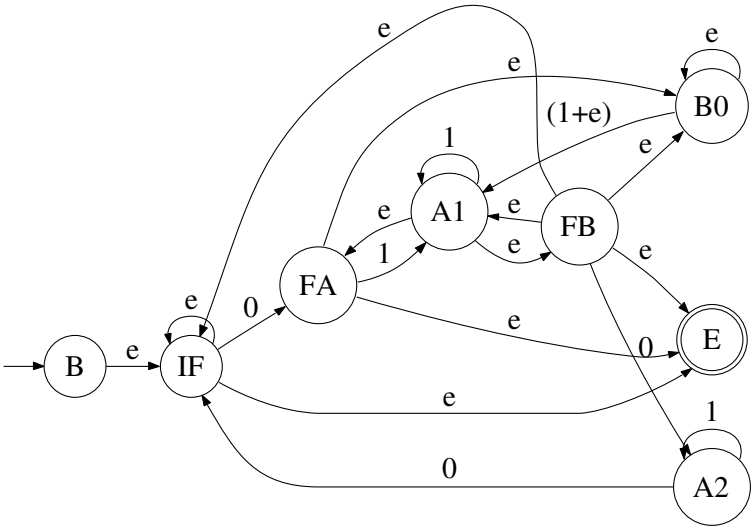


Fig. 10.7. Result of the preprocessing step

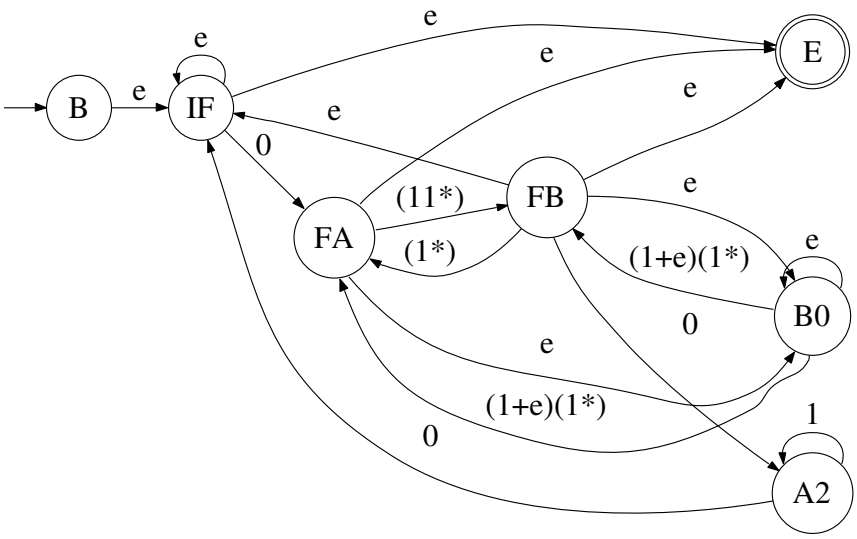


Fig. 10.8. Result of eliminating state A1

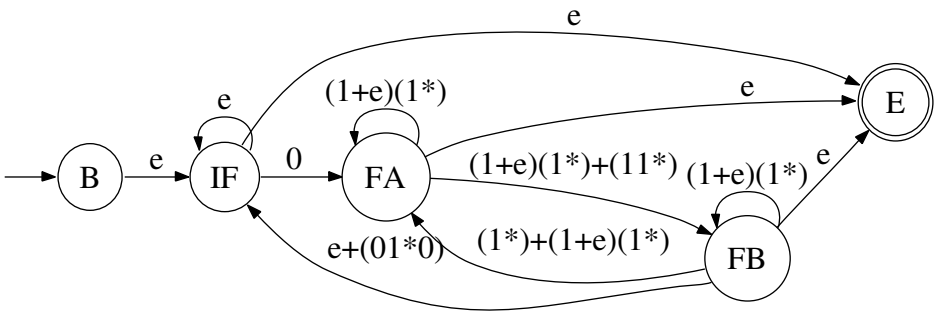


Fig. 10.9. Result of Eliminating B0 and A2

`FB` to `IF` labeled by `e`.
`FB` to `FA` labeled by `((1*)+(1+e)(1*))`.
`FA` to `FB` labeled by `((1+e)(1*)+(11*))`.
Self-loop `FA` to `FA` labeled by `(1+e)(1*)`.
Self-loop `FB` to `FB` labeled by `(1+e)(1*)`.

We don't depict this result yet. Let us also eliminate state A2 and then depict the combined results of eliminating B0 and A2. In eliminating A2, we merge the resulting `FB` to `IF` path with the existing one, resulting

in the single FB to IF path labeled by the following label (with no other changes in the GNFA):

```
e+(01*0)
```

The combined result of eliminating B0 and A2 appear in Figure 10.9.

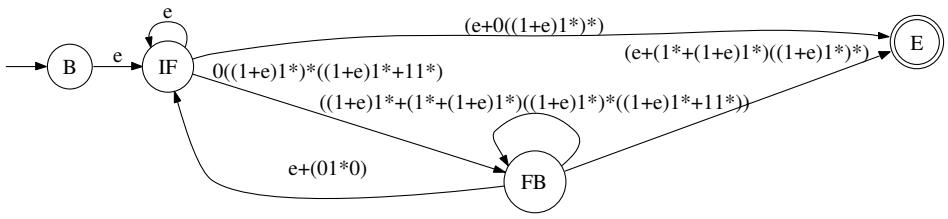


Fig. 10.10. Result of Eliminating FA

Now, let us choose to eliminate state FA. Introduce the following paths:

IF to E labeled by `(e+0((1+e)1*)*)`.
Self-loop FB to FB labeled by `((1+e)1*+(1*+(1+e)1*)((1+e)1*)*((1+e)1*+11*))`.
IF to FB labeled by `0((1+e)1*)*((1+e)1*+11*)`.
FB to E labeled by `(e+(1*+(1+e)1*)((1+e)1*)*)`.

The results appear in Figure 10.10.
We leave the final result (obtainable by eliminating FB and IF) as Exercise 10.17.

From this example, it should be evident that

- NFAs can *often* express regular languages far more intuitively than corresponding regular expressions can. The intuitiveness is due to the use of intermediate states that help split the behavior into various categories.
- Sometimes, minimal DFAs (Section 10.3.3) are not very intuitive (e.g., Figure 10.13), while regular expressions (see Section 10.4.2) and their corresponding NFAs (see Section 10.3.1) are quite intuitive.