State Machines and Equivalence Checking

Testing & Verification

Dept. of Computer Science & Engg, IIT Kharagpur

Pallab Dasgupta

Professor, Dept. of Computer Science & Engg.,
Professor-in-charge, AVLSI Design Lab,
Indian Institute of Technology Kharagpur

Agenda

- Finite Automata
- Equivalence of Finite Automata
- Product of Finite Automata
- Acceptors for Finite Sequences
- Büchi Automata and acceptance of infinite sequences
- CNF Satisfiability
- Equivalence Checking
  - Combinational Equivalence Checking
    - Register Correspondence
    - Equivalence Checking of Retimed Circuits
  - Sequential Equivalence Checking
  - Equivalence and Minimization Algorithms

Finite Automaton

A finite deterministic automaton M (transducer, Mealy machine, finite state machine FSM) is a 6-tuple:

$$M = (Q, \Sigma, \Delta, \delta, \lambda, q^0)$$

where:

Q *is the finite set of states*

Σ *is the input alphabet*

Δ *is the output alphabet*

δ: **Q X** $\Sigma \rightarrow$ **Q** *is the transition function*

λ: **Q X** $\Sigma \rightarrow \Delta$ *is the output function*

q^0 *is the start state (initial state)*

If λ *is of the form* λ: **Q** $\rightarrow \Delta$, *then we have a Moore machine.*

State and Output Sequences

Path function: $\delta^*: Q \times (N \rightarrow \Sigma) \rightarrow Q$

Given an input sequence $\tilde{a}$, we have:

$\delta^*(q, \tilde{a}) := q'$ with $q^0 := q$, $q^{i+1} = \delta(q^i, a^i)$, and $q' := q^{|\tilde{a}|}$

Path output sequence: $\lambda^*: Q \times (N \rightarrow \Sigma) \rightarrow (N \rightarrow \Delta)$

Given an input sequence $\tilde{a}$, we have:

$\lambda^*(q, \tilde{a}) := \tilde{u}$ with $q^0 := q$, $q^{i+1} = \delta(q^i, a^i)$, and $u^i = \lambda(q^i, a^i)$

Automata Equivalence

Two automata M and M′ are called equivalent, if for an arbitrary input sequence applied at both automata, the same output sequence results:

$$\forall \tilde{a} \; . \; \lambda^*(q^0, \tilde{a}) = \lambda'^*(q^0, \tilde{a})$$

State Equivalence

Given two Mealy machines with the same input and output alphabet, $M = (Q, \Sigma, \Delta, \delta, \lambda, q^0)$ ***and*** $M' = (Q', \Sigma, \Delta, \delta', \lambda', q'^0)$.

The state equivalence relation $\sim \subseteq Q \times Q'$ ***is the largest relation which satisfies the following:***

$$q \sim q' : \Leftrightarrow \forall a, a \in \Sigma \ . \ \lambda(q, a) = \lambda'(q', a) \ \text{ and } \ \delta(q, a) \sim \delta'(q', a)$$

Two states q and q′ are said to be equivalent, if $q \sim q'$ ***holds.***

Results:

- It holds that $\forall \tilde{a}, \tilde{a} \in (N \rightarrow \Sigma) \ . \ \ q \sim q' \ \Rightarrow \ \ \delta^*(q, \tilde{a}) \sim \delta'^*(q', \tilde{a})$
- Two Mealy machines M and M′ are equivalent, written as $M \approx M'$, iff their initial states are equivalent: $q^0 \sim q'^0$.

State Minimization

- **Necessary and sufficient condition for two states to be equivalent:**

$$q_1 \sim q_2 \Leftrightarrow \forall a, a \in \Sigma \ . \ \lambda(q_1, a) = \lambda(q_2, a) \ \text{and} \ \delta(q_1, a) \sim \delta(q_2, a)$$

- **Equivalent states can be merged**

Product Automaton

The product automaton of two automata $M = (Q, \Sigma, \Delta, \delta, \lambda, q^0)$ *and* $M' = (Q', \Sigma, \Delta, \delta', \lambda', q'^0)$ *is defined as:*

$$M^P = (Q \times Q', \Sigma, B, \delta^P, \lambda^P, (q0, q'^0))$$

with δ^P: $(QXQ') \times \Sigma \rightarrow (QXQ')$ *and* λ^P: $(QXQ') \times \Sigma \rightarrow B$, *defined by:*

$$\delta^P((q, q'), a) := (\delta(q, a), \delta'(q', a))$$

$$\lambda^P((q, q'), a) := (\lambda(q, a) = \lambda'(q', a))$$

The product delivers only a value B which indicates whether for a given input the outputs of both automata are equal (T) *or not* (F).

Acceptors

A deterministic finite acceptor (called DFA) M^a is a 5-tuple:

$M^a = (Q, \Sigma, \delta, q^0, F)$

where:

Q is the finite set of states

Σ is the input alphabet

δ: Q X Σ → Q is the transition function

q^0 is the start state (initial state)

F $\subseteq$ Q is the set of final states (accepting states)

A finite sequence ã is said to be accepted by $M^a = (Q, \Sigma, \delta, q^0, F)$, if $\delta^*(q^0, \tilde{a}) \in F$.

Acceptance of Infinite Sequences

- **Büchi automaton:**

  An accepting Buchi automaton M^{aB} *is a 5-tuple,*

  $M^{aB} = (Q, \Sigma, \delta, q^0, F)$

  where Q is the finite set of states, Σ *is the input alphabet,* δ: $Q \times \Sigma \rightarrow Q$ *is the transition function,* q^0 *is the start state (initial state).* $F \subseteq Q$ *is the set of final states (accepting states).*

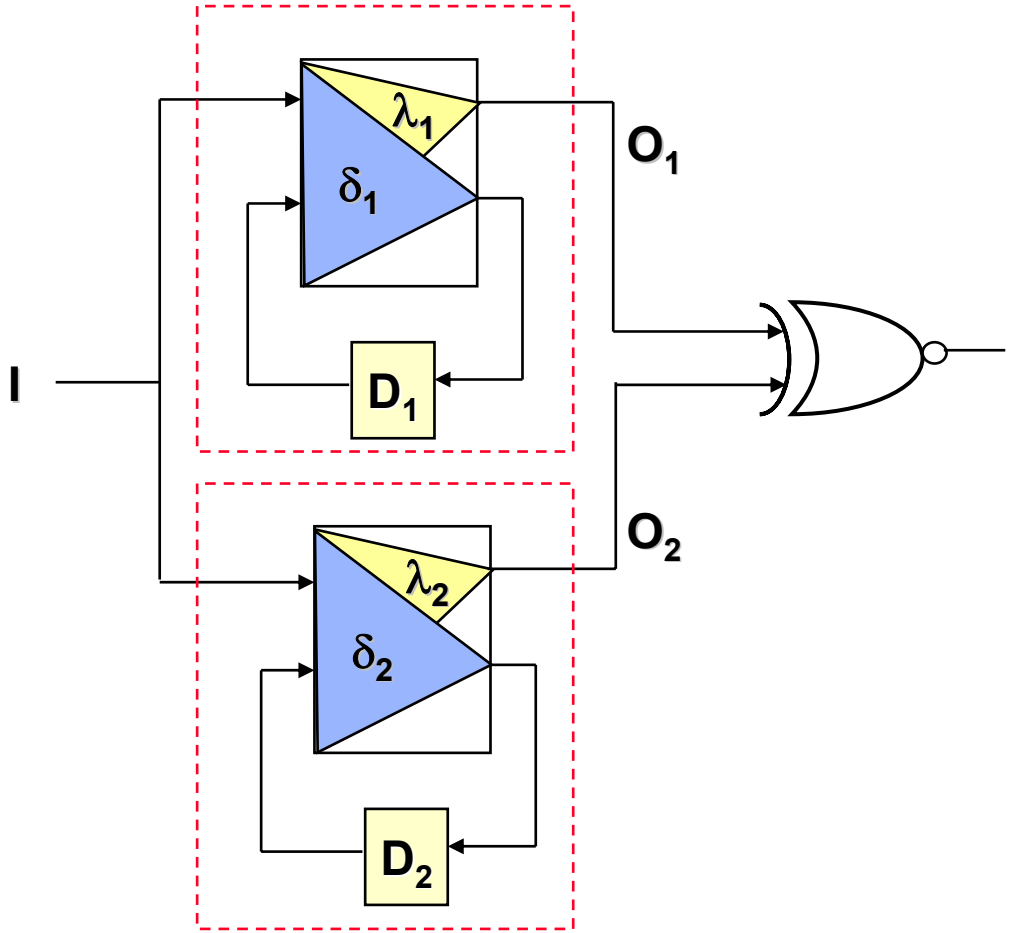
- **Büchi acceptance:**

  An infinite sequence ã is accepted by the Buchi automaton $M^{aB} = (Q, \Sigma, \delta, q^0, F)$, *if* $\forall t\, \exists t', t' > t \,.\, \delta^*(q^t, \tilde{a}^{t \ldots t'}) \in F$.

  In other words, an infinite sequence is accepted if the final set is visited infinitely often.

Equivalence Checking Problem

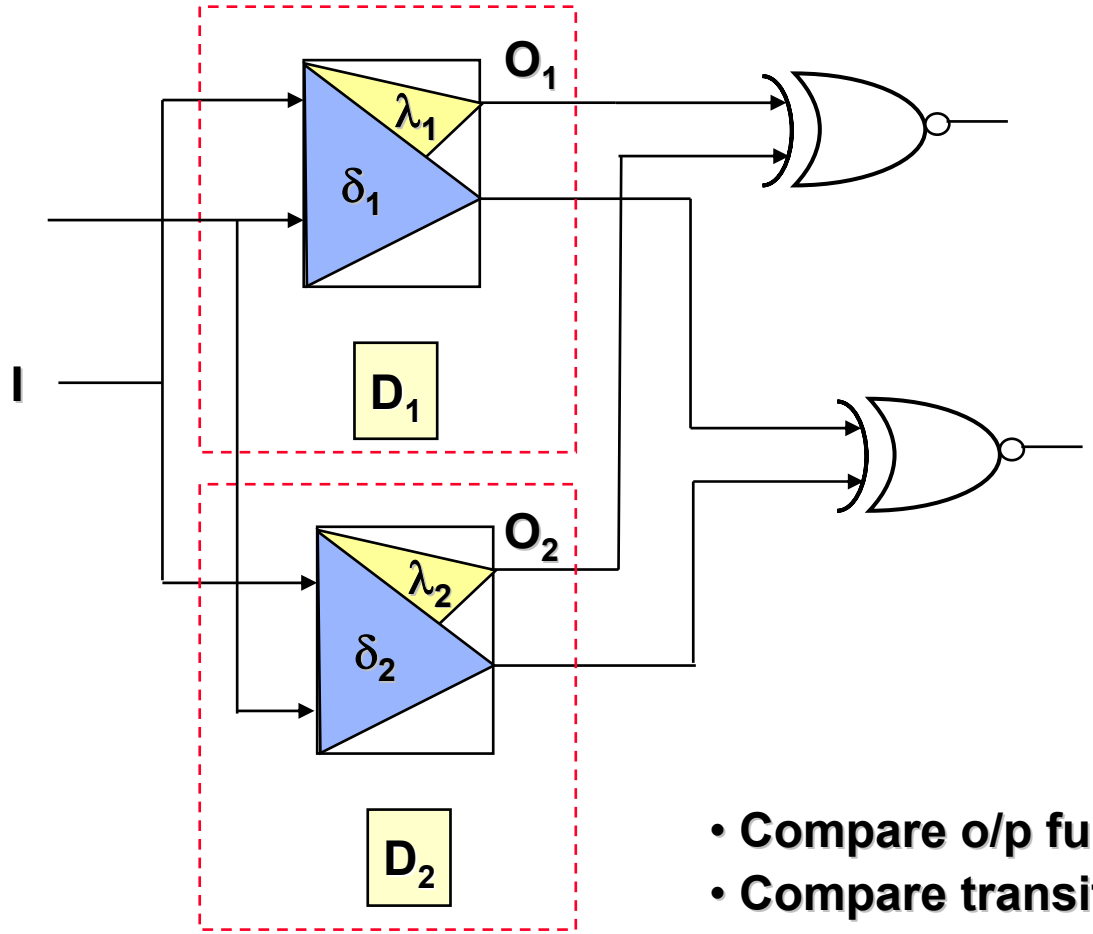
- Two designs are defined to be functionally equivalent if they produce identical output sequences for all valid input sequences



Equivalence Checking Paradigms

- **Sequential Equivalence Checking**
  - Compare state machines
- **Combinational Equivalence Checking**
  - Compare combinational Boolean functions
- **If a one-to-one correspondence between the registers is given, then sequential equivalence checking can be solved using combinational equivalence checking**
  - **This is a popular approach – very useful in practice**

Combinational Equivalence Checking



- Compare o/p function
- Compare transition function

Basic Approach

- **Step-1: Register Correspondence**
  - The register correspondence is either guessed using simple heuristics or computed exactly
- **Step-2: Functional Comparison**
  - This step involves the actual functional comparison of the individual circuits
  - This can be done using a variety of methods, including BDDs, SAT and ATPG

Regsiter Correspondence

- In many practical design flows, a candidate register correspondence is derived from naming conventions
- Otherwise, register correspondence can be computed automatically as a greatest fixed point (to be explained)
  - The algorithm starts with one equivalence class (bucket) containing all the registers
  - During each iteration:
    - A unique variable is introduced for the outputs of all registers of each bucket
    - All next state functions are computed based on these variables
    - Next the buckets are partitioned into pieces that have identical next-state functions

Register Correspondence Algorithm

REGISTER CORRESPONDENCE( ) {

put all registers r into bucket[0]

do {

forall buckets *i* do {

initialize output of all registers $r \in i$ with variable v[*i*]

}

forall registers *r* do {

compute next state function δ[r] based on inputs v

}

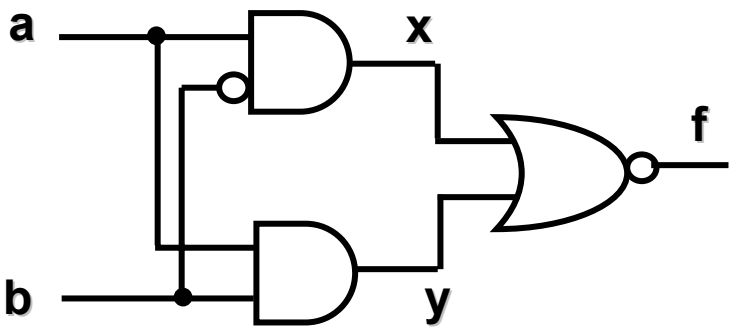
if ∀ buckets i: $r_1, r_2 \in i \Leftrightarrow \delta[r_1] = \delta[r_2]$ return

split all buckets *i* into multiple buckets i_j s.t. $r_1, r_2 \in i_j \Leftrightarrow \delta[r_1] = \delta[r_2]$

}

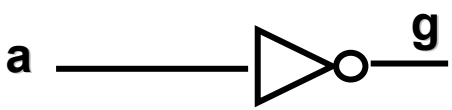
}

Equivalence Checking with CNF-SAT



Clauses:

$(a \vee \neg y), (b \vee \neg y), (\neg a \vee \neg b \vee y),$
$(a \vee \neg x), (\neg b \vee \neg x), (\neg a \vee b \vee x),$
$(\neg x \vee \neg f), (\neg y \vee \neg f), (x \vee y \vee f)$



Clauses:

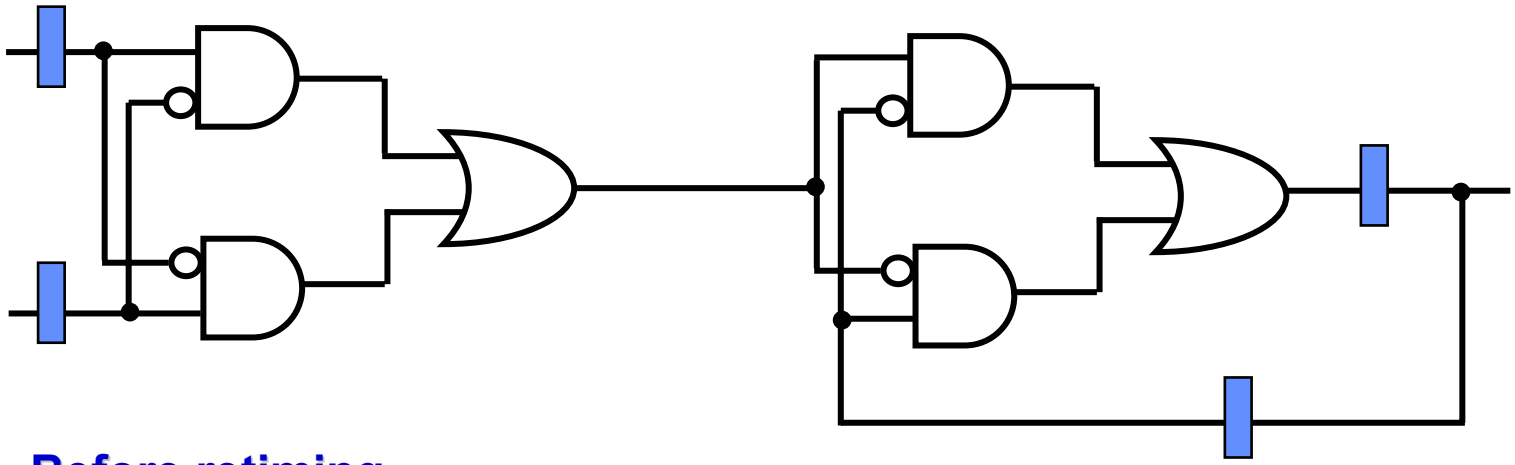
$(a \vee g), (\neg a \vee \neg g)$

To check equivalence between *f* and *g*, we add the following clauses:

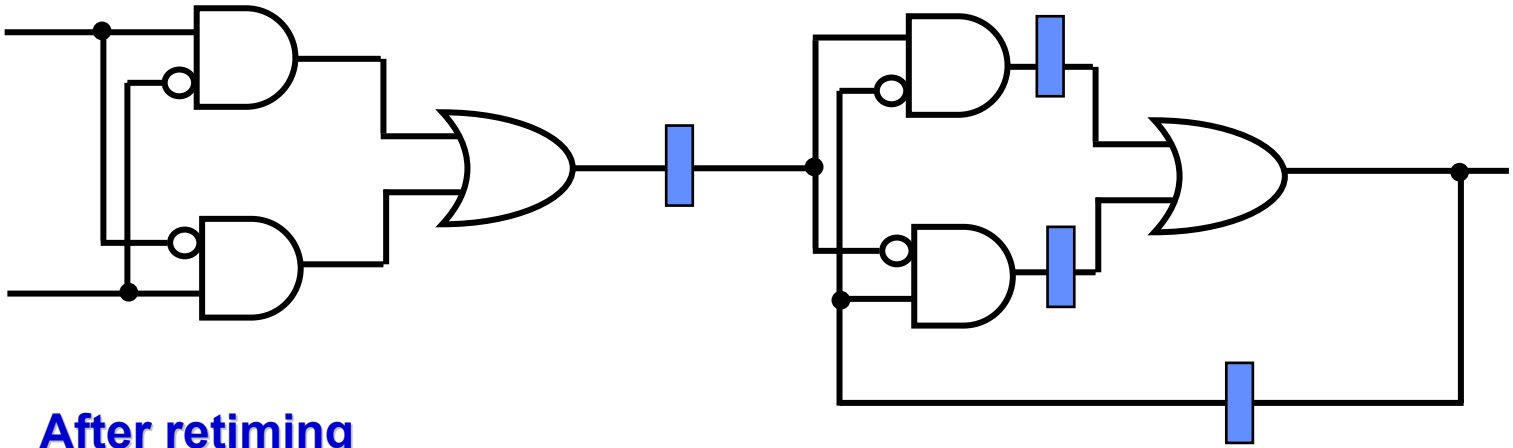
$(f \vee g), (\neg f \vee \neg g)$

which is the EXOR between f and g. If the set of clauses is satisfiable, then we have a valuation of *a* and *b* such that *f* and *g* receive conflicting values. Otherwise (as in this case), *f* and *g* are equivalent.

Retiming and Equivalence Checking



Before retiming



After retiming

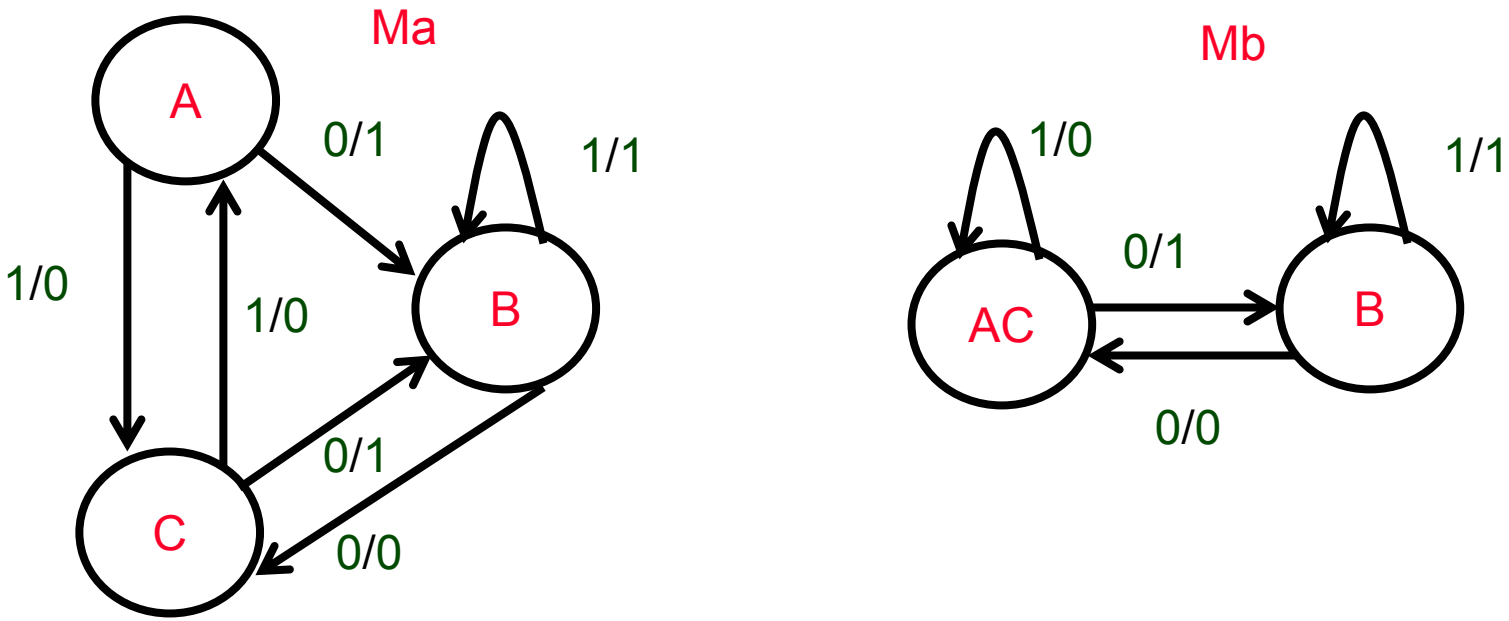
Equivalence Checking of Retimed Logic

- In case of retiming, the next-state functions are not comparable
  - However, by preserving the retime logic from the synthesis flow and applying it to make both designs comparable, the equivalence checking problem can be reduced to a combinational problem
  - Both machines are patched with pieces of the retime logic to make the interfaces comparable

Sequential Equivalence Checking

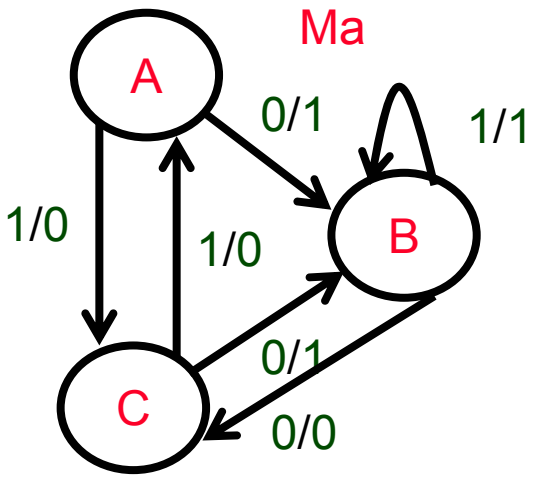
- When register correspondence cannot be found easily or it does not exist, we may compare the state machines
- Basic approach
  - Core problem: *Partition the state space into sets of equivalent states*
  - Equivalence can be defined in terms of input/output behavior
    - Bisimulation equivalence
    - Stuttering equivalence

Redundant States and Minimization



A or C is redundant state

Definitions



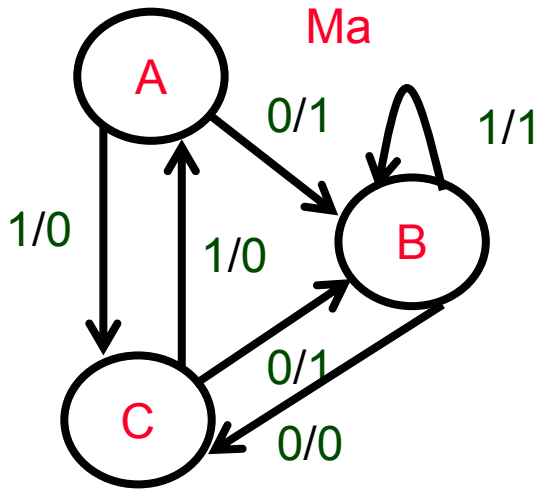
If an input sequence X takes a machine from a state Si to Sj, then Sj is said to be the X-successor of Si.

B is 110-succesor of A

Two states Si and Sj are distinguishable iff there exists at least one finite input sequence which when applied to M, causes different output sequences, depending on whether Si or Sj is the initial state.

A and B are distinguishable. Consider input sequence 0.

k-distinguishable states



If there exists for pair (Si,Sj), a distinguishing sequence of length k, the states in (Si,Sj) are said to be k-distinguishable.

States that are not k-distinguishable are called k-equivalent.

A, B are 1-distinguishable

A, C are not 2-distinguishable and hence are 2-equivalent

States Si and Sj are said to be equivalent iff for every possible input sequence, the same output sequence is produced regardless of whether Si or Sj is the initial state.

A, C are equivalent

The State Minimization Problem

Input : state machine M

Output : minimize (M), the state machine with the fewest states that is equivalent to M

Two machines Mi and Mj are equivalent iff, for every state in Mi, there is a corresponding equivalent state in Mj and vice versa.

The Minimization Procedure

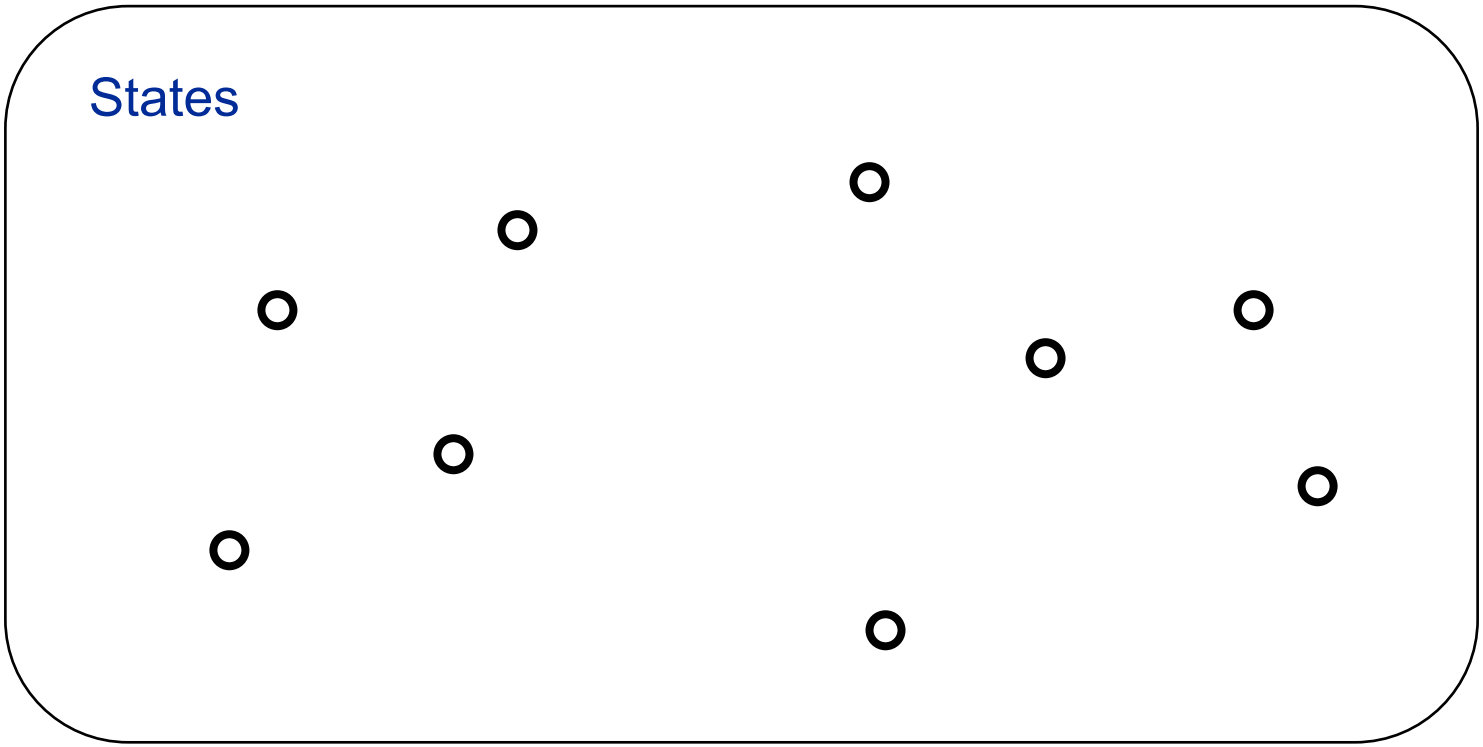
1. Partitions states of M into subsets such that all states in the same subset are 1-equivalent: P_1

2. Partitions states of M into subsets such that all states in the same subset are 2-equivalent: P_2

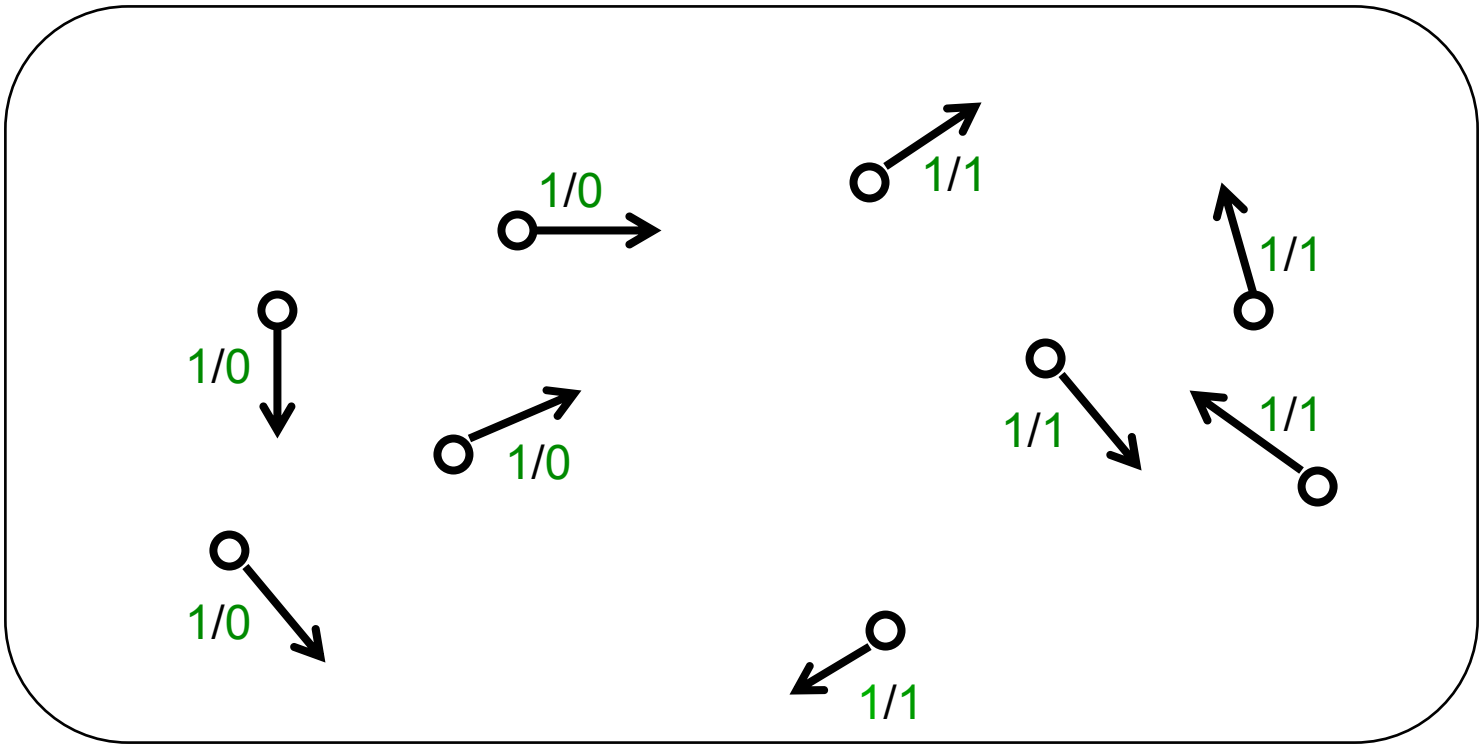
…

Until for some k, $P_{k+1} = P_k$

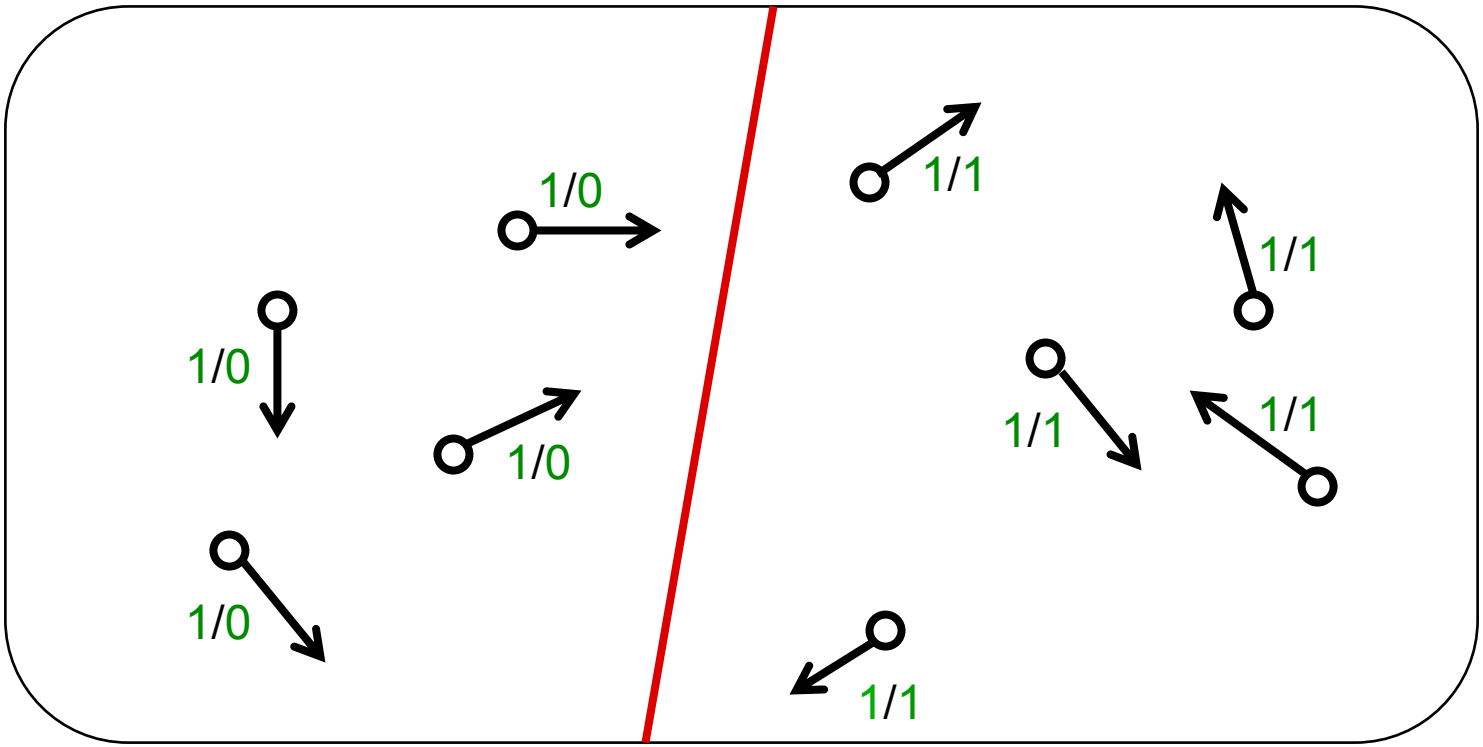
The Minimization Procedure



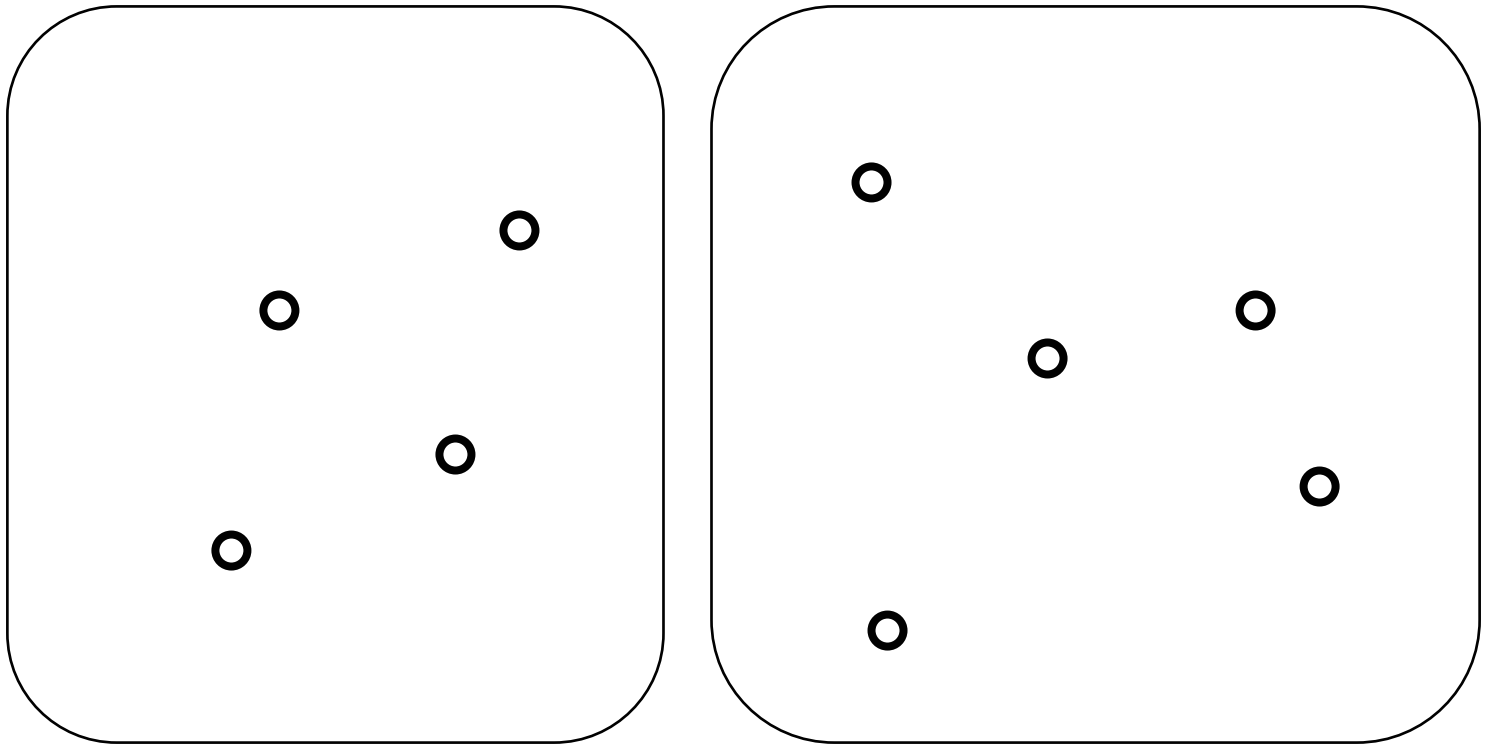
The Minimization Algorithm



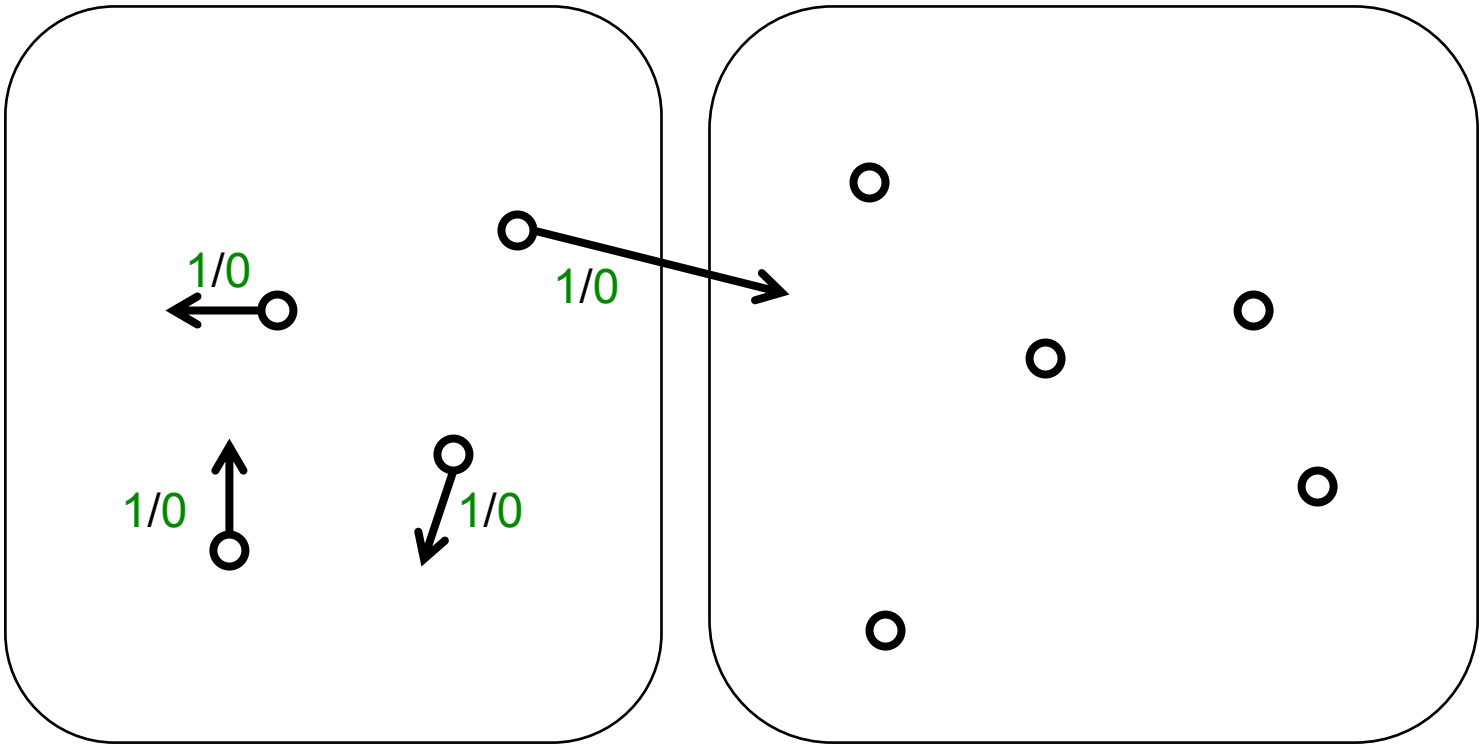
The Minimization Algorithm



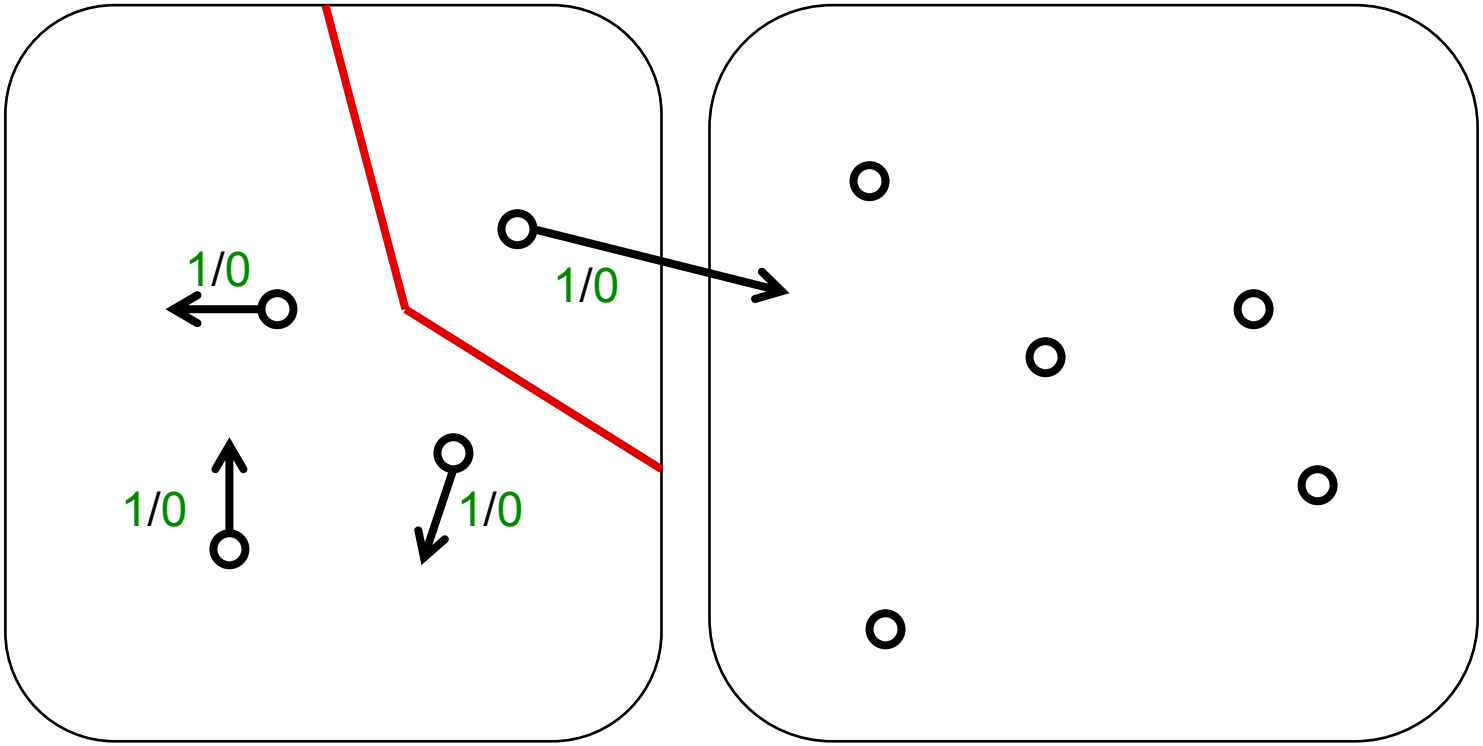
The Minimization Algorithm



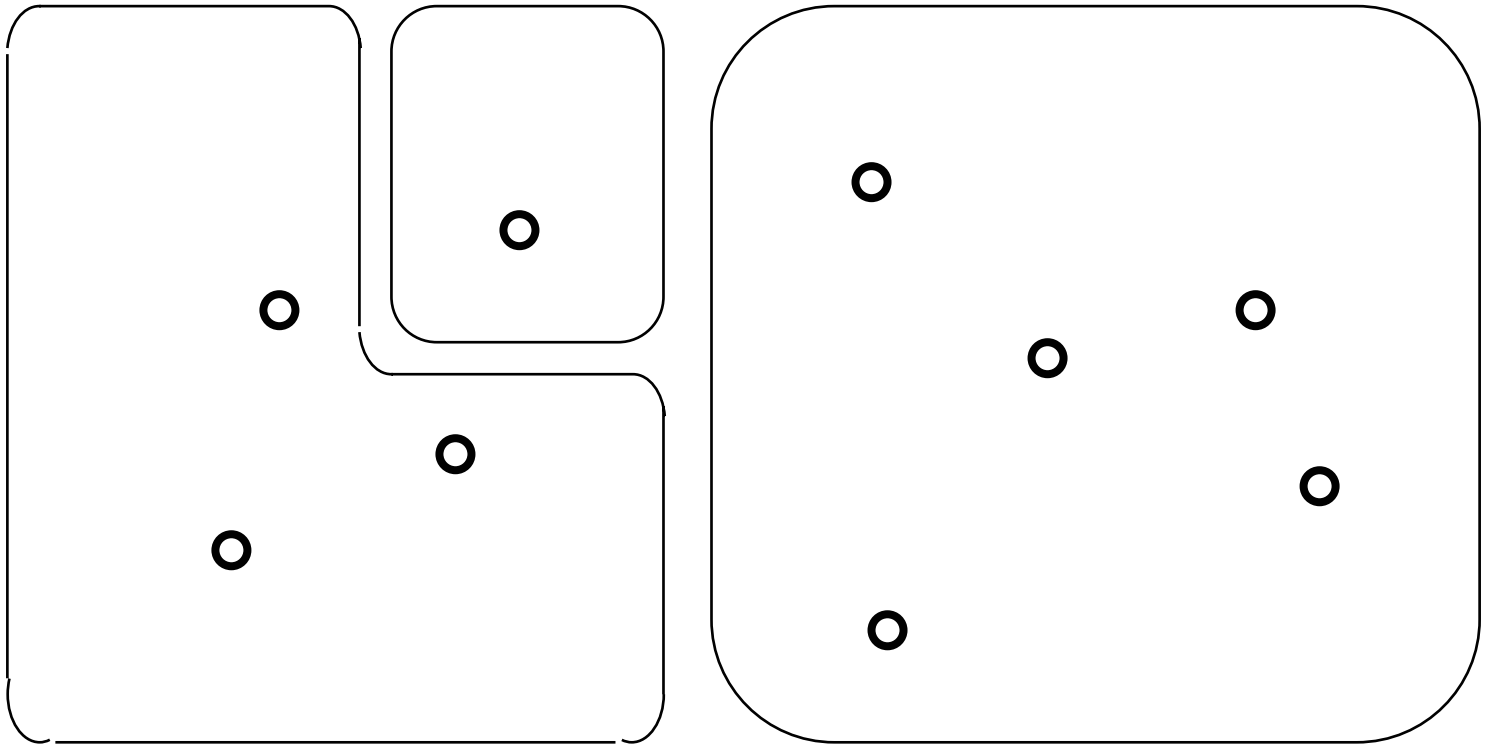
The Minimization Algorithm



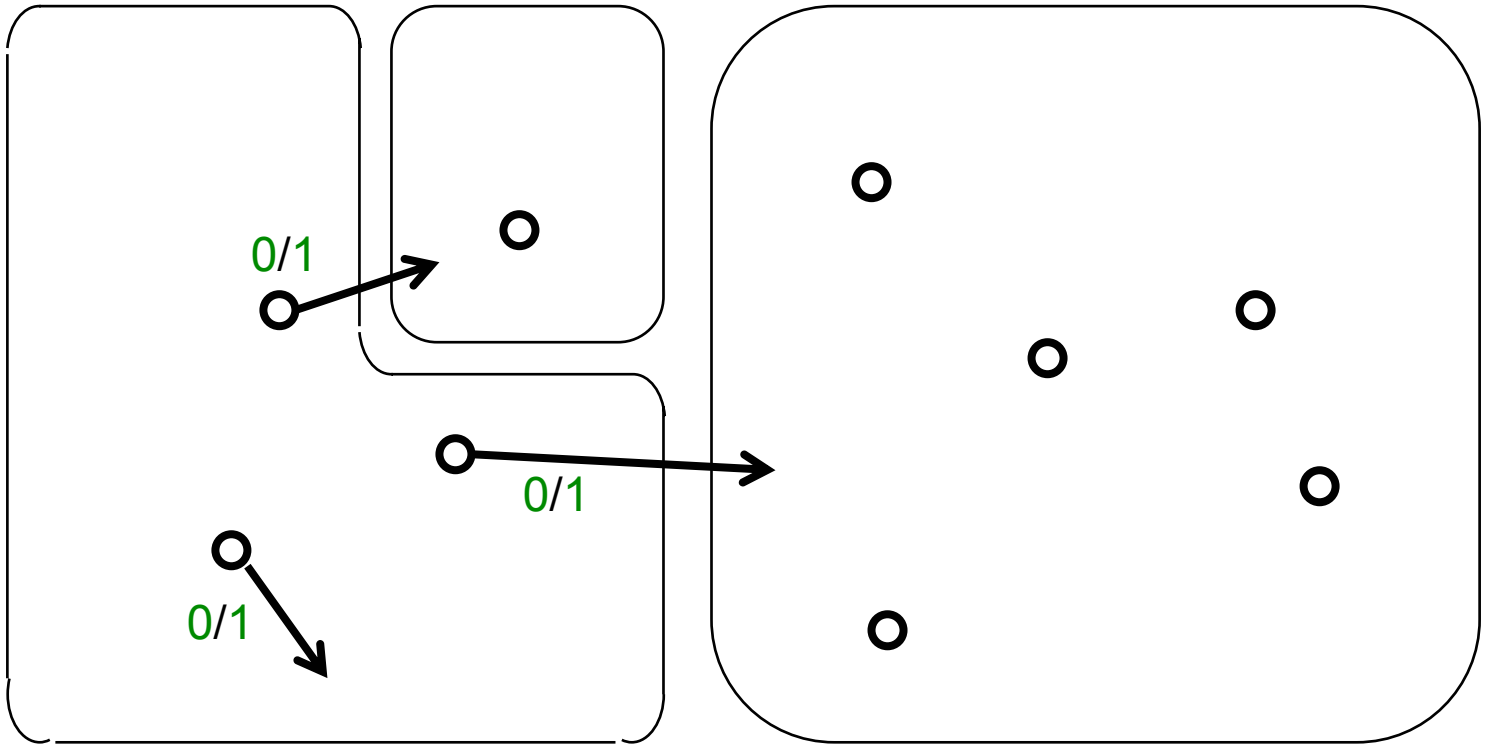
The Minimization Algorithm



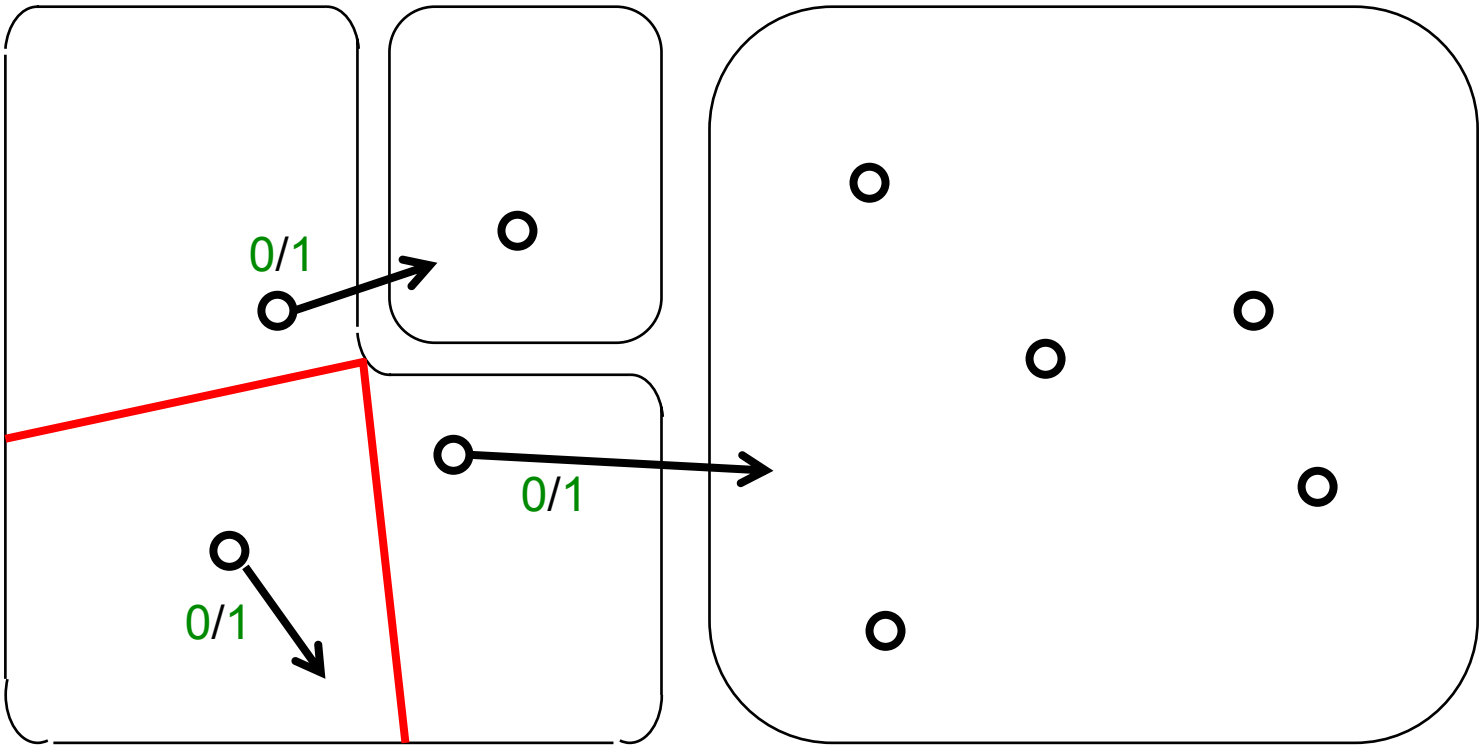
The Minimization Algorithm



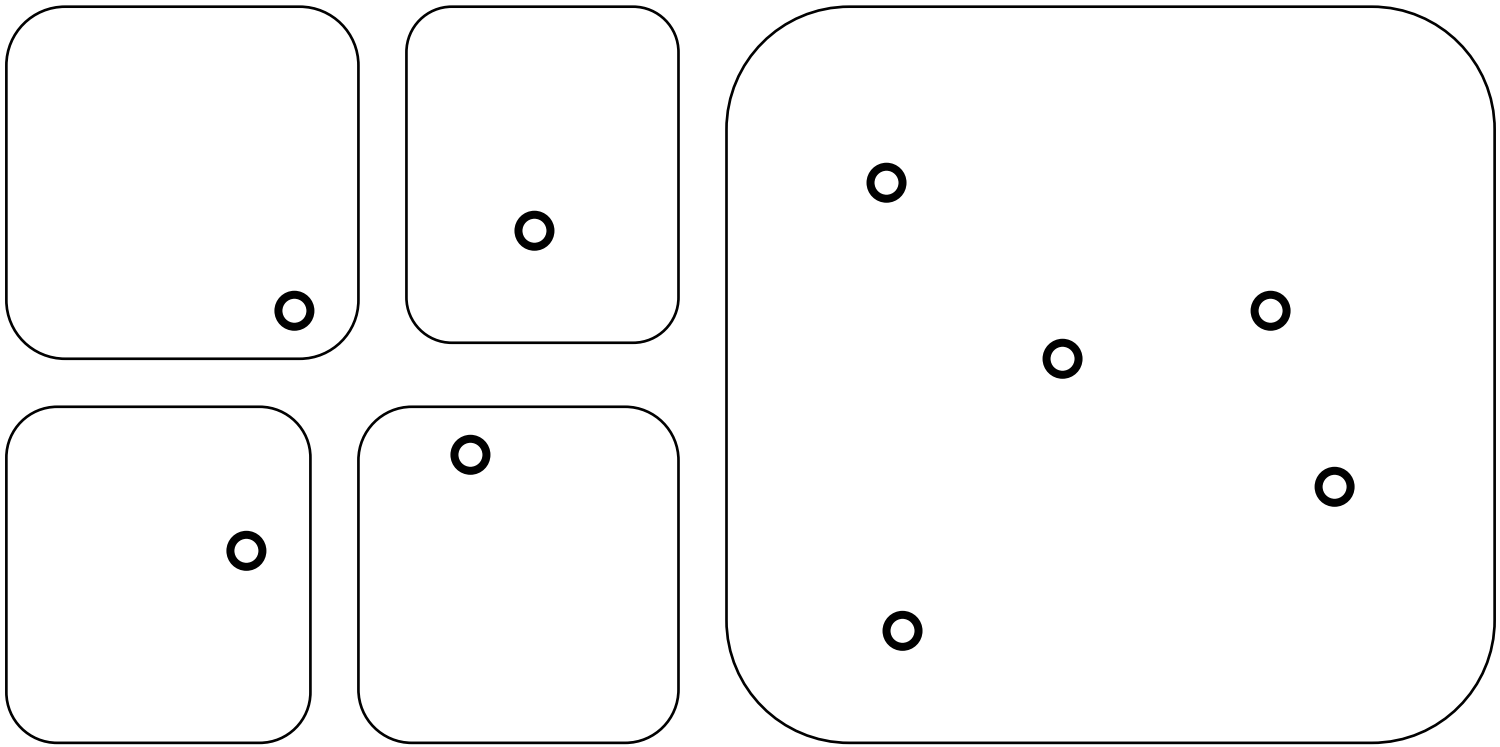
The Minimization Algorithm



The Minimization Algorithm



The Minimization Algorithm



The Minimization Algorithm

1. Let Q be set of all reachable states of M.
2. Maintain a set P of state sets:

   Initially let P = { Q }.

   2a. Repeat until no longer possible: output split P.

   2b. Repeat until no longer possible: next-state split P.
3. When done, every state set in P represents a single state of the smallest state machine equivalent to M.

Output split P

If there exist

a state set $R \in P$

two states $r1 \in R$ and $r2 \in R$

an input $x \in$ Inputs

such that

$output(r1, x) \neq output(r2, x)$

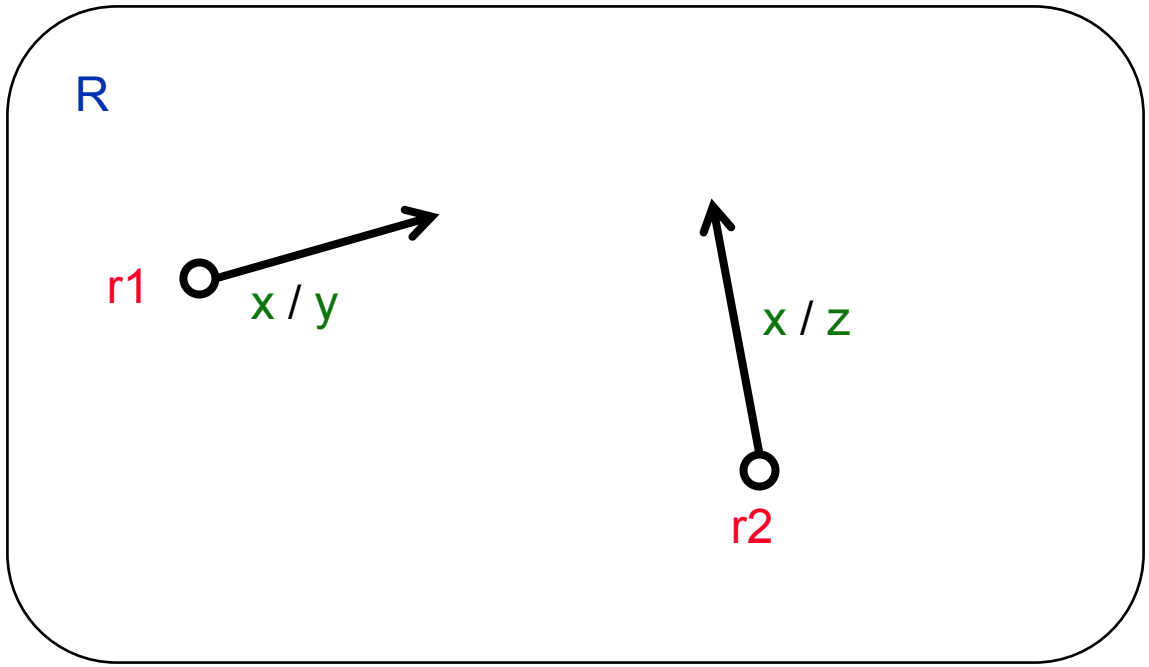
then

let $R1 = \{ r \in R \mid output(r,x) = output(r1,x) \}$;

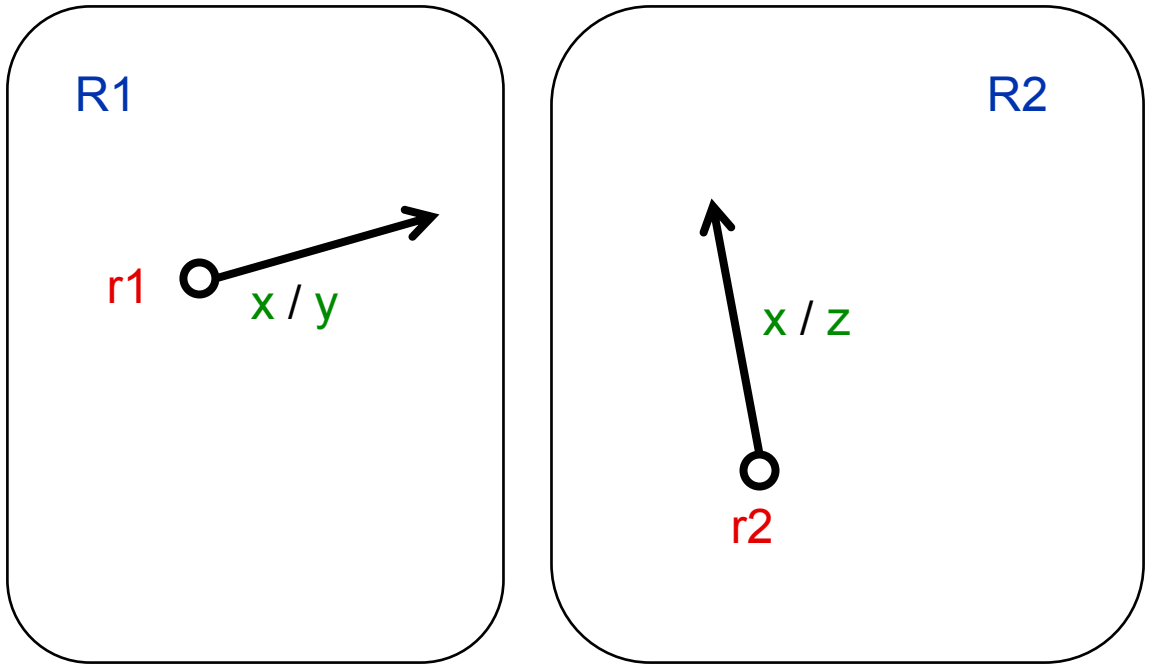
let $R2 = R \setminus R1$;

let $P = (P \setminus \{R\}) \cup \{R1, R2\}$.

Output split



Output split



Next-state split P

If there exist

two state sets $R \in P$ and $R' \in P$

two states $r1 \in R$ and $r2 \in R$

an input $x \in$ Inputs

such that

$nextState(r1, x) \in R'$ and $nextState(r2, x) \notin R'$

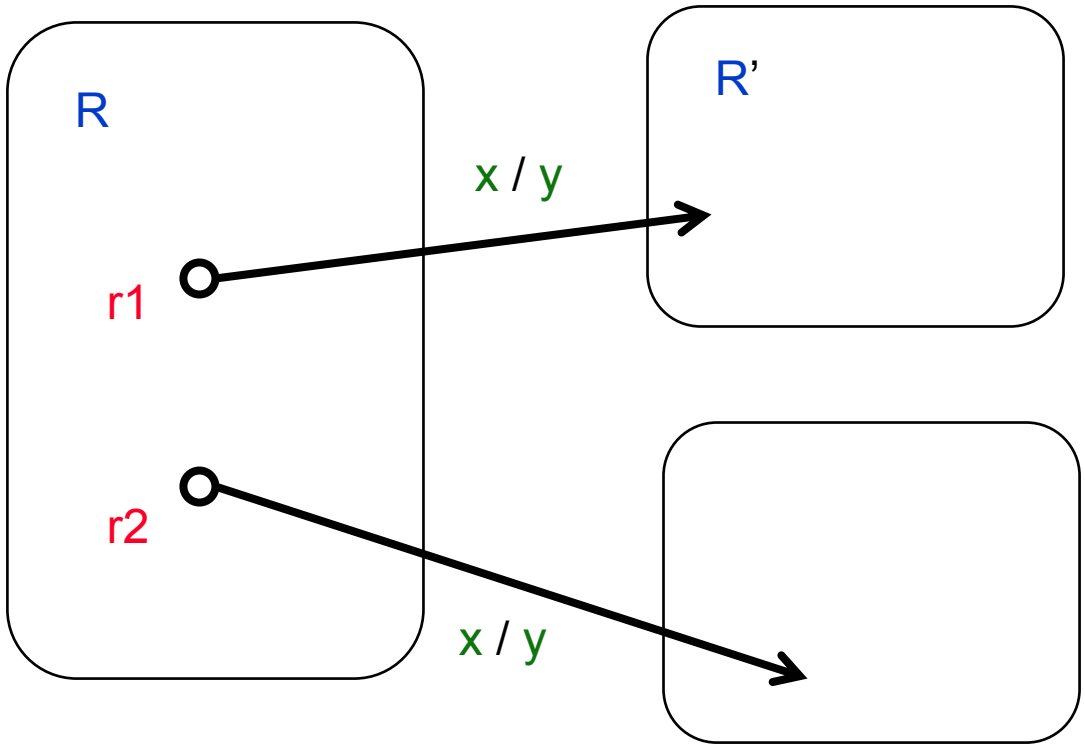
then

let $R1 = \{ r \in R \mid nextState(r,x) \in R' \}$;

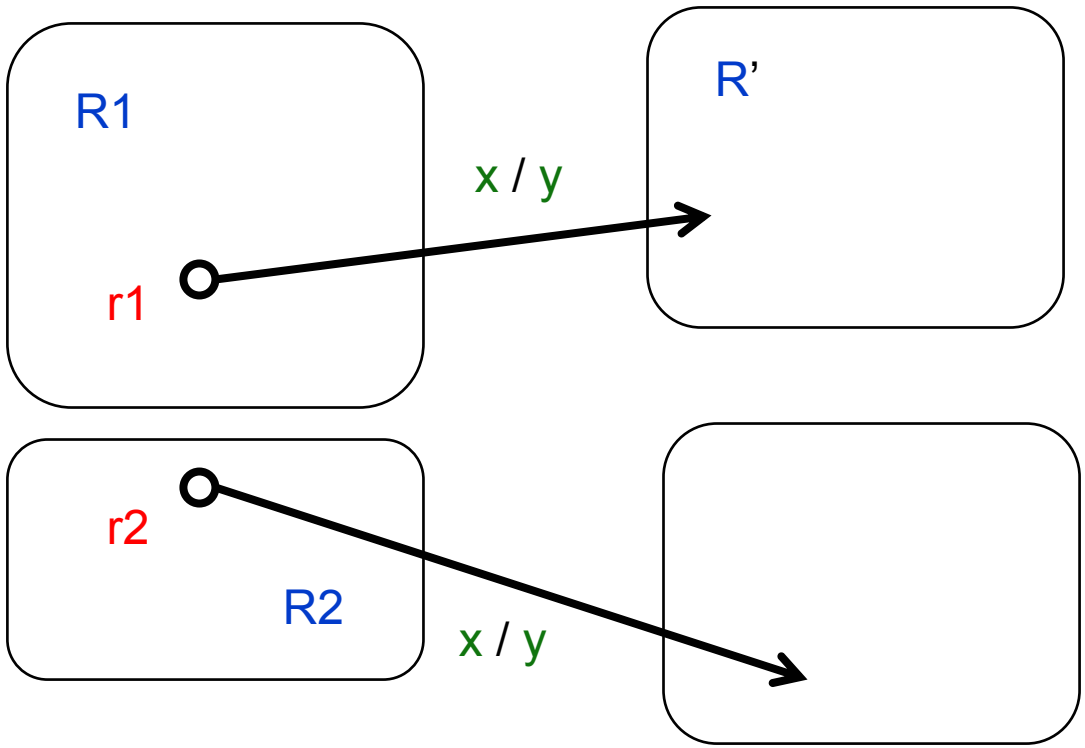
let $R2 = R \setminus R1$;

let $P = (P \setminus \{ R \}) \cup \{ R1, R2 \}$.

Next-state split

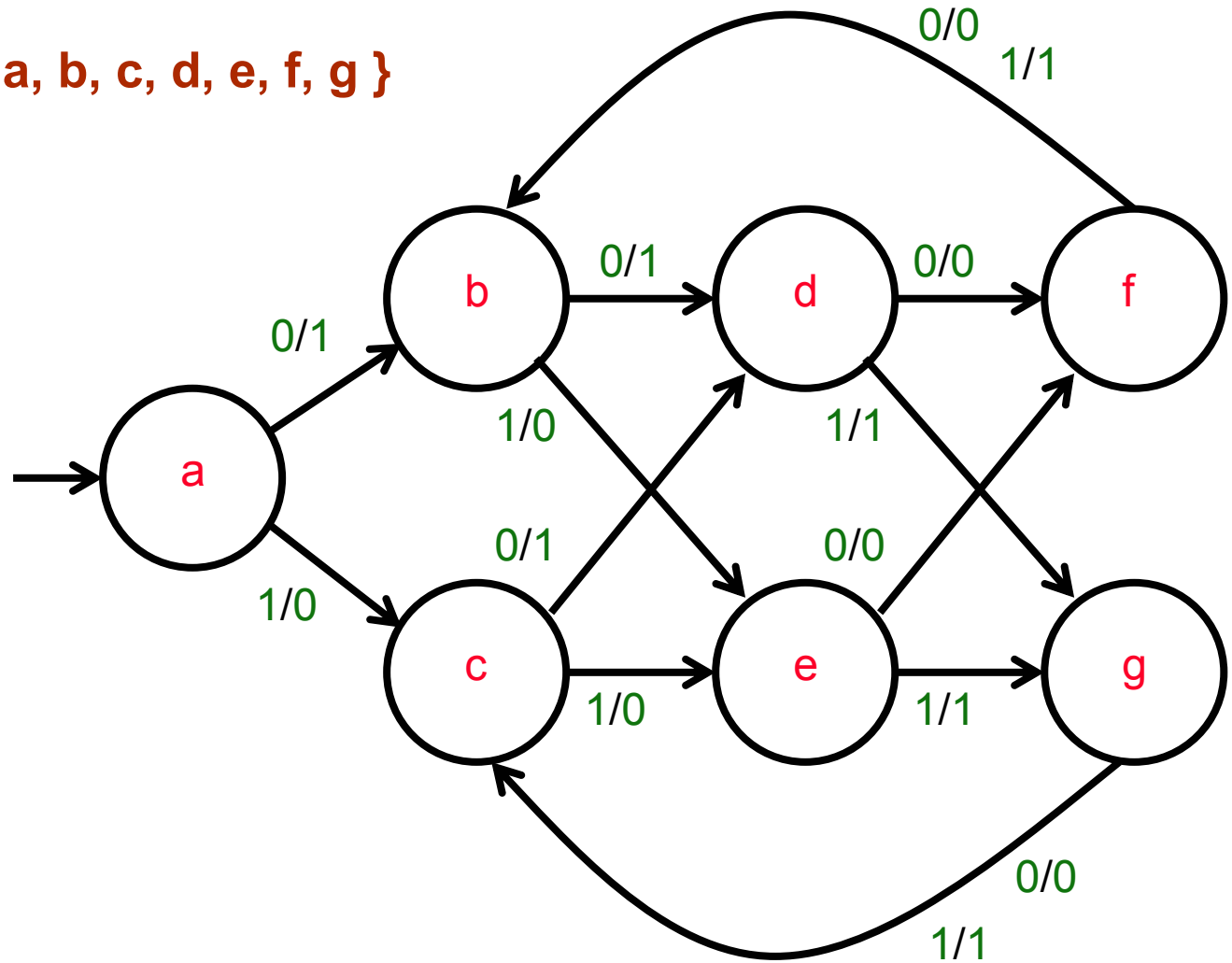


Next-state split

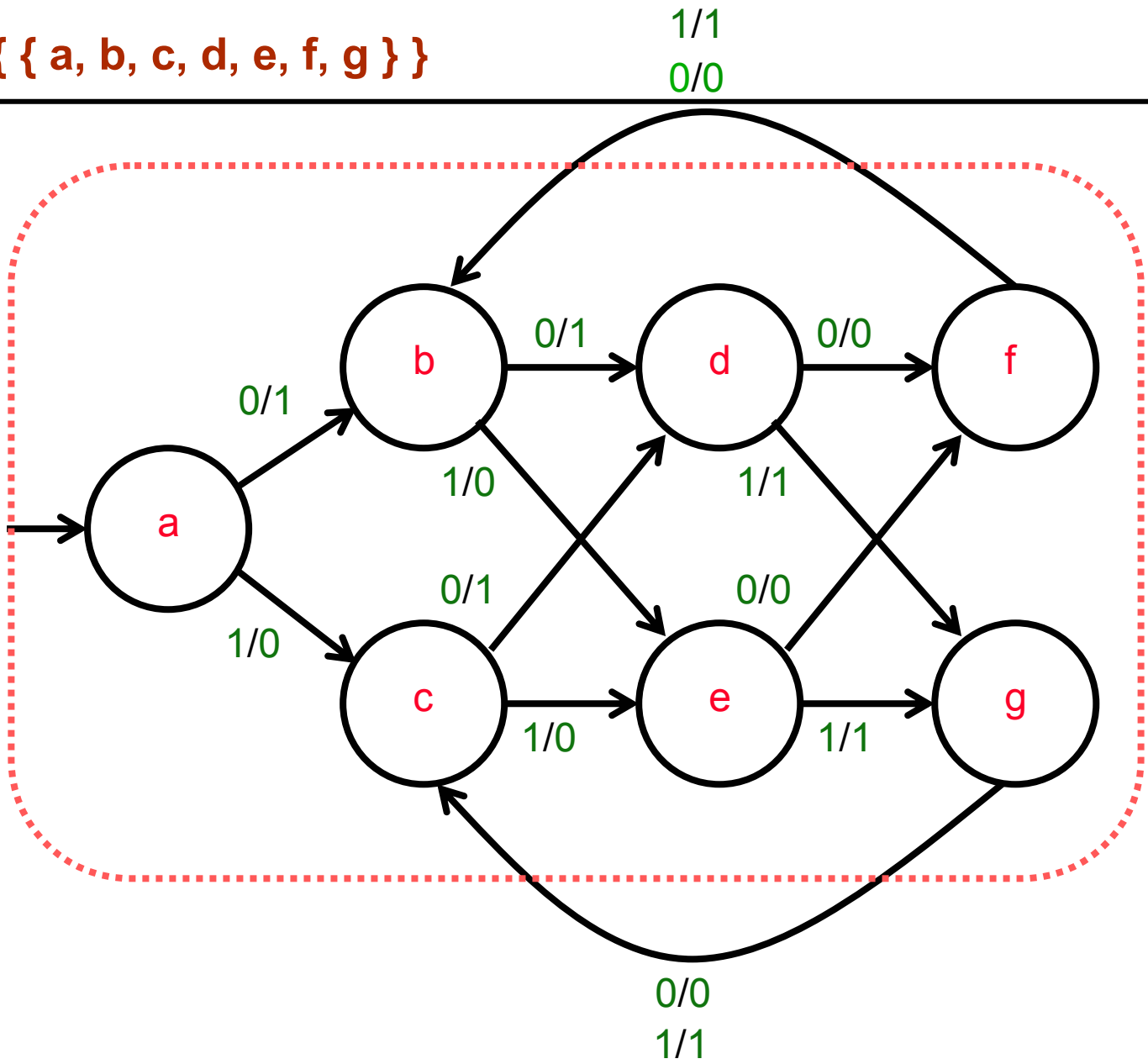


Example

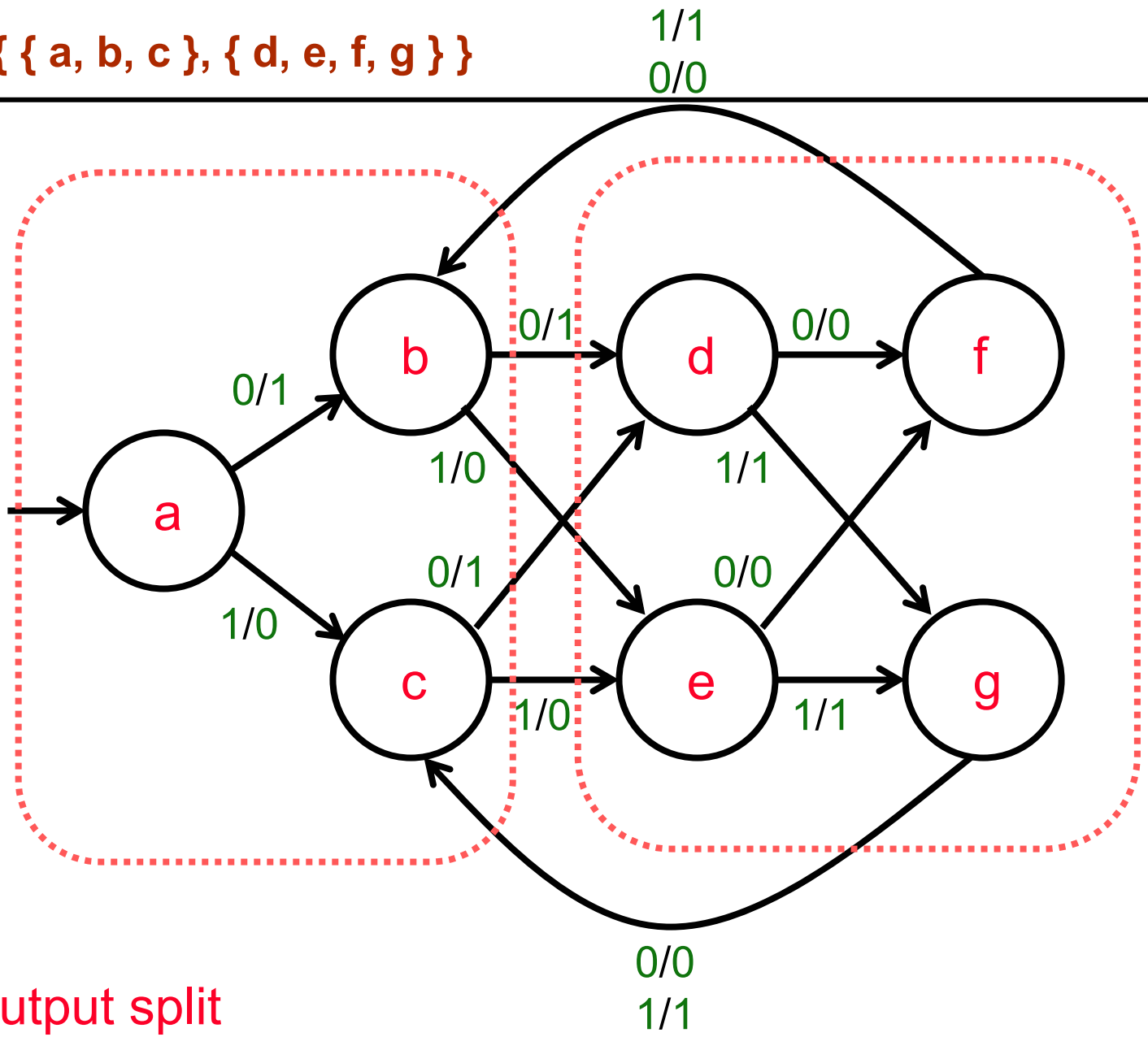
Q = { a, b, c, d, e, f, g }



P = { { a, b, c, d, e, f, g } }

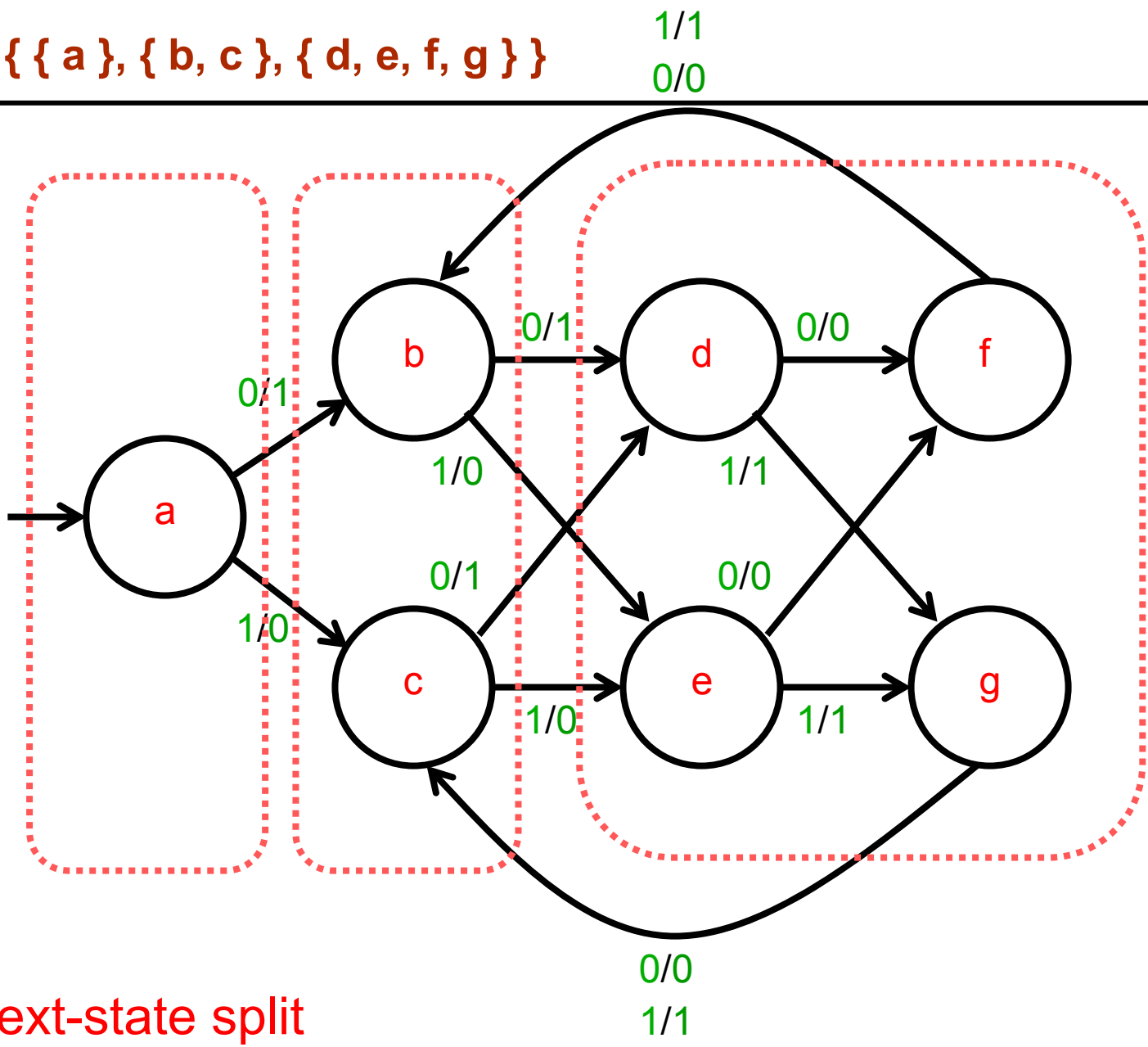


P = { { a, b, c }, { d, e, f, g } }



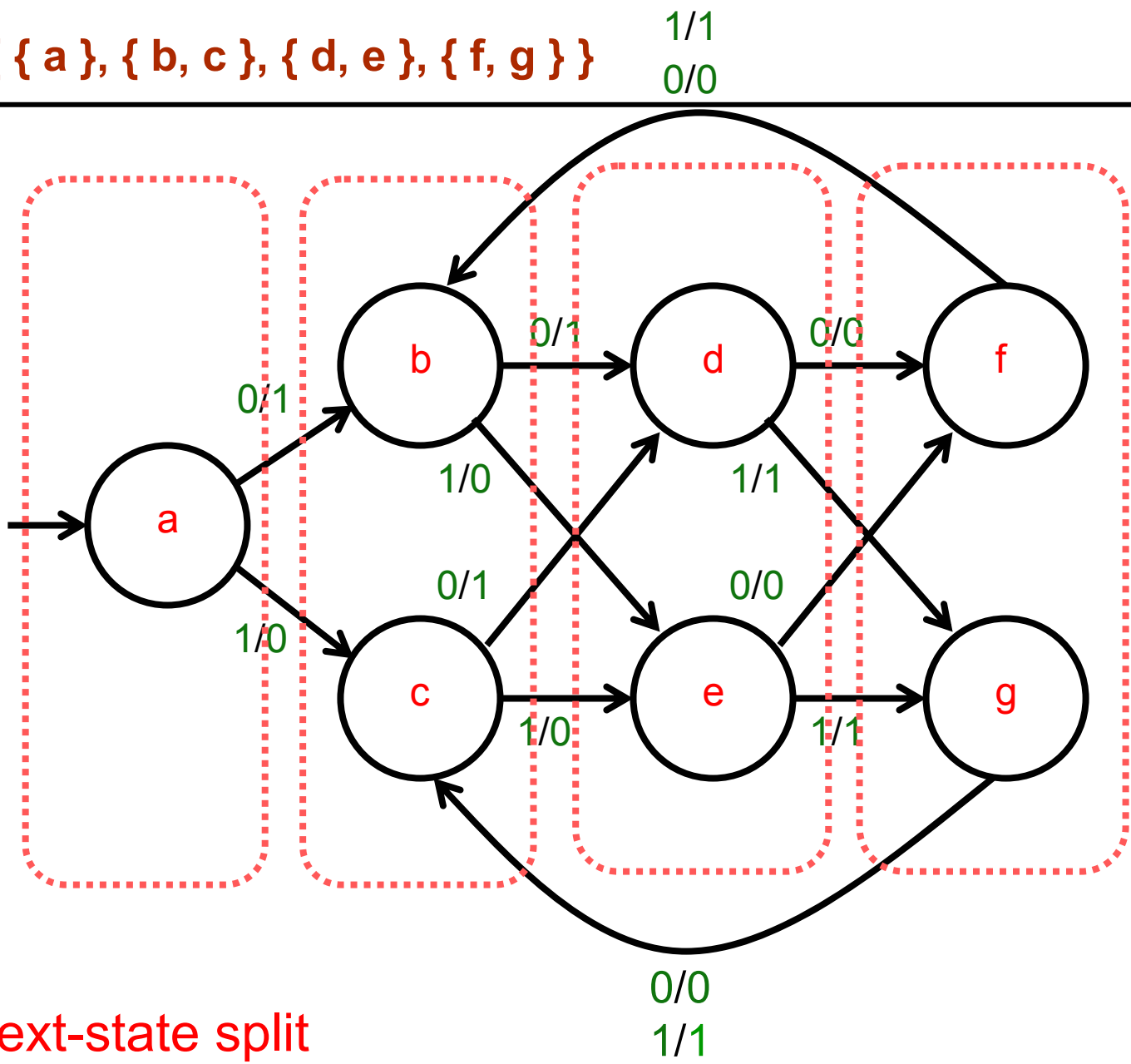
Output split

P = { { a }, { b, c }, { d, e, f, g } }

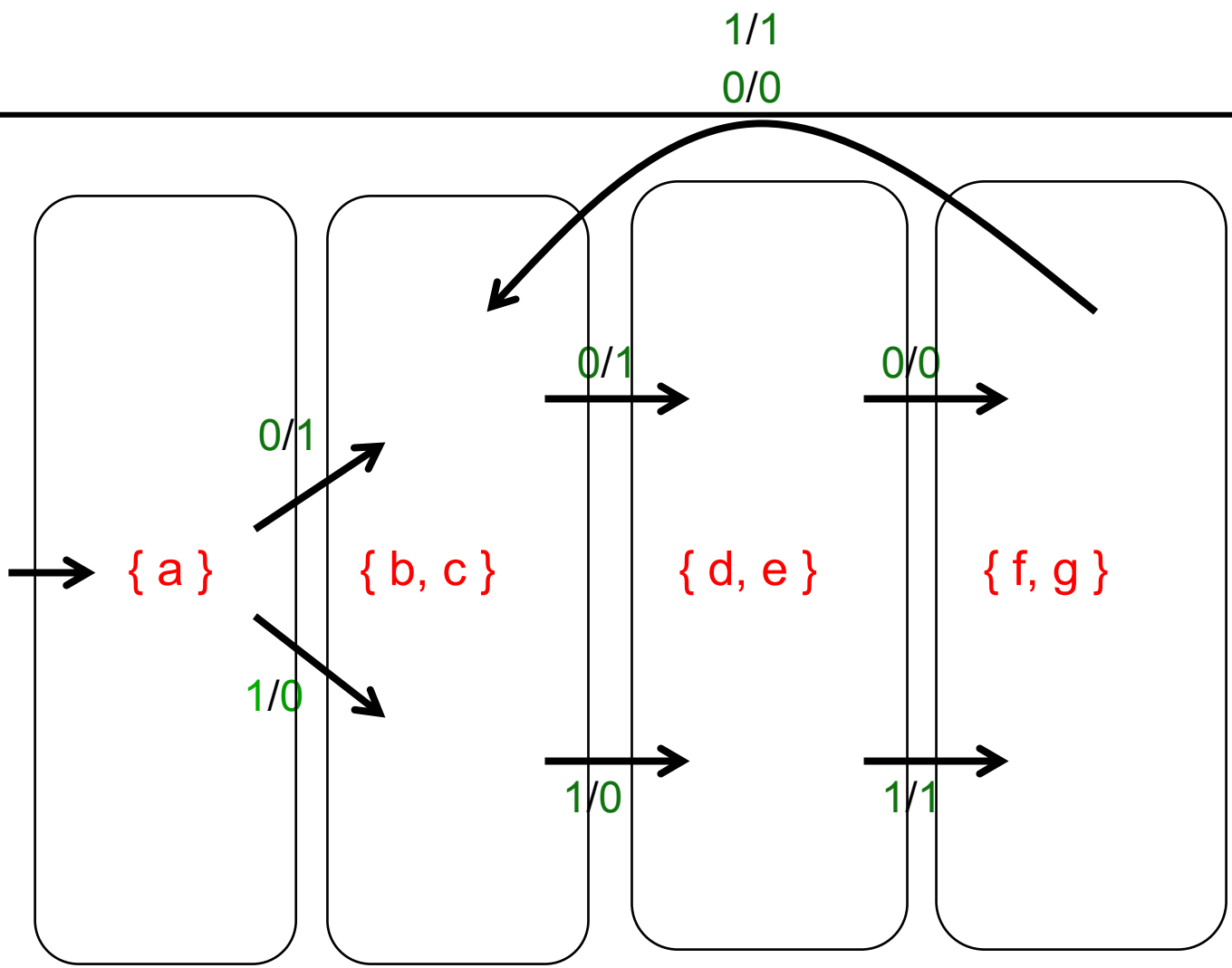


Next-state split

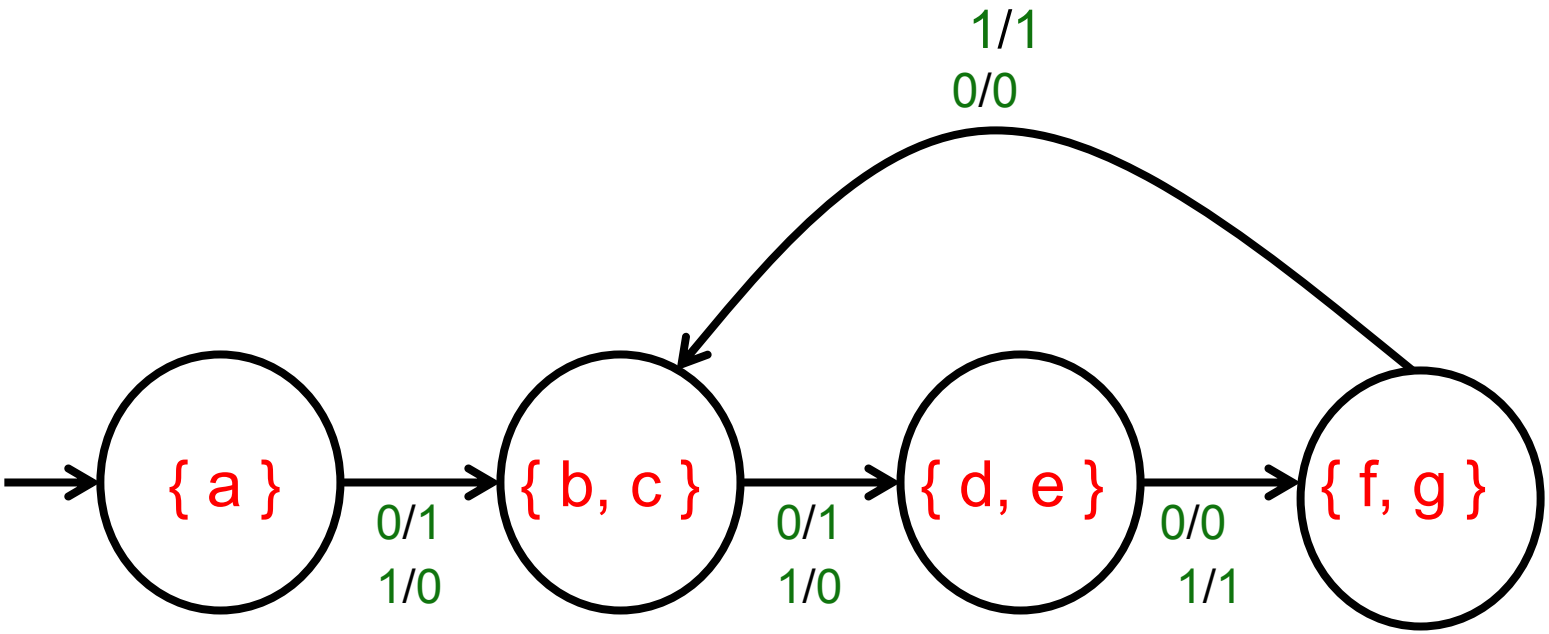
P = { { a }, { b, c }, { d, e }, { f, g } }



Next-state split



Minimal bisimilar state machine



4 instead of 7 states

How to check if M1 and M2 are equivalent

1. Minimize M1 and call the result N1
2. Minimize M2 and call the result N2
3. Check if the states of N1 can be renamed so that N1 and N2 are identical