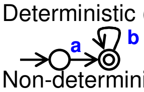
CMSC330

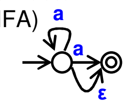
Finite Automata 2

Last Lecture

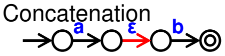
- Finite automata
  - Alphabet, states…
  - (Σ, Q, q_0, F, δ)
- Types
  - Deterministic (DFA)



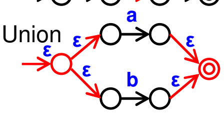
  - Non-deterministic (NFA)



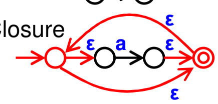
- Reducing RE to NFA
  - Concatenation



  - Union

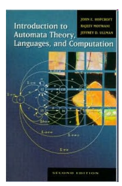


  - Closure



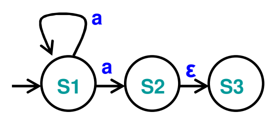
This Lecture

- Reducing NFA to DFA*
  - ε-closure
  - Subset algorithm
- Minimizing DFA*
  - Hopcroft reduction
- Complementing DFA
- Implementing DFA*



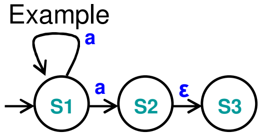
How NFA Works

- When NFA processes a string
  - NFA may be in several possible states
    - Multiple transitions with same label
    - ε-transitions
- Example
  - After processing "a"
    - NFA may be in states
      S1
      S2
      S3

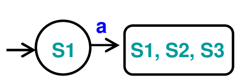


Reducing NFA to DFA

- NFA may be reduced to DFA
  - By explicitly tracking the set of NFA states
- Intuition
  - Build DFA where
    - Each DFA state represents a set of NFA states
- Example



NFA



DFA

Reducing NFA to DFA (cont.)

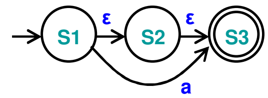
- Reduction applied using the subset algorithm
  - DFA state is a subset of set of all NFA states
- Algorithm
  - Input
    - NFA (Σ, Q, q_0, F_n, δ)
  - Output
    - DFA (Σ, R, r_0, F_d, δ)
  - Using
    - ε-closure(p)
    - move(p, a)

ε-transitions and ε-closure

- We say p $\xrightarrow{\varepsilon}$ q
  - If it is possible to go from state p to state q by taking only ε-transitions
  - If ∃ p, p_1, p_2, … p_n, q ∈ Q such that
    - {p,ε,p_1} ∈ δ, {p_1,ε,p_2} ∈ δ, … , {p_n,ε,q} ∈ δ
- ε-closure(p)
  - Set of states reachable from p using ε-transitions alone
    - Set of states q such that p $\xrightarrow{\varepsilon}$ q
    - ε-closure(p) = {q | p $\xrightarrow{\varepsilon}$ q }
  - Note
    - ε-closure(p) always includes p
    - ε-closure( ) may be applied to set of states (take union)

ε-closure: Example 1

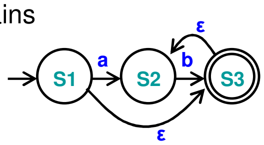
- Following NFA contains
  - S1 $\xrightarrow{\varepsilon}$ S2
  - S2 $\xrightarrow{\varepsilon}$ S3
  - S1 → S3



- ε-closures
  - ε-closure(S1) = { S1, S2, S3 }
  - ε-closure(S2) = { S2, S3 }
  - ε-closure(S3) = { S3 }
  - ε-closure({ S1, S2 }) = { S1, S2, S3 } ∪ { S2, S3 }

ε-closure: Example 2

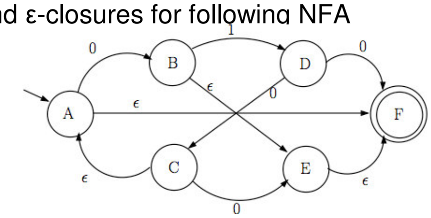
- Following NFA contains
  - S1 $\xrightarrow{\varepsilon}$ S3
  - S3 $\xrightarrow{\varepsilon}$ S2
  - S1 $\xrightarrow{\varepsilon}$ S2



- ε-closures
  - ε-closure(S1) = { S1, S2, S3 }
  - ε-closure(S2) = { S2 }
  - ε-closure(S3) = { S2, S3 }
  - ε-closure({ S2,S3 }) = { S2 } ∪ { S2, S3 }

ε-closure: Practice

- Find ε-closures for following NFA



- Find ε-closures for the NFA you construct for
  - The regular expression (0|1*)111(0*|1)

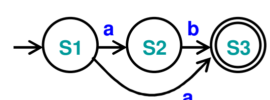
Calculating move(p,a)

- move(p,a)

  - Set of states reachable from p using exactly one transition on a
    - Set of states q such that {p, a, q} ∈ δ
    - move(p,a) = {q | {p, a, q} ∈ δ}
  - Note move(p,a) may be empty Ø
    - If no transition from p with label a

move(p,a) : Example 1

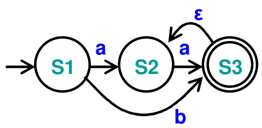
- Following NFA
  - Σ = { a, b }



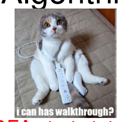
- Move
  - move(S1, a) = { S2, S3 }
  - move(S1, b) = Ø
  - move(S2, a) = Ø
  - move(S2, b) = { S3 }
  - move(S3, a) = Ø
  - move(S3, b) = Ø

move(p,a) : Example 2

- Following NFA
  - Σ = { a, b }
- Move
  - move(S1, a) = { S2 }
  - move(S1, b) = { S3 }
  - move(S2, a) = { S3 }
  - move(S2, b) = Ø
  - move(S3, a) = Ø
  - move(S3, b) = Ø



NFA → DFA Reduction Algorithm

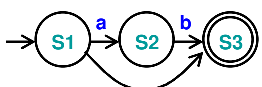


- Input NFA (Σ, Q, q_0, F_n, δ),
  Output DFA (Σ, R, r_0, F_d, δ)
- Algorithm
  - Let r_0 = ε-closure(q_0), add it to R   // DFA start state
  - While ∃ an unmarked state r ∈ R   // process DFA state r
    - Mark r   // each state visited once
    - For each a ∈ Σ // for each letter a
      - Let S = {s | q ∈ r & move(q,a) = s}   // states reached via a
      - Let e = ε-closure(S)   // states reached via ε
      - If e ∉ R   // if state e is new
        Let R = e ∪ R   // add e to R (unmarked)
      - Let δ = δ ∪ {r, a, e}   // add transition r→e
  - Let F_d = {r | ∃ s ∈ r with s ∈ F_n} // final if include state in F_n

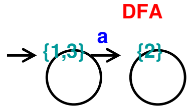
NFA → DFA Example 1

- Start = ε-closure(S1) = { {S1,S3} }
- R = { {S1,S3} }
- r ∈ R = {S1,S3}
- Move({S1,S3},a}) = {S2}
  - e = ε-closure({S2}) = {S2}
  - R = R ∪ {S2} = { {S1,S3}, {S2} }
  - δ = δ ∪ {{S1,S3}, a, {S2}}
- Move({S1,S3},b}) = Ø

NFA



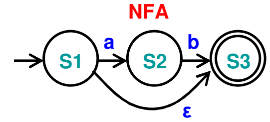
DFA



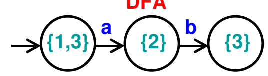
NFA → DFA Example 1 (cont.)

- R = { {S1,S3}, {S2} }
- r ∈ R = {S2}
- Move({S2},a} = Ø
- Move({S2},b} = {S3}
  - e = ε-closure({S3}) = {S3}
  - R = R ∪ {S3} = { {S1,S3}, {S2}, {S3} }
  - δ = δ ∪ {{S2}, b, {S3}}

NFA



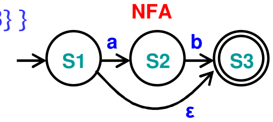
DFA



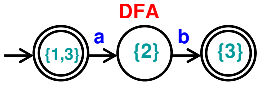
NFA → DFA Example 1 (cont.)

- R = { {S1,S3}, {S2}, {S3} }
- r ∈ R = {S3}
- Move({S3},a} = Ø
- Move({S3},b} = Ø
- F_d = {{S1,S3}, {S3}}
  - Since S3 ∈ F_n
- Done!

NFA

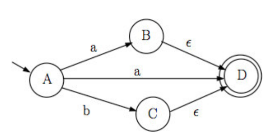


DFA

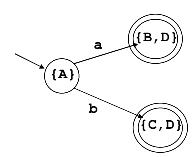


NFA → DFA Example 2

- NFA



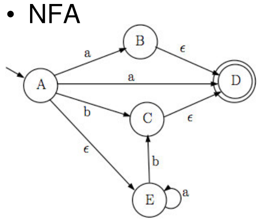
- DFA



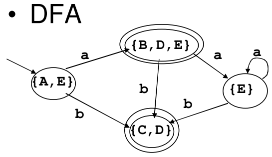
R = { {A}, {B,D}, {C,D} }

NFA → DFA Example 3

- NFA



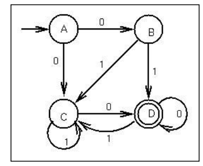
- DFA



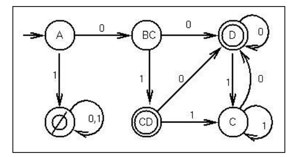
R = { {A,E}, {B,D,E}, {C,D}, {E} }

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



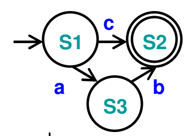
DFA

Equivalence of DFAs and NFAs (cont.)

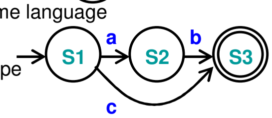
- Can reduce any NFA to a DFA using subset algorithm
- How many states in the DFA?
  - Each DFA state is a subset of the set of NFA states
  - Given NFA with n states, DFA may have 2^n states
    - Since a set with n items may have 2^n subsets
  - Corollary
    - Reducing a NFA with n states may be $O(2^n)$

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



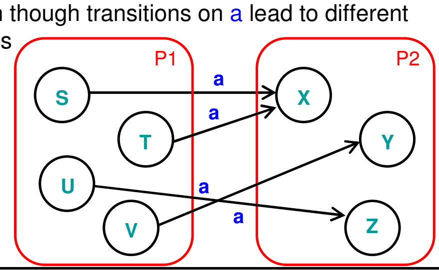
Minimizing DFA: Hopcroft Reduction

- 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 & 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 & remove dead states

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

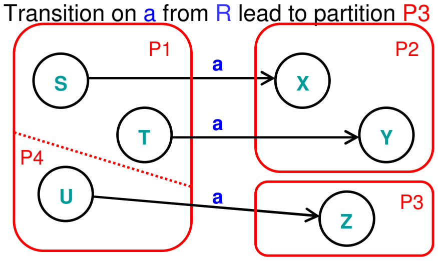
Splitting Partitions

- 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

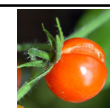


Splitting Partitions (cont.)

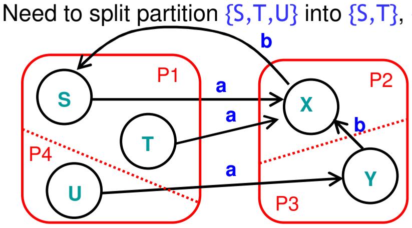
- Need to split partition {S,T,U} into {S,T}, {U}
  - Transitions on a from S,T lead to partition P2
  - Transition on a from R lead to partition P3



Resplitting Partitions

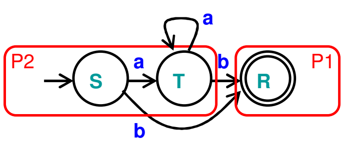


- Need to reexamine partitions after splits
  - Initially no need to split partition {S,T,U}
  - After splitting partition {X,Y} into {X}, {Y}
  - Need to split partition {S,T,U} into {S,T}, {U}



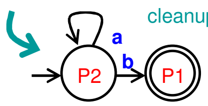
Minimizing DFA: Example 1

- DFA



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

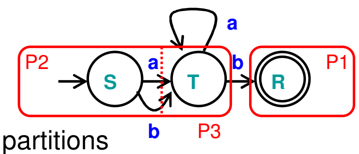
After cleanup



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

Minimizing DFA: Example 2

- DFA



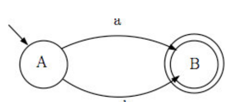
- Initial partitions
  - Accept { R } → P1
  - Reject { S, T } → P2

DFA already minimal

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

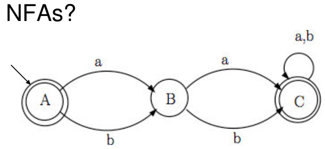
Complement of DFA

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

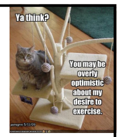


Complement of DFA (cont.)

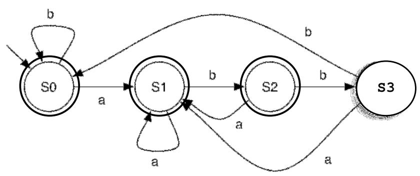
- Algorithm
  - Add explicit transitions to a dead state
  - Change every accepting state to a non-accepting state & every non-accepting state to an accepting state
- Note this only works with DFAs
  - Why not with NFAs?



Practice

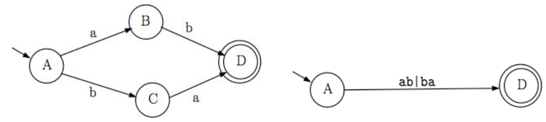


Make the DFA which accepts the complement of the language accepted by the DFA below.



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

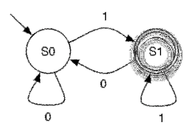


Relating REs to DFAs and NFAs

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

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, q0, F, δ) of a DFA:
let 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 DFA

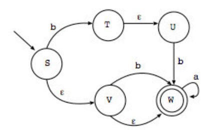
- How long for DFA to decide to accept/reject string s?
  - Assume we can compute $\delta(q, c)$ in constant time
  - Then time to process s is $O(|s|)$
    - Can't get much faster!
- Constructing DFA for RE A may take $O(2^{|A|})$ time
  - But usually not the case in practice
- So there's the initial overhead
  - But then processing strings is fast

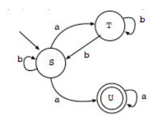
Regular Expressions in Practice

- 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 $(\Sigma, Q_A, q_A, \{f_A\}, \delta_A)$, the components of the DFA produced from the RE
- 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

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)

Summary of Regular Expression Theory

- Finite automata
  - DFA, NFA
- Equivalence of RE, NFA, DFA
  - RE → NFA
    - Concatenation, union, closure
  - NFA → DFA
    - ε-closure & subset algorithm
- DFA
  - Minimization, complement
  - Implementation

Discussion Tomorrow

- I will be in cville
- Going through

  regexp->NFA->DFA->minimization examples
- Quiz!
  - Your ego at work here…
  - Everything from last week (so no RE theory, no NFA/DFA)