# Finite-State Methods in Natural-Language Processing: Algorithms

**Ronald M. Kaplan**

**and**

**Martin Kay**

# Data Structures

## FSM

- states
- start
- sigma
- properties
  - (epsilon-free, determinisitc ...)

## State

- final
- arcs
- name
- mark

## Arc

- label
- destination

# A Traversal Function

```
Traverse(FSMs, StartFn, FinalFn, ArcsFn)
   Start := StartFn(FSMs);
   States := (Start);
   STP := (Start);
   while s := pop(STP) do
         s.final := FinalFn(s.name);
         s.arcs := ArcsFn(s.name);
return new FSM[states = States,
               start=start];
```

```
GetState(n)
     if there is an s in States
                    with s.name = n
          return s ;
     else s := new State[name=n];
          push s, States
          push s, STP
          return s
```

# Copy

```
Traverse(FSMs, StartFn, FinalFn,
   ArcsFn)
   Start := StartFn(FSMs);
   States := (Start);
   STP := (Start);
   while s := pop(STP) do
         s.final := FinalFn(s.name);
         s.arcs := ArcsFn(s.name);
return new[states = States,
              start=start];
```

N.B. The *name* of a state in the copied machine is the *state* itself in the machine being copied.

StartFn(n) = **new** State[name=n.start]
FinalFn(n) = n.final
ArcsFn(n) = {**new** Arc[label=a.label,
destination=GetState(a.destination)] |
a **in** n.arcs}

# The Paradigm

```
Copy(FSM) =
 Travers(FSM, 'CopyStartFn, 'CopyFinalFn,
   'CopyArcsFn)
Inverse(FSM) =
 Travers(FSM, 'InverseStartFn, 'InverseFinalFn,
   'InverseArcsFn)
CrossProduct(<FSM1, FSM2>) =
 Travers(<FSM1, FSM2>, 'CrossProductStartFn,
   'CrossProductFinalFn, 'CrossProductArcsFn)
Intersection(<FSM1, FSM2>) =
 Travers(<FSM1, FSM2>, 'IntersectionStartFn,
   'IntersectionFinalFn, 'IntersectionArcsFn)
```

# Inverse

```
Traverse(FSMs, StartFn, FinalFn,
    ArcsFn)
    Start := StartFn(FSMs);
    States := (Start);
    STP := (Start);
    while s := pop(STP) do
            s.final := FinalFn(s.name);
            s.arcs := ArcsFn(s.name);
return new[states = States,
            start=start];
```

StartFn(n) = **new** State[name=n.start]
FinalFn(n) = n.final
ArcsFn(n) = {**new** Arc[label=y:x,
destination=GetState(a.destination)] |
a **in** n.arcs & a.label = x:y}

# Prune

**Prune(F)=Reverse(Copy(Reverse(F)))**

**N.B. Not ε-free**

Dead states are not reachable in **Reverse(F)**
∴ They are not included in **Copy(Reverse(F))**
∴ They are not included in **Copy(Reverse((Reverse(F)))**

# Cross Product

```
StartFn(<f1, f2>) = new State[name=<f1.start, f2.start>]
FinalFn(<s1, s2>) = s1.final & s2.final
ArcsFn(<s1, s2>) = {new Arc[label=a1.label:a2.label,
                          destination=GetState(<a1.destination,
                                                a2.destination>)] |
                        a1 in s1.arcs & a2 in s2.arcs}
                  ∪ {new Arc[label=a1.label:ε,
                          destination=GetState(<ε, a2.destination>)] |
                        a1 in s1.arcs & (s2.final | s2=ε}
                  ∪ {new Arc[label=ε:a2.label,
                          destination=GetState(<ε, a2.destination>)] |
                        a1 in (s1.final | s1=ε & a2 in s2.arcs}
```

```
Traverse(FSMs, StartFn, FinalFn,
    ArcsFn)
    Start := StartFn(FSMs);
    States := (Start);
    STP := (Start);
    while s := pop(STP) do
            s.final := FinalFn(s.name);
            s.arcs := ArcsFn(s.name);
return new[states = States,
                start=start];
```

# Intersection

N.B.

- Result is not necessarily pruned because some paths die.
- FSMs must be ε-free.

StartFn(<f1, f2>) = **new** State[name=<f1.start, f2.start>]

FinalFn(<s1, s2>) = s1.final & s2.final

ArcsFn(<s1, s2>) = {**new** Arc[label=L,
destination=GetState(<a1.destination,
a2.destination>)] |
a1 **in** s1.arcs & a2 **in** s2.arcs & L=a1.label=a2.label}

# Intersection

## (with ε)

```
StartFn(<f1, f2>) = new State[name=<f1.start, f2.start>]
FinalFn(<s1, s2>) = s1.final & s2.final
ArcsFn(<s1, s2>) =
    {new Arc[label=L, destination=GetState(<a1.destination,
                                             a2.destination>)] |
    a1 in s1.arcs & a2 in s2.arcs & L=a1.label=a2.label}
  ∪ {new Arc[label= ε , destination=GetState(<s1,
                                             a2.destination >)] |
    a1 in s1.arcs & a2 in s2.arcs & a1.label ≠ε & a2.label=ε}
  ∪ {new Arc[label= ε , destination=GetState(<a1.destination ,
                                             s2 >)] |
    a1 in s1.arcs  & a2 in s2.arcs & a1.label=ε & a2.label ≠ε}
```

# StringToFSM

StartFn(<string, f>) = **new** State[name=<string, 0>]
FinalFn(<string, s>) = string=“”“
ArcsFn(<[First | Rest], s>) =
    {**new** Arc[label=First, destination=<s+1, Rest>]}

# Composition (*draft!*)

```
Traverse(FSMs, StartFn, FinalFn,
   ArcsFn)
   Start := StartFn(FSMs);
   States := (Start);
   STP := (Start);
   while s := pop(STP) do
         s.final := FinalFn(s.name);
         s.arcs := ArcsFn(s.name);
return new[states = States,
              start=start];
```

StartFn(<F1, F2>) = **new** State[name=<F1.start, F2.start>]
FinalFn(<s1, s2>) = s1.final & s2.final
ArcsFn(<s1, s2>) = {**new** Arc[label=x:y,
destination=GetState(<a1.destination,
a2.destination>)] |
a1 **in** s1.arcs & a1.label=x:z & a2 **in** s2.arcs & a2.label=z:y}

# Composition

StartFn(<f1, f2>) = **new** State[name=<f1.start, f2.start>]

FinalFn(<s1, s2>) = s1.final & s2.final

ArcsFn(<s1, s2>) =

{**new** Arc[label=x:y, destination=GetState(<a1.destination, a2.destination>)] |
a1 **in** s1.arcs & a1.label=x:z & a2 **in** s2.arcs & a2.label=z:y}

∪ {**new** Arc[label= x:ε , destination=GetState(<a1.destination , s2>)] |
a1 **in** s1.arcs & a1.label= x:ε & a2 **in** s2.arcs & a2.label=z:y & z≠ ε}

∪ {**new** Arc[label= ε:y , destination=GetState(<s1, a2.destination>)] |
a1 **in** s1.arcs & a1.label= x:z & a2 **in** s2.arcs & a2.label= ε:y & z≠ ε}

# Epsilon-closure

```
Epsilon-closure(state) =
  closure := {state} ; STP := {state} ;
  while s := pop(STP) do
     for d in { a.destination | a in s.arcs &
                                a.label=ε &
                                a.detination ∉ closure}
     do if d ∉ closure then { closure := closure ∪ d;
                               STP := STP ∪ d}
  return closure
EC-Arcs(state) = {a | a in s.arcs & a.label ≠ ε
                         for some s in Epsilon-closure(state)}
```

# RemoveEpsilons

StartFn(n) = **new** State[name=n.start]
FinalFn(n) = There is s in Epsilon-closure(n) such that s.final
ArcsFn(n) = {**new** Arc[label=a.label,
destination=GetState(a.destination)] |
a **in** EC-Arcs(n)}

# Intersection — again

```
StartFn(<f1, f2>) = new State[name=<f1.start, f2.start>]
FinalFn(<s1, s2>) = s1.final & s2.final
ArcsFn(<s1, s2>) = {new Arc[label=L,
                        destination=GetState(<a1.destination,
                                              a2.destination>)] |
                    a1 in EC-Arcs(s1) &
                    a2 in EC-Arcs(s2) &
                    L=a1.label=a2.label}
```

# Determinize

StartFn(f) = **new** State[name={f.start}]
FinalFn(n) = There is an s in n such that s.final;
ArcsFn(n) =
{**new** Arc[label=L,
destination=GetState(D)] |
$D=\cup_{s\in n}$ {a.destination | a ∈ EC-Arcs(s) &
a.label = L}}

# Complete

DeadState:=**new** State ;
DeadState.arcs := {**new** Arc[label=l,
destination = DeadState] | l **in** Σ}

StartFn(f) = **new** State[name=f.start]
FinalFn(n) = n.final;
ArcsFn(n) =
{**new** Arc[label=a.label, destination=GetState(a.destination)] |
a **in** n.arcs}
∪ {**new** Arc[label=l, destination=DeadState] | There is no a in
n.arcs such that a.label=l for l in Σ}

# Complement

```
Complement(FSM) =
  Traverse(Determinize(Complete(FSM)), S, F, A)

where

  S (f) = new State[name = f.start]
  F(n) = ~n.final
  ArcsFn(n) =
        {new Arc[label=a.label,
                destination=GetState(a.destination)] | a in n.arcs}
```

# Minus

**Minus(FSM1, FSM2) =**

**Intersect(FSM1, Complement(FSM2))**

# Empty

Otherwise the FSM could contain a final state not reachable from the start state.

```
Empty(FSM) =
  New := Copy(FSM) ;
  There is no s in New.states such that s.final.
```

# Equivalence

**L1=L2 ≡ L1-L2=L2-L1={}**

# Minimization

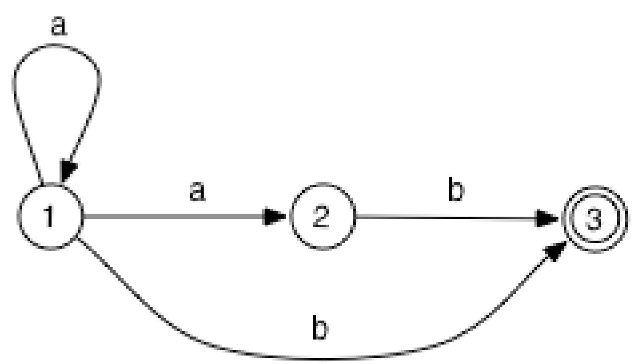
## using the Brzozovsky Construction

**Determinize(Reverse(Determinize(Reverse(f))))**

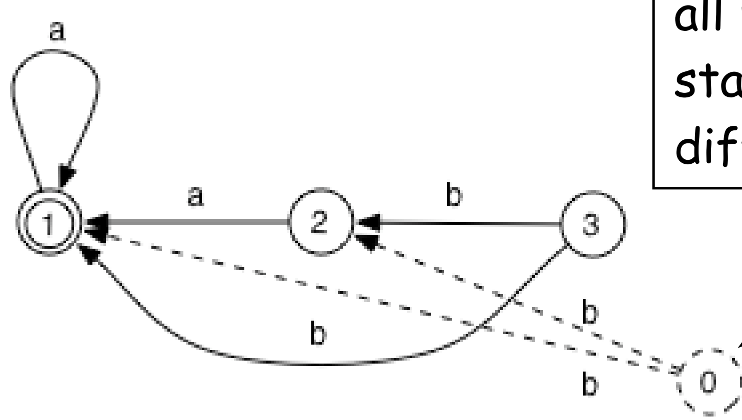
- Each suffix s of f is a prefix s’ of Reverse(f) and, in DR(f)=Determinize(Reverse(f)), δ(start(DR(f)), s’) is a unique state q.
- In Reverse(Determinize(Reverse(f))), q is the only state whose suffix set contains s.
- Since each state in Determinize(Reverse(Determinize(Reverse(f)))) corresponds to a different subset of the states of Reverse(Determinize(Reverse(f))), each has a unique suffix set.☐

# Minimize this

## using the Brzozovsky Construction

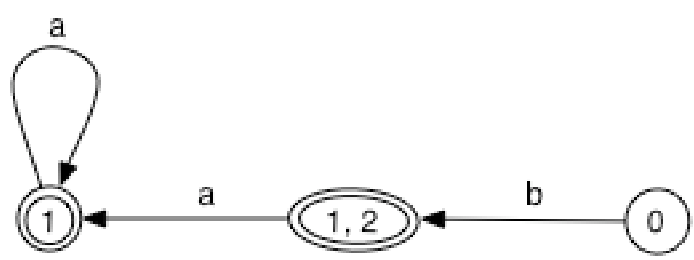


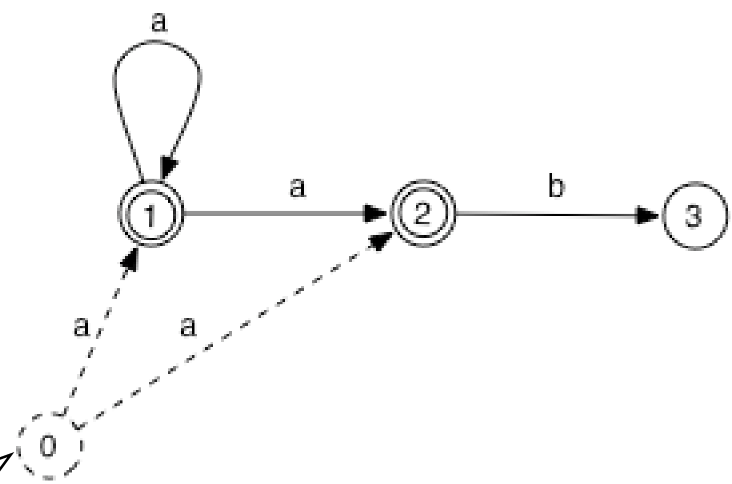
Reverse




New start state points to all previous final states—makes no difference in this case.

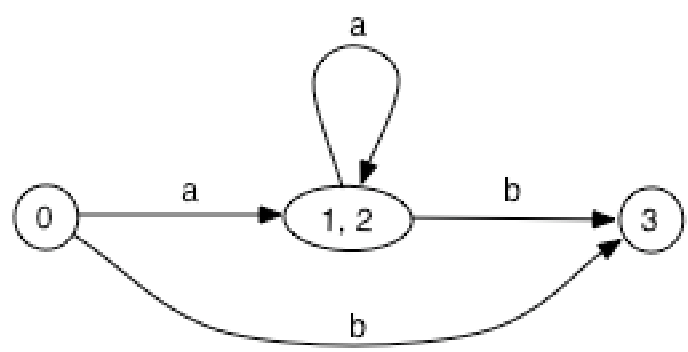
Determinize






Reverse

This time it does make a difference!

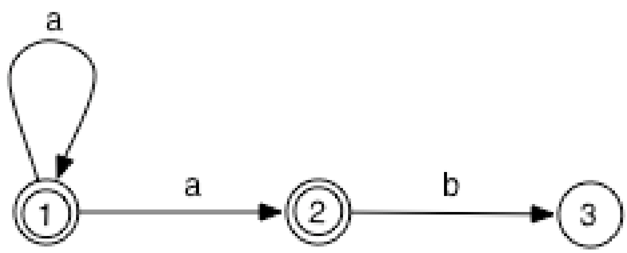



Determinize

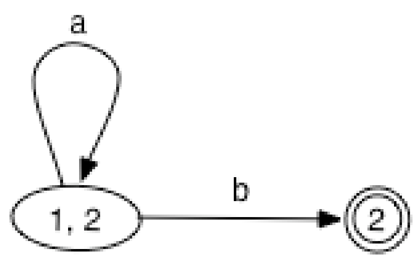
**But it's not minimal!**

# Multiple Start States

Reverse




Determinize




**But it's ~~not~~ minimal!**

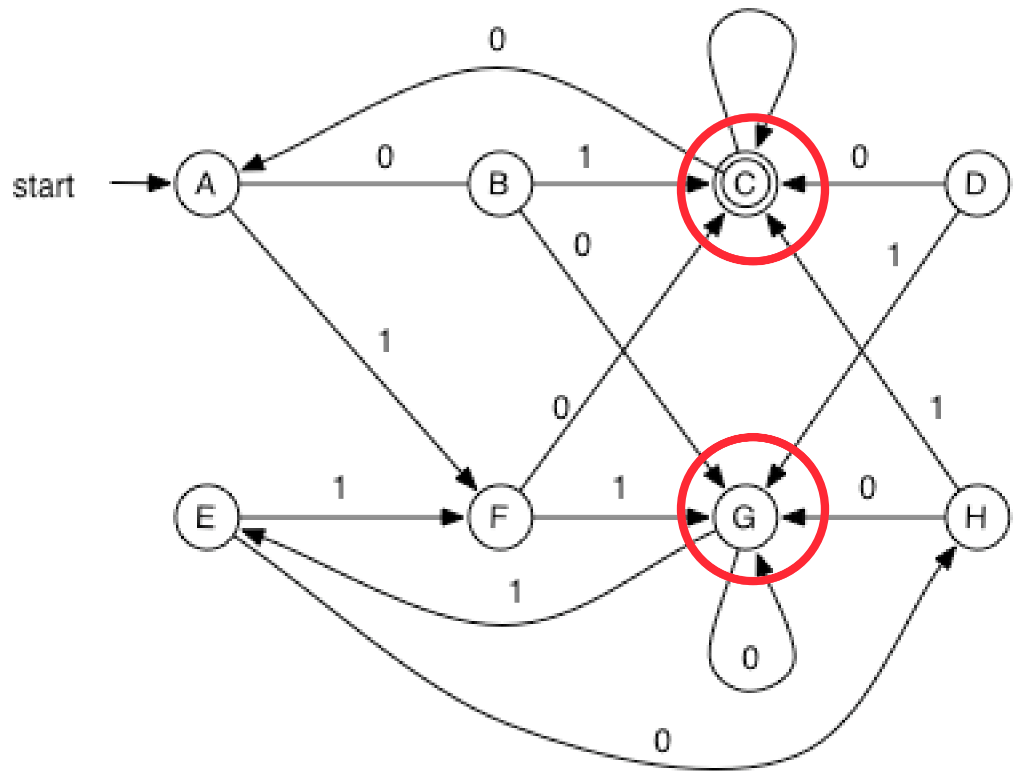
# Minimization

Give an FSM f and a state s, let suffix(f, s) be the FSM that results from replacing the start state of f with s

To minimize an FSM f, conflate all pairs of states s1 and s2 in F iff equivalent(suffix(f, s1), suffix(f, s2))

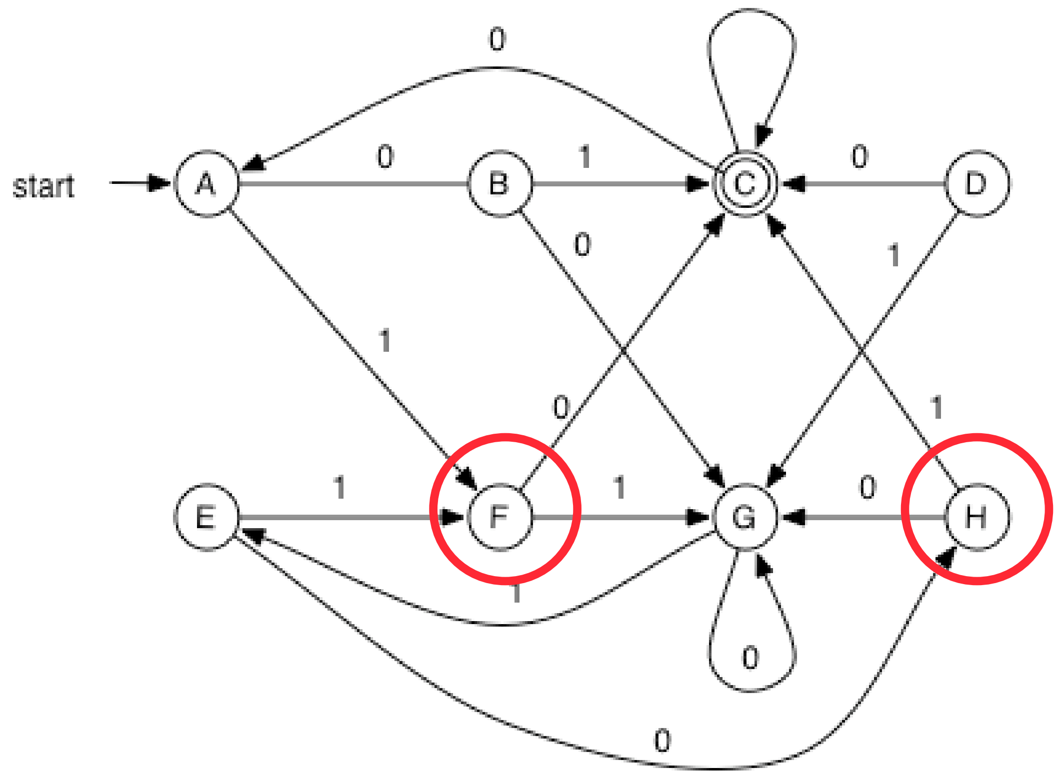
Exponential because involves determinization

# Strings *distinguish* states



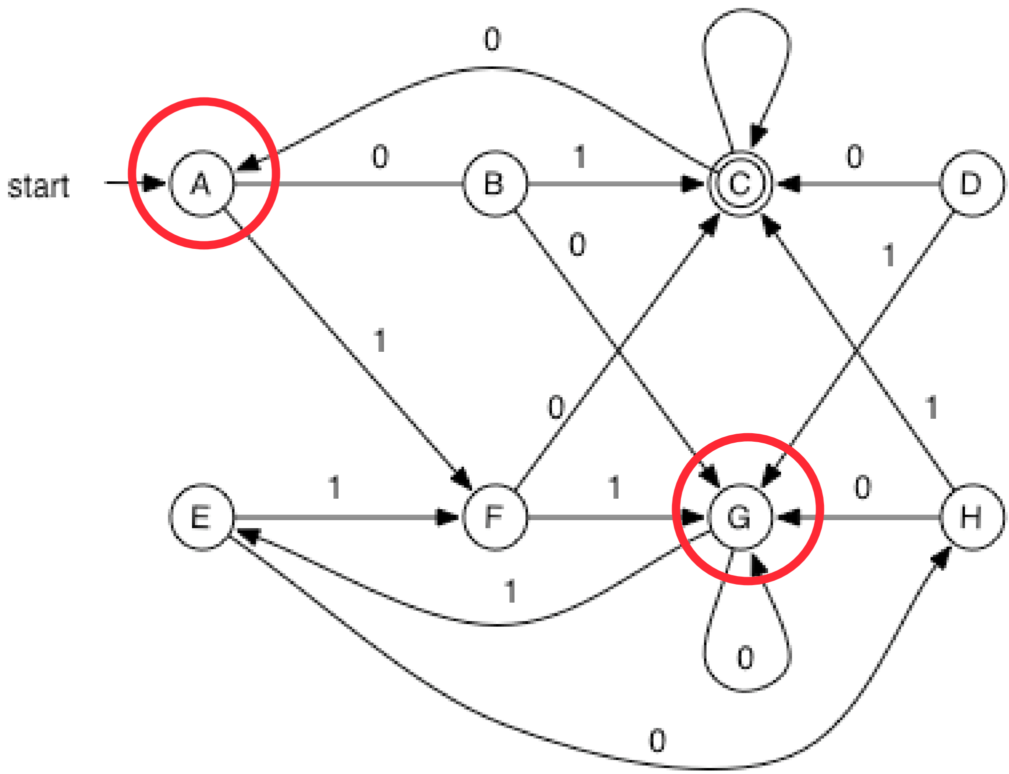

Disitnguished by ε

# Strings distinguish states




Disitnguished by 0 (and 1)

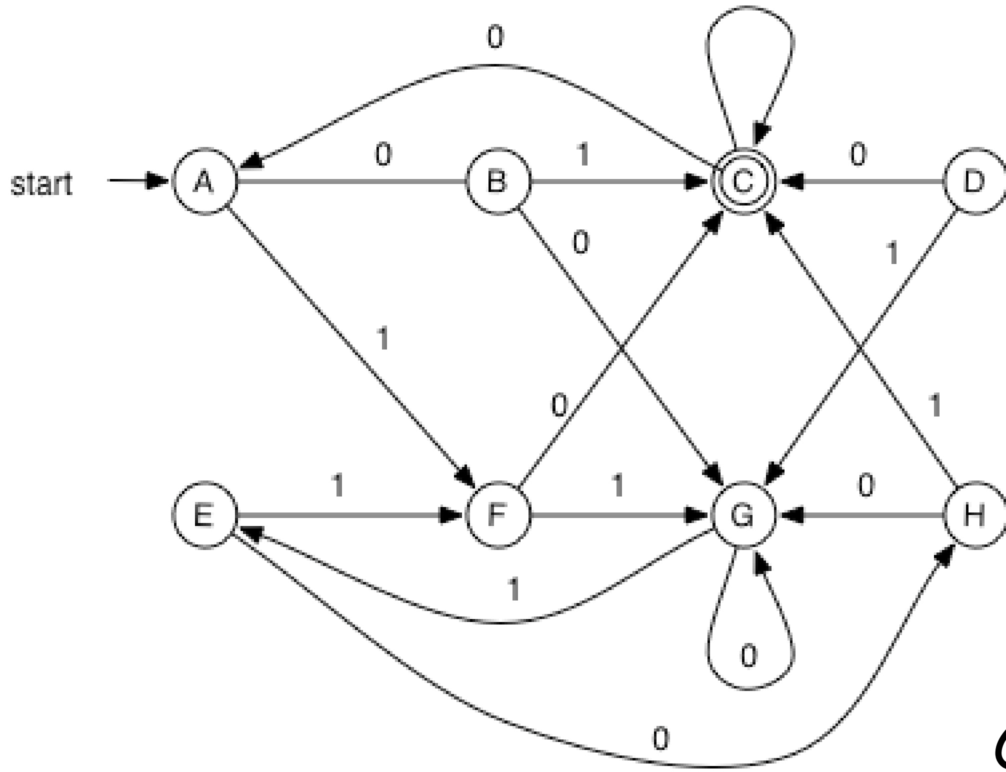
# Strings *distinguish* states




Disitnguished by 01

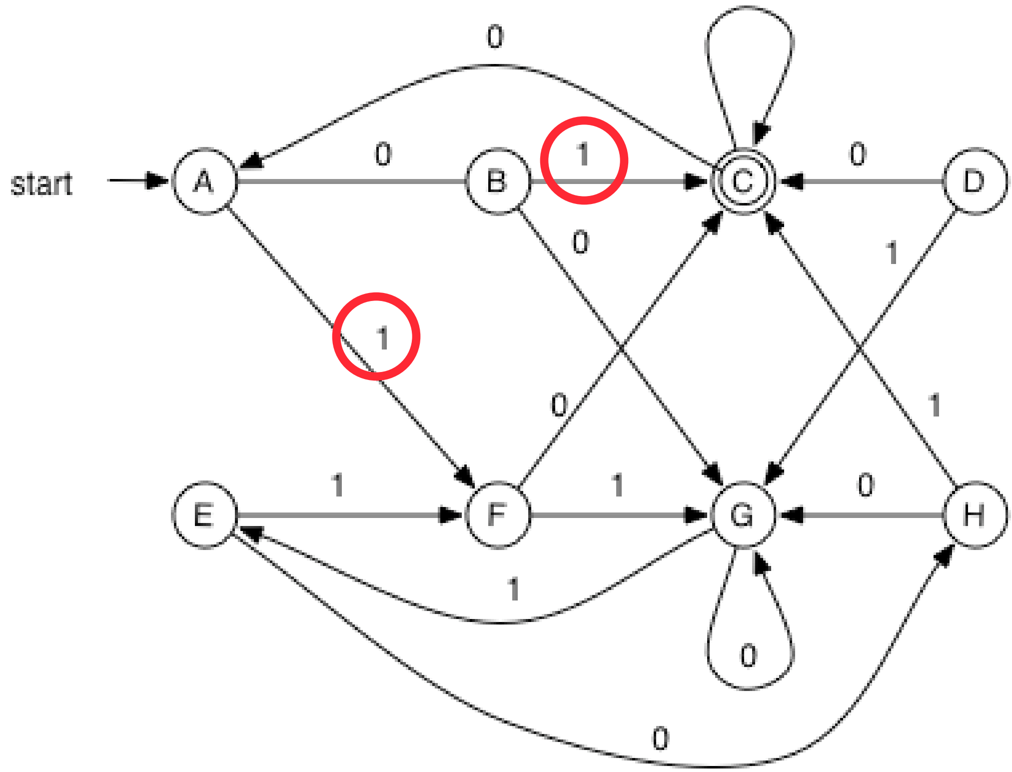
# Minimization

**To determine if a pair of states in an *n*-state FSA is distinguishable, it is sufficient to consider strings of length *n* because no states need be visited more than once.**

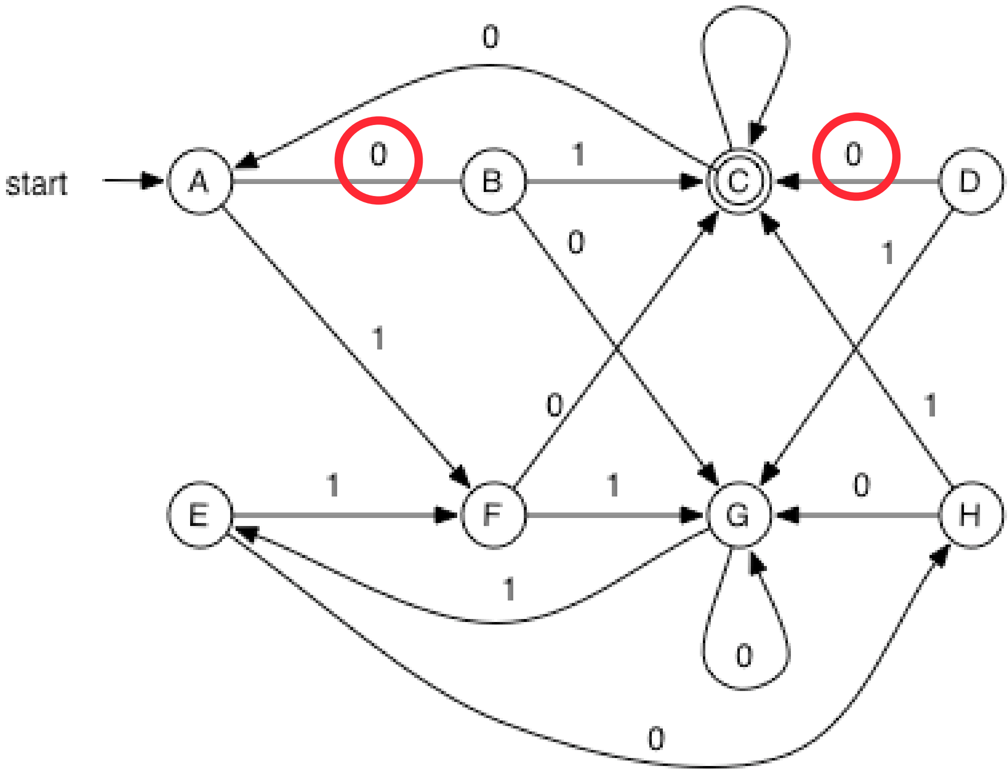



| | | | | | | | |
|---|---|---|---|---|---|---|---|
| B | | | | | | | |
| C | x | x | | | | | |
| D | | | x | | | | |
| E | | | x | | | | |
| F | | | x | | | | |
| G | | | x | | | | |
| H | | | x | | | | |
| | A | B | C | E | D | F | G |

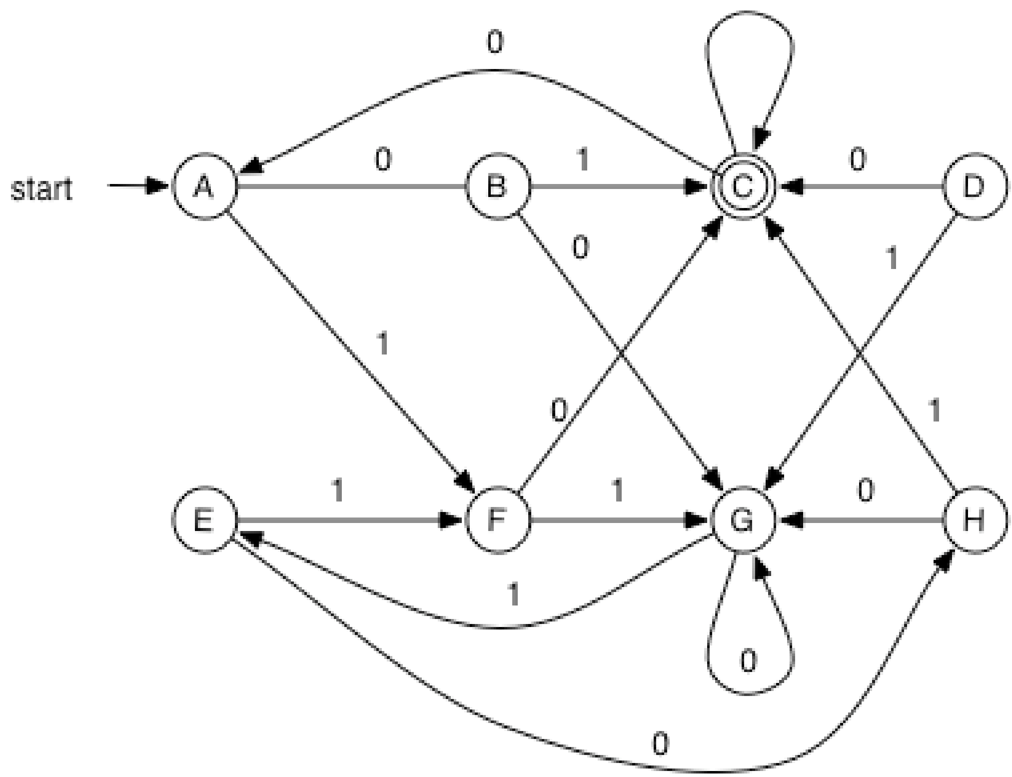
C is the only final state, so it is distinguishable from all others.




| | A | B | C | E | D | F | G |
|---|---|---|---|---|---|---|---|
| B | x | | | | | | |
| C | x | x | | | | | |
| D | | | x | | | | |
| E | | | x | | | | |
| F | | | x | | | | |
| G | | | x | | | | |
| H | | | x | | | | |

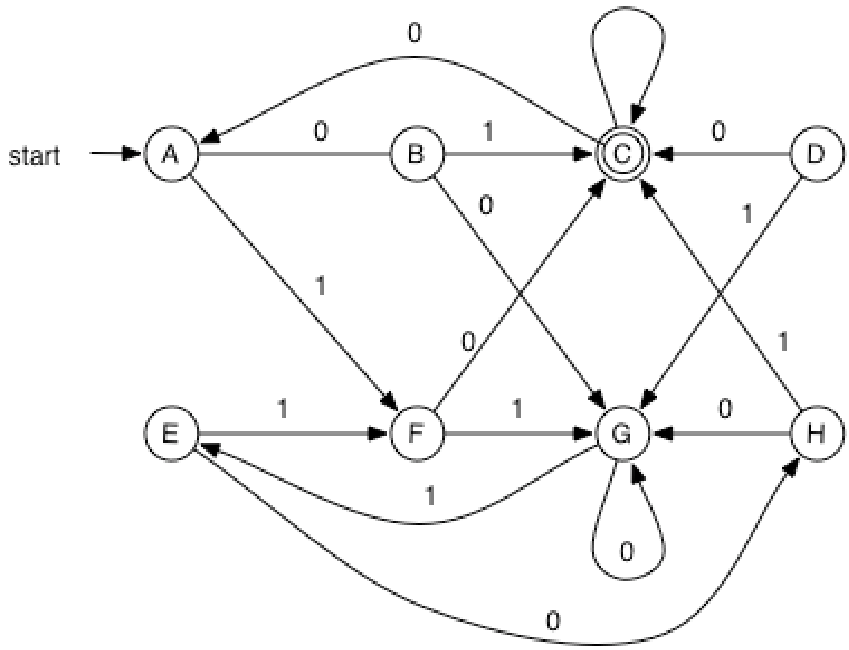


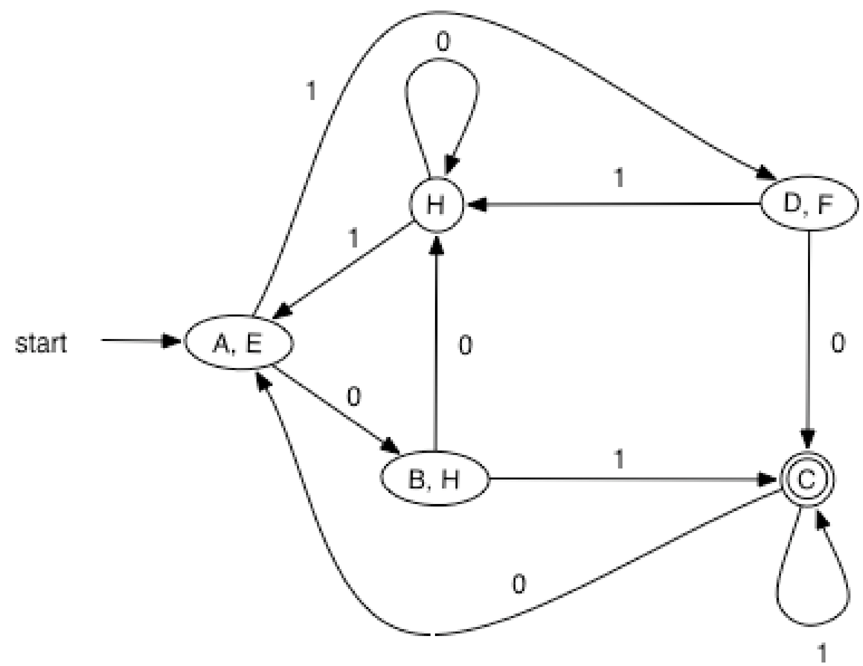

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| B | x | | | | | | |
| C | x | x | | | | | |
| D | x | | x | | | | |
| E | | | x | | | | |
| F | | | x | | | | |
| G | | | x | | | | |
| H | | | x | | | | |
| | A | B | C | E | D | F | G |




| | | | | | | | |
|---|---|---|---|---|---|---|---|
| B | x | | | | | | |
| C | x | x | | | | | |
| D | x | x | x | | | | |
| E | ☐ | x | x | x | | | |
| F | x | x | x | ☐ | x | | |
| G | x | x | x | x | x | x | |
| H | x | ☐ | x | x | x | x | x |
| | A | B | C | E | D | F | G |

Equivalent
{A, E}
{B, H}
{D, F}


start
A
B
C
D
E
F
G
H


start
A, E
B, H
H
D, F
C

# Complexity

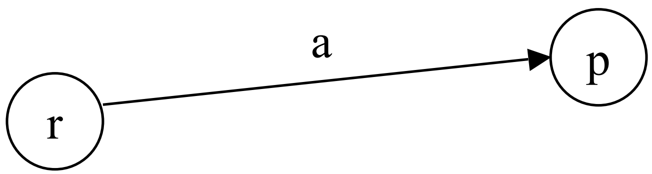
$\binom{n}{2}$ pairs of states

$\binom{n+1}{2}$ iterations of the main loop

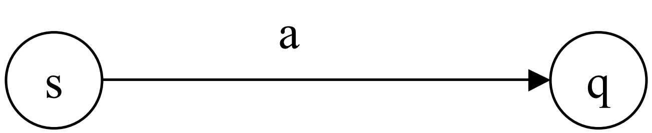
$\Rightarrow \mathbf{n^4}$

# Reducing Complexity

- **Associate with each pair of states {*r*, *s*}, a list of pairs {*p*, *q*} such that *p* and *q* must be distinguishable if *r* and *s* are,**




**Put {*r*, *s*} on the list for {*p*, *q*}**




**Start with {*p*, *q*} where *p* is final and *q* non-final**

# Minimization — Mark_Partition

*Annotate with* mark *all states reachable form the states in* class *over arcs with* label.

```
Mark_partition(class, label, mark) =
  for state in class do
      for arc in state.arcs do
          if arc.label=label then
              a.destination.mark=mark
```

# Minimization — Marked? & Final?

Return true iff state is marked 1

Marked?(state) =

state.mark=1

Make functions for these so that they can be passed as arguments to other functions.

Return true iff state is final

Final?(state) =

state.final

# Minimization — Split

Return a new partition containing states removed from *partition*. The moved states are either those of which *predicate* is true or those of which it is false, whichever gives the smaller new partition.

```
Split(class, predicate) =
  i:=0; new:={};
  for state in class do
      if predicate(state) then i := i+1;
  p = ( i < |class|/2) ;
  for state in class do
      if predicate(state)=p then
        new := new ∪ delete(state, class)
return new
```

States are moved to the smaller class. The old class becomes the larger member of the new pair

# Minimization — 3

```
Minimize(FSM) =
  push(active, Split(FSM.states, final?)) ;
  while p1 := pop(active) do
        push(inactive, p1);
        for label in Σ do
              Mark_partition(p1, label, 1)
              for p2 in active ∪ inactive do
                    push(active, Split(p2, Marked?))
              Mark_partition(p1, label, 0)
```

# Membership

**Member(string, FSM) =**
**~Empty(Intersect(FSM, StringToFsm(string)))**

# Membership for complete, deterministic, ε-free FSMs

```
Member(string, FSM) =
  state := FSM.start;
  for pos := 1 to length(string) do
      a := a in state.arcs such that
                  a.label=string[pos]
      state := a.destination
      else return false
  return state.final
```

*Linear in length of string*

# Membership for pruned, deterministic, ε-free FSMs

```
Member(string, FSM) =
  state := FSM.start;
  for pos := 1 to length(string) do
      if there is a in state.arcs such that
                a.label=string[pos]
      then state := a.destination
      else return false
  return state.final
```

# Membership for arbitrary FSMs

Member(string, FSM) = M(string, 1, FSM.start)

M(string, pos, state) =

**if** pos > length(string)

**then return** ∃ s in Epsilon-closure(state) such that s.final

**for** a **in** EC-Arcs(state.arcs) **do**

**if** a.label = string[pos] & M(string, pos+1, a.destination)
**then return** true

**return** false

*Recursive because backtracking*

# Pair Membership for arbitrary FSTs

```
PMember(string1, string2, FST) = PM(string1, 1, string2, 1, FST.start)

PM(s1, p1, s2, p2, state) =
  if p1 > length(s1) & p2 > length(s2)
    then return ∃ s in Epsilon-closure(state) such that s.final
  for a in EC-Arcs(state.arcs) do
    if (a.label.1 = ε & a.label.2=s2[p2] & PM(s1, p1, s2, p2+1, a.destination)
or
      (a.label.1 = s1[p1] & a.label.2= ε & PM(s1, p1+1, s2, p2, a.destination) or
      (a.label.1 = s1[p1] & a.label.2 = s2[p2] &
        PM(s1, p1+1, s2, p2+1, a.destination)))
    then return true
  return false
```

Closure over ε:ε

# Empty

**Empty(ID(StringToFSM(s1)) ° FSM ° ID(StringToFSM(s2)))**

# Image and Inverse Image

**Image(string, FST) =**

**Range(Compose(ID(StringToFSM(String)), FST))**

**InverseImage(string, FST) =**

**Domain(Compose(FST, ID(StringToFSM(string)))) =**

**Image(string, Inverse(FST))**

# Image

```
Image(string, FST) =
  results={},
  Im (string, 1, "", 1, FST.start)
  return results

Im(s1, p1, s2, p2, state) =
  if p1 > length(s1) & ∃ s in Epsilon-closure(state) such that s.final
    then push(results, CopyString(s2, p2));
  for a in EC-Arcs(state.arcs) do
    if a.label.1 = ε then s2[p2] := a.label.2; Im(s1, p1, s2, p2+1,
a.destination);
    else if a.label.1 = s1[p1] then
      if a.label.2= ε then Im(s1, p1+1, s2, p2, a.destination)
      else  s2[p2]:=a.label.2; Im(s1, p1+1, s2, p2+1, a.destination)
```

# Image and Inverse

```
Image(string, FST, inverse) =
  results={},
  Im (string, 1, "", 1, FST.start)
  return results

Im(s1, p1, s2, p2, state) =
  if p1 > length(s1) & ∃ s in Epsilon-closure(state) such that s.final
    then push(results, CopyString(s2, p2));
  for a in EC-Arcs(state.arcs) do
    if  inverse then inlab:=a.label.2, outlab := a.label.1
      else  inlab := a.label.1; outlab := a.label.2
    if inlab = ε & outlab = ε then Im(s1, p1, s2, p2, a.destination)
    else if inlab = ε then s2[p2] := outlab; Im(s1, p1, s2, p2+1,
a.destination);
              else if inlab = char(s1, p1) then
                if outlab= ε then Im(s1, p1+1, s2, p2, a.destination)
                else  s2[p2]:=outlab; Im(s1, p1+1, s2, p2+1, a.destination)
```

# Linear Bounded?

Recursive algorithm (Inverse) Image algorithms may not halt if FST is not linear bounded.

Composition algorithms halt but may produce cyclic FSMs.

Challenge: A recursive algorithm that always halts and produces output in the form of an FSM. Hint: try Traverse.