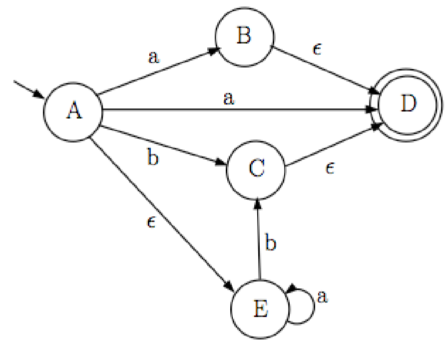
CMSC 330: Organization of Programming Languages

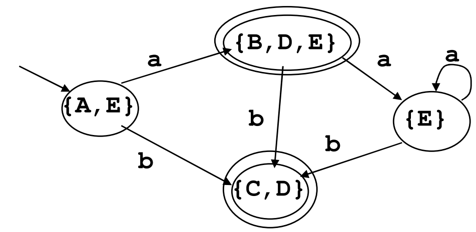
Finite Automata, con't.

NFA → DFA Example 3

- NFA



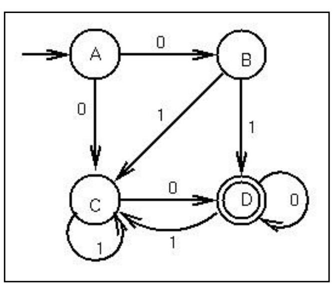
- DFA



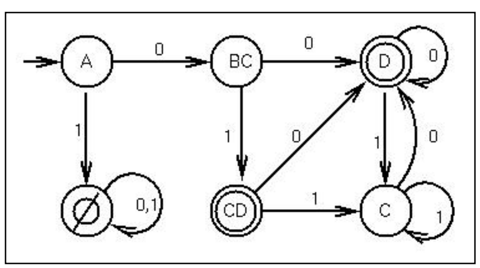
R = { {A,E}, {B,D,E}, {C,D}, {E}

Equivalence of DFAs and NFAs

- Any string from {A} to either {D} or {CD} represents a path from A to D in the original NFA



NFA



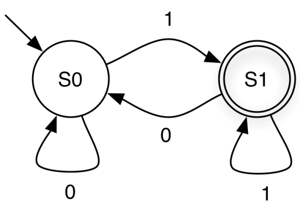
DFA

Equivalence of DFAs and NFAs (cont.)

- Can reduce any NFA to a DFA using subset alg.
- How many states in the DFA?
  - Each DFA state is a subset of the set of NFA states
  - Given NFA with n states, DFA may have 2^n states
    - Since a set with n items may have 2^n subsets
  - Corollary
    - Reducing a NFA with n states may be $O(2^n)$

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");
             break;
  }
}
```


Implementing DFAs (Alternative)

Alternatively, use generic table-driven DFA

given components (Σ, Q, q_0, F, δ) of a DFA:
q := q_0
while (there exists another symbol s of the input string)
  q := δ(q, s)
if q$\in$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

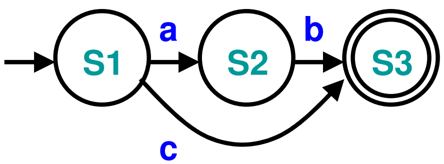
- Given a string s, how long does algorithm take to decide whether s is accepted?
  - assume we can compute δ(q0, c) in constant time
  - then the time per string s to determine acceptance is O(|s|)
  - can't get much faster!
- But recall that constructing the DFA from the NFA constructed from the regular 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

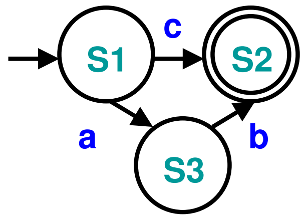
Regular Expressions in Practice

- 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_A, q_A, {f_A}, δ_A), 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

Minimizing DFA

- Result from CS theory
  - Every regular language is recognizable by a minimum-state DFA that is **unique** up to state names
- In other words
  - For every DFA, there is a unique DFA with minimum number of states that accepts the same language
  - Two minimum-state DFAs have **same** underlying shape





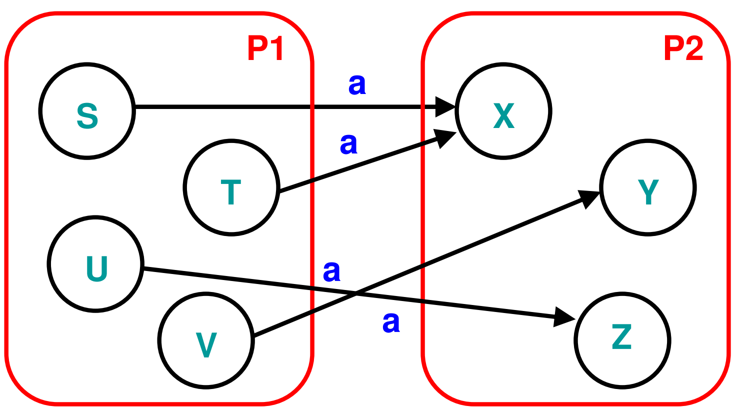
Minimizing DFA: Hopcroft Reduction

- Intuition
  - Look for states that can be distinguish from each other
    - End up in different accept / non-accept state with identical input
- Algorithm
  - Construct initial partition
    - Accepting and non-accepting states
  - Iteratively refine partitions (until partitions remain fixed)
    - Split a partition if members in partition have transitions to different partitions for same input
      - Two states x, y belong in same partition if and only if for all symbols in Σ they transition to the same partition
  - Update transitions and remove dead states

J. Hopcroft, "An n log n algorithm for minimizing states in a finite automaton," 1971

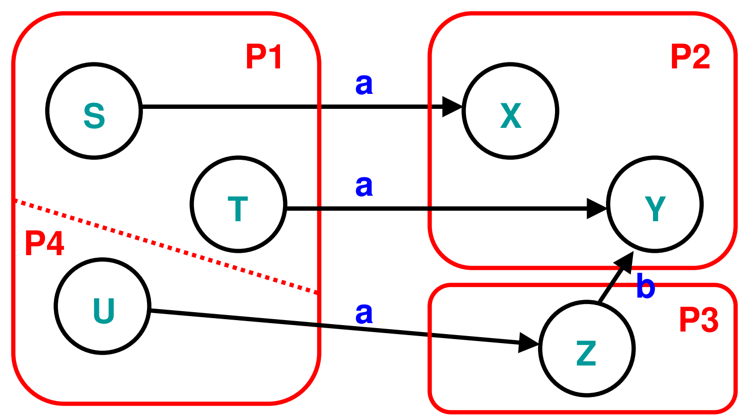
Splitting Partitions

- No need to split partition {S,T,U,V}
  - All transitions on a lead to identical partition P2
  - Even though transitions on a lead to different states



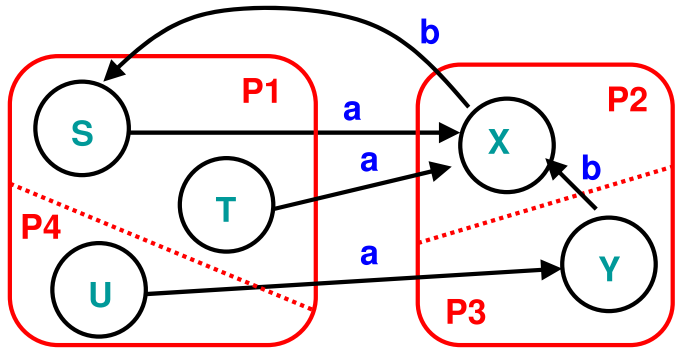
Splitting Partitions (cont.)

- Need to split partition {S,T,U} into {S,T}, {U}
  - Transitions on a from S,T lead to partition P2
  - Transition on a from U lead to partition P3



Resplitting Partitions

- Need to reexamine partitions after splits
  - Initially no need to split partition {S,T,U}
  - After splitting partition {X,Y} into {X}, {Y} need to split partition {S,T,U} into {S,T}, {U}



DFA Minimization Algorithm (1)

- Input DFA (Σ, Q, q_0, F_n, δ_n), output DFA (Σ, R, r_0, F_d, δ_d)
- Algorithm

  Let $p_0 = F_n$, $p_1 = Q – F$ // initial partitions = final, nonfinal states

  Let R = { p | p $\in$ {p_0, p_1} and p != $\varnothing$ }, P = $\varnothing$ // add p to R if nonempty

  While P != R do // while partitions changed on prev iteration

  Let P = R, R = $\varnothing$ // P = prev partitions, R = current partitions

  For each p $\in$ P // for each partition from previous iteration

DFA Minimization Algorithm (2)

- Algorithm for split(p,P)

  Choose some r $\in$ p, let q = p – {r}, m = { } // pick some state r in p

  For each s $\in$ q // for each state in p except for r

  For each c $\in$ Σ // for each symbol in alphabet

  If δ_d(r ,c) = q_0 and δ_d(s, c) = q_1 and // q's = states reached for c

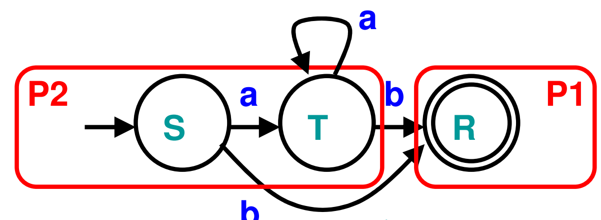
  there is no $p_1 \in$ P such that $q_0 \in p_1$ and $q_1 \in p_1$ then

  m = m $\cup$ {s} // add s to m if q's not in same partition

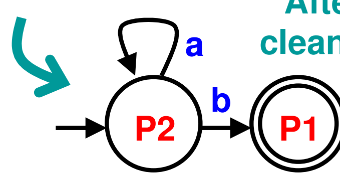
  Return p – m, m // m = states that behave

Minimizing DFA: Example 1

- DFA



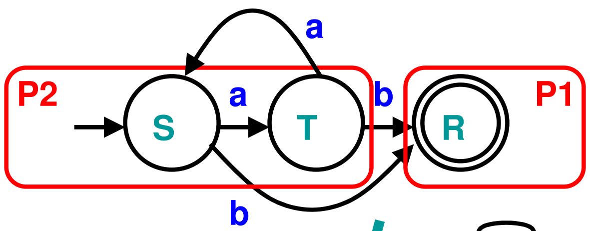
After cleanup



- Initial partitions
  - Accept **{ R }** → **P1**
  - Reject **{ S, T }** → **P2**
- Split partition? → Not required, minimization done
  - move(S, a) **= T → P2**
  - move(T, a) **= T → P2**
  - move(S, b) **= R → P1**
  - move(T, b) **= R → P1**

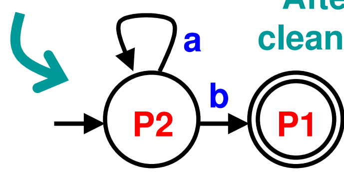
Minimizing DFA: Example 2

- DFA



- Initial partitions
  - Accept { R } → P1
  - Reject { S, T } → P2

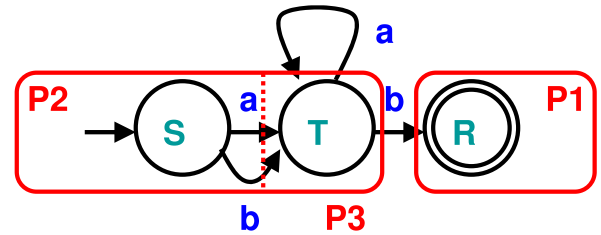
After cleanup



- Split partition? → Not required, minimization done
  - move(S, a) = T → P2
  - move(T, a) = S → P2
  - move(S, b) = R → P1
  - move (T, b) = R → P1

Minimizing DFA: Example 3

- DFA



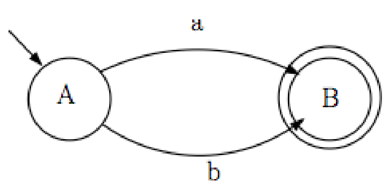
- Initial partitions
  - Accept { R } → P1
  - Reject { S, T } → P2

DFA already minimal

- Split partition? → Yes, different partitions for b
  - move(S, a) = T → P2
  - move(T, a) = T → P2
  - move(S, b) = T → P2
  - move(T, b) = R → P1

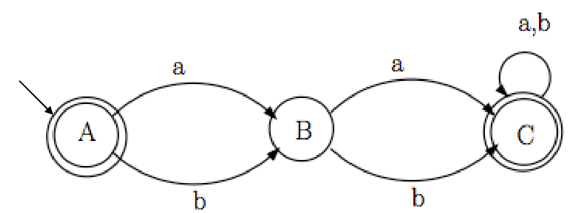
Complement of DFA

- Given a DFA accepting language L, how can we create a DFA accepting its complement?
  - Example DFA
    - $\Sigma = \{a,b\}$



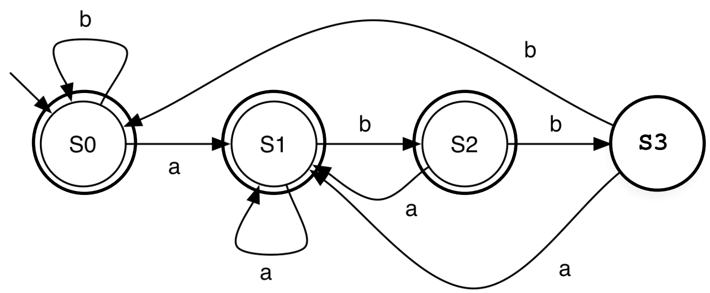
Complement of DFA (cont.)

- Algorithm:
  - Add explicit transitions to a dead state
  - Change every final state to a nonfinal state and every nonfinal state to a final state
- Note this **only** works with DFAs
  - Why not with NFAs?



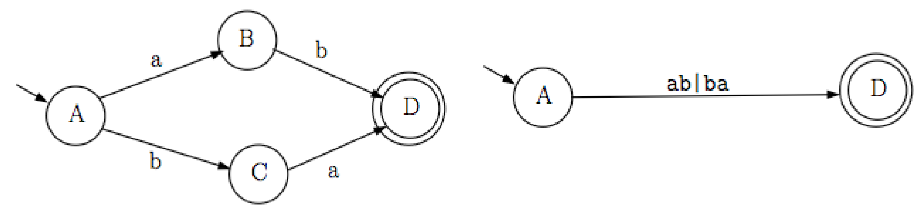
Practice

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



Reducing 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 REs to DFAs and NFAs

- Why do we want to convert between these?
  - Can make it easier to express ideas
  - Can be easier to implement