



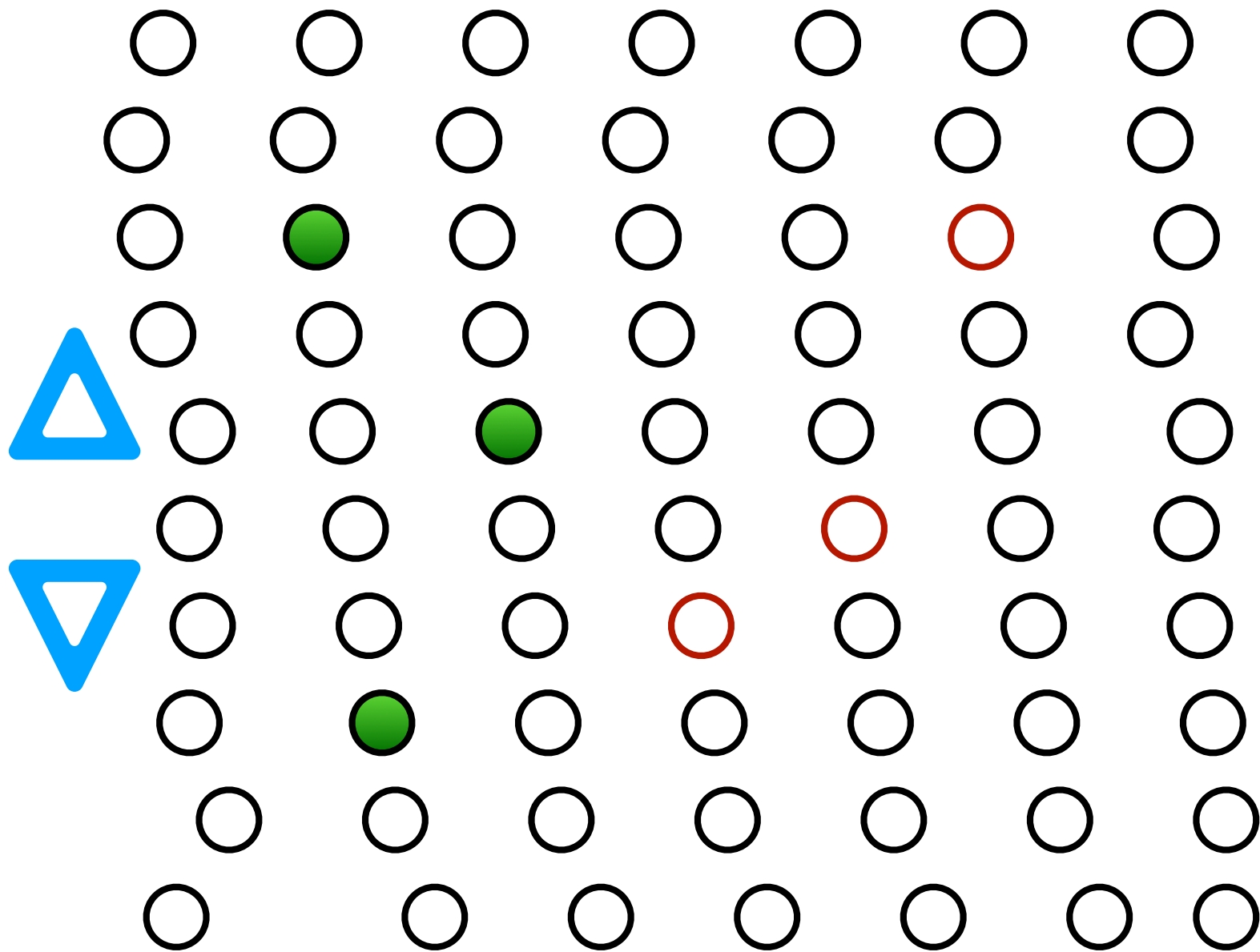
Computation and Logic

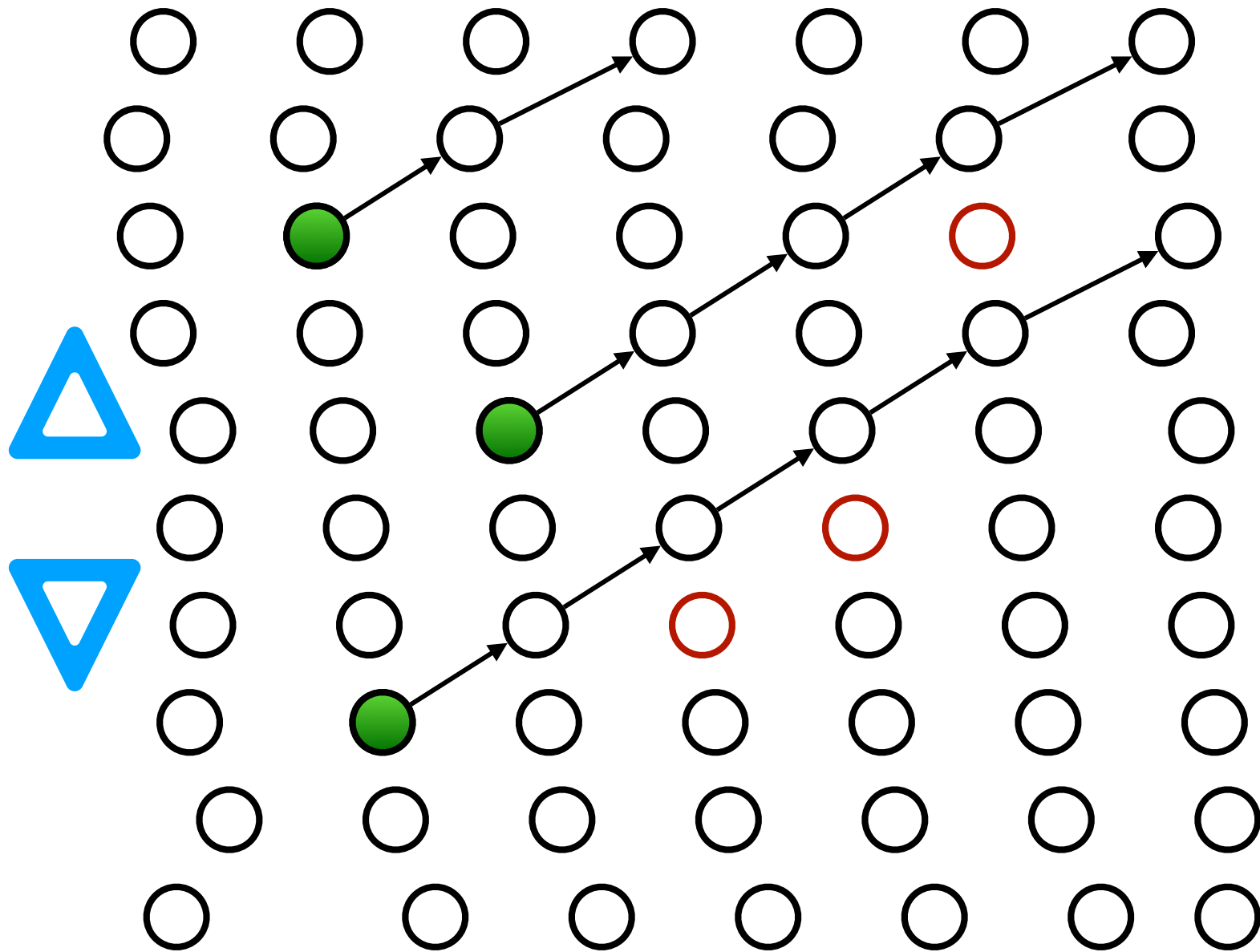
regex and FSM

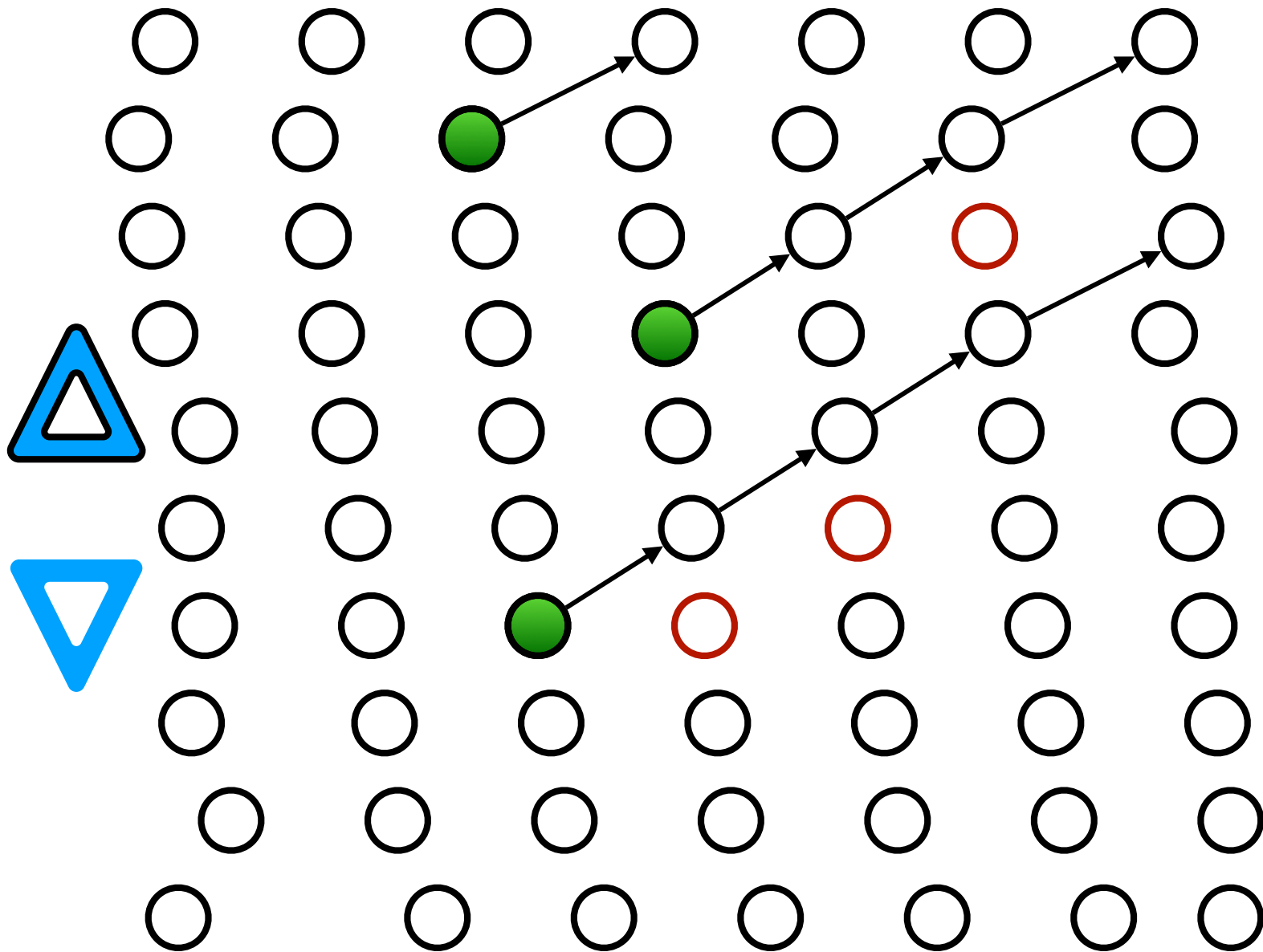
Michael Fourman
@mp4man

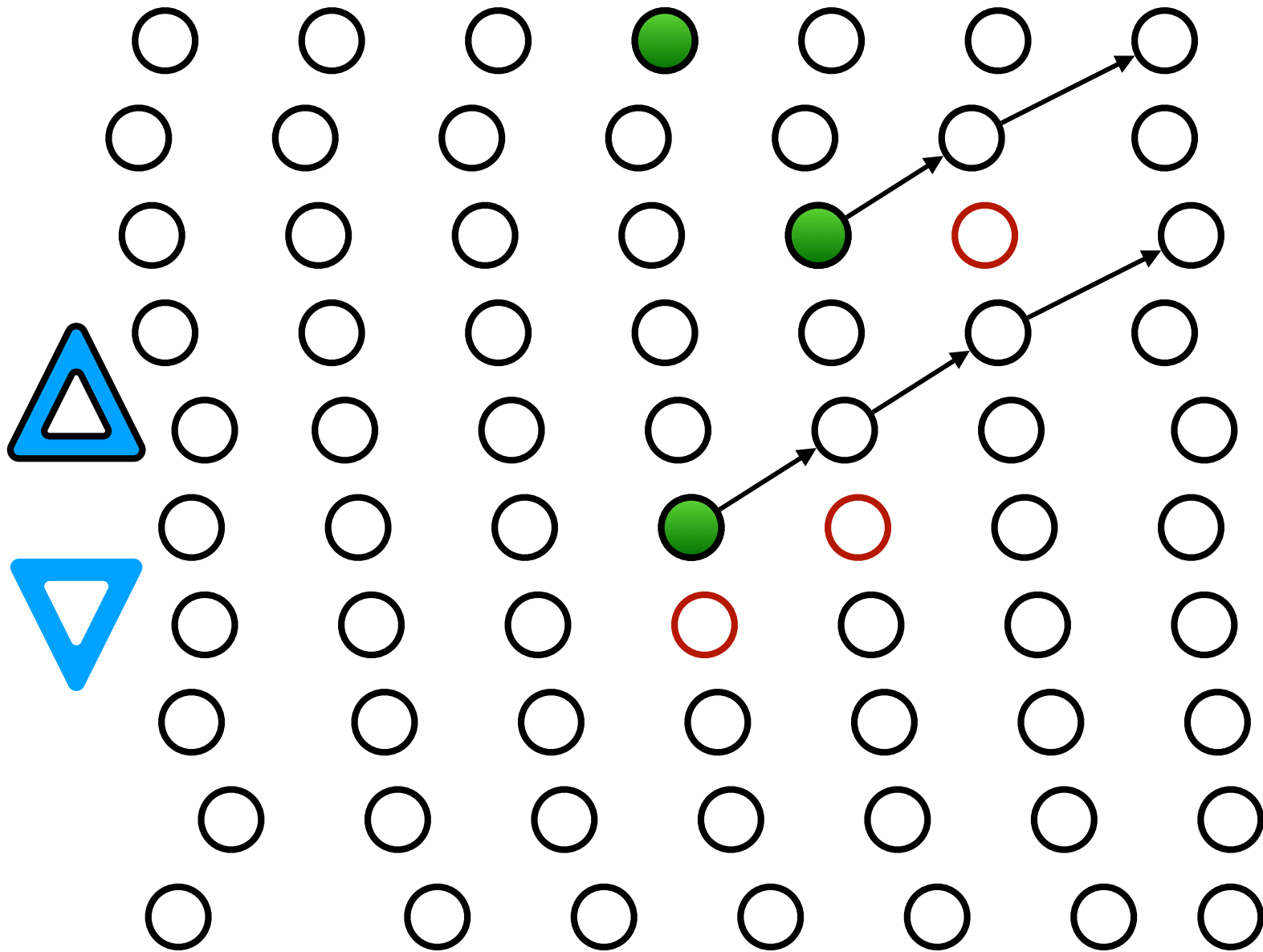


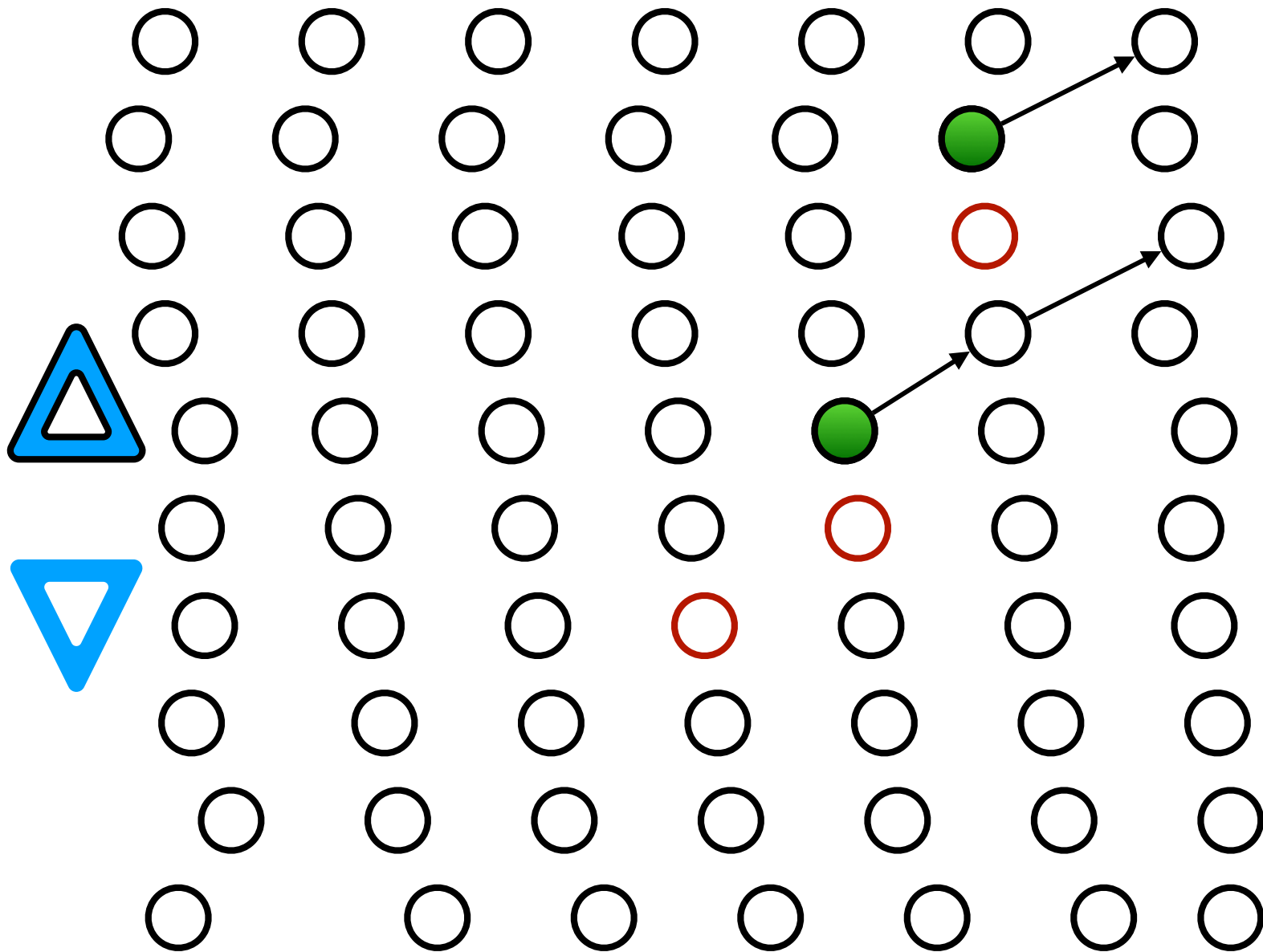


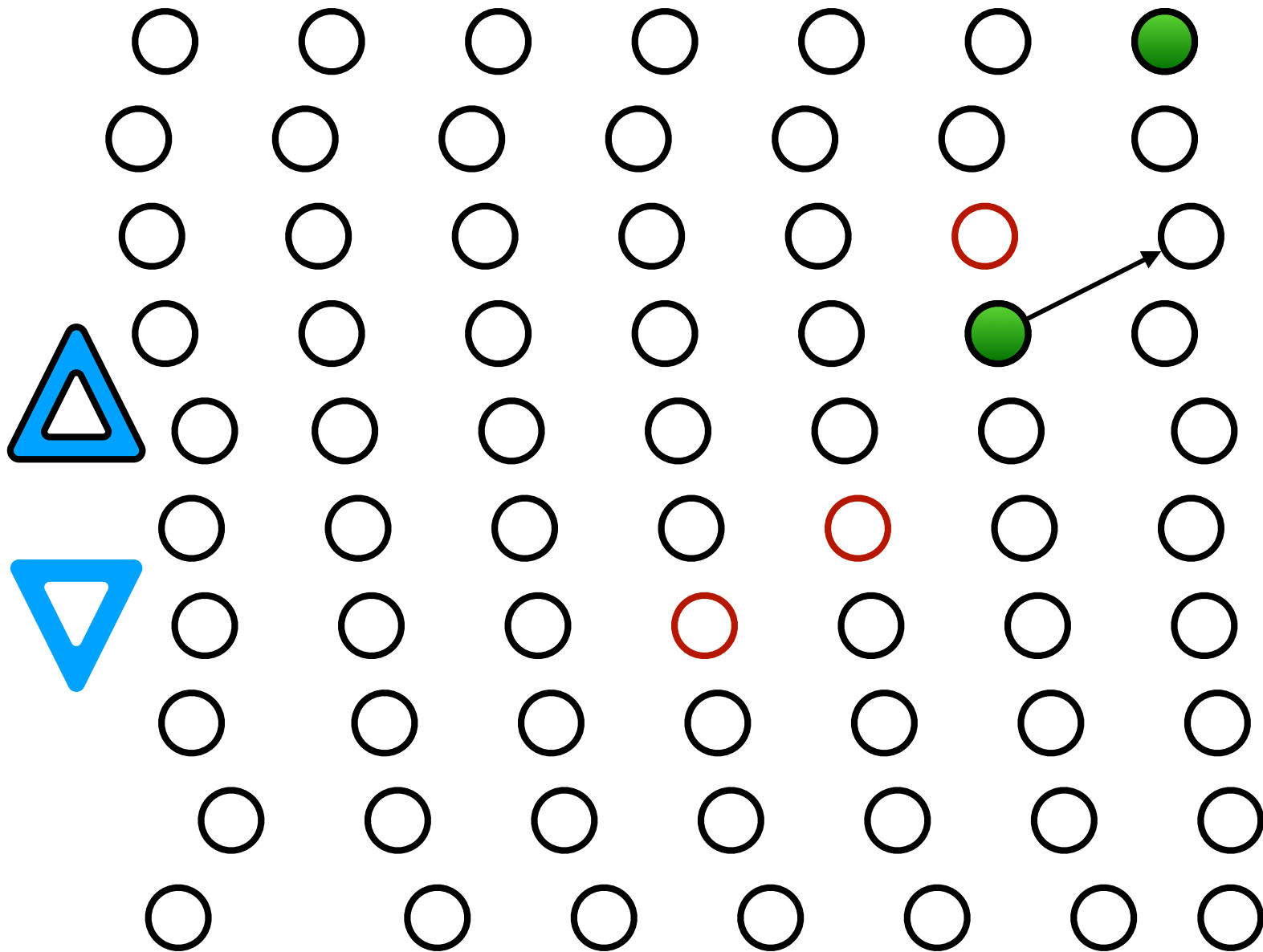


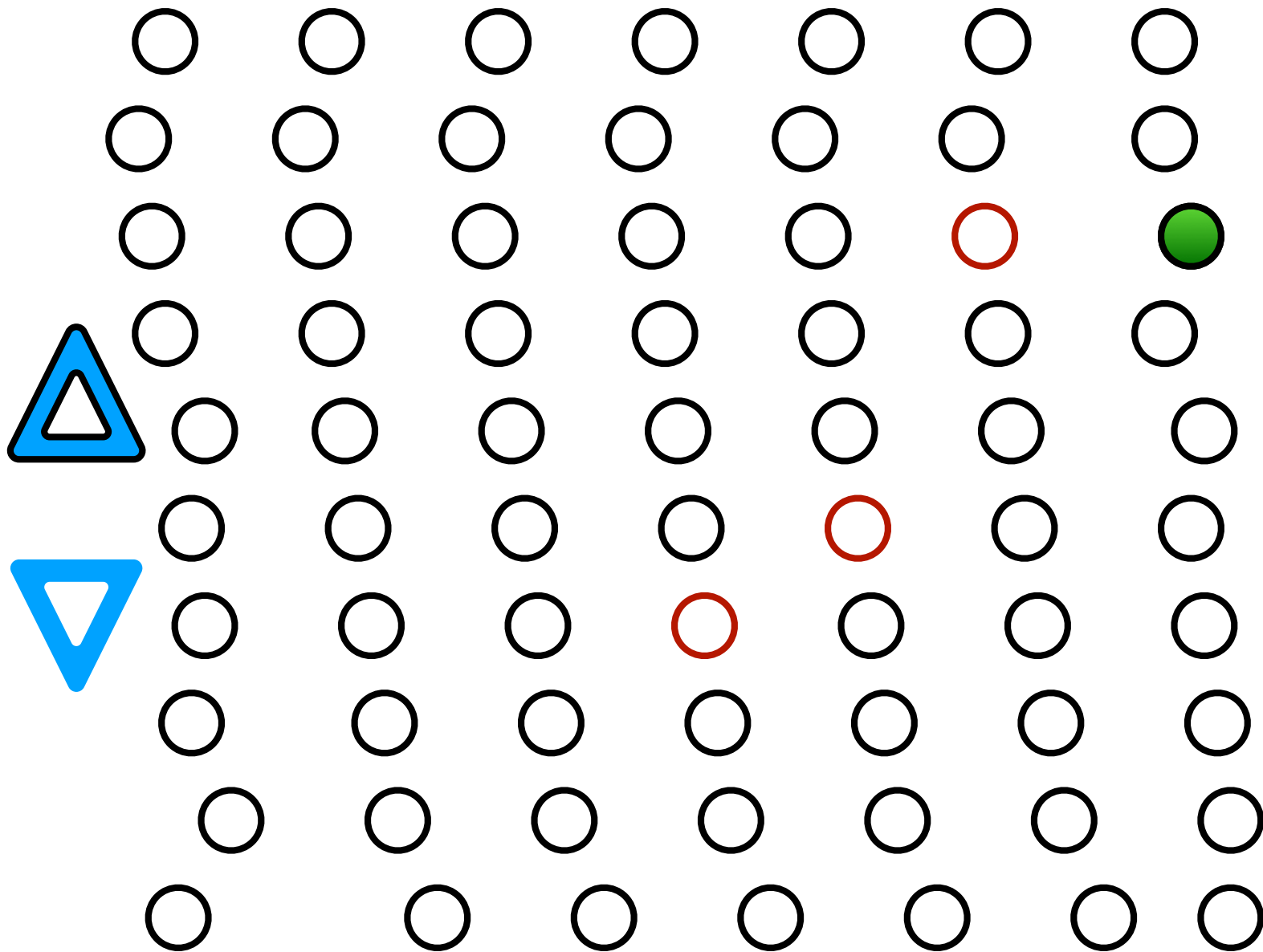


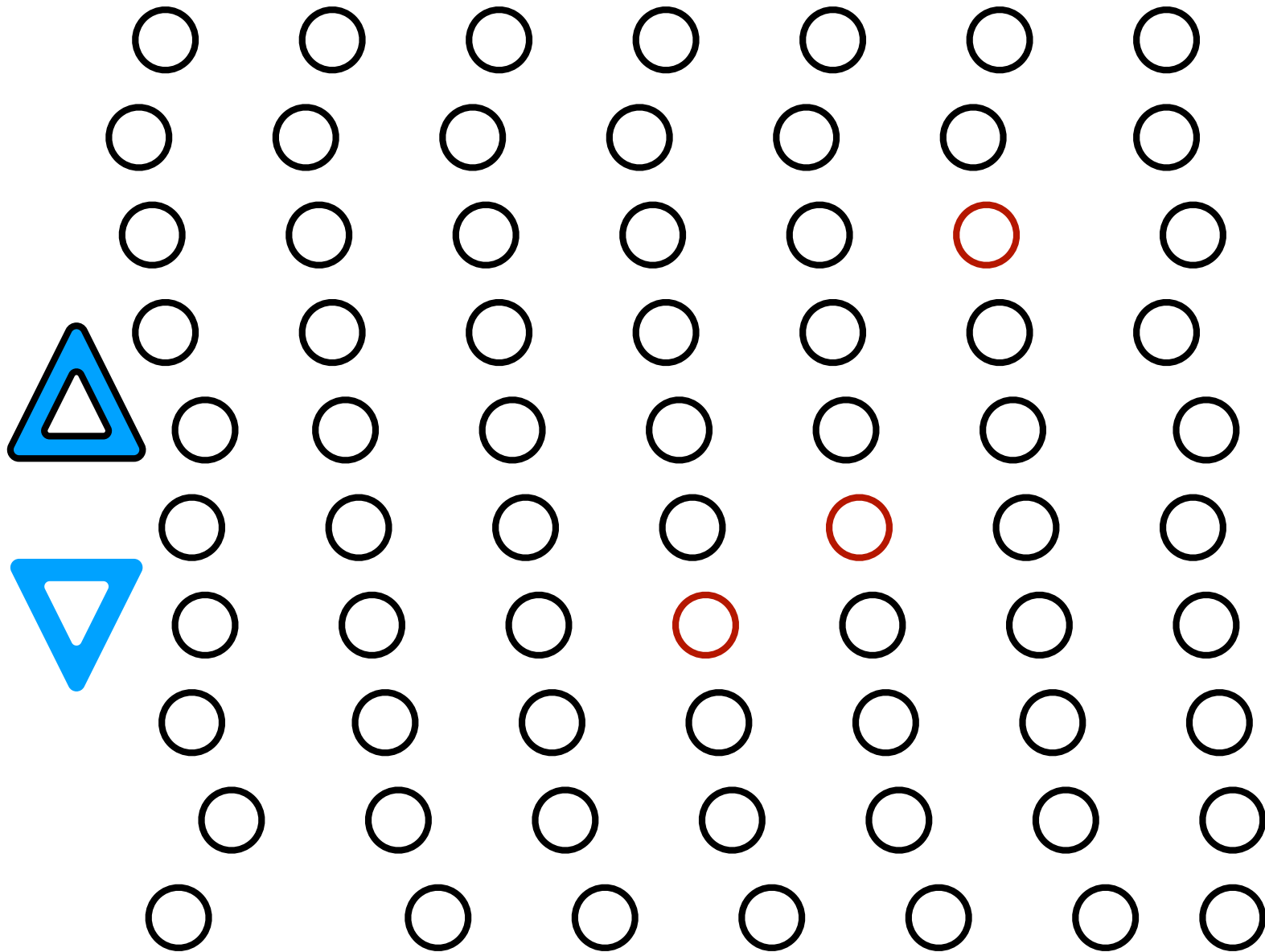


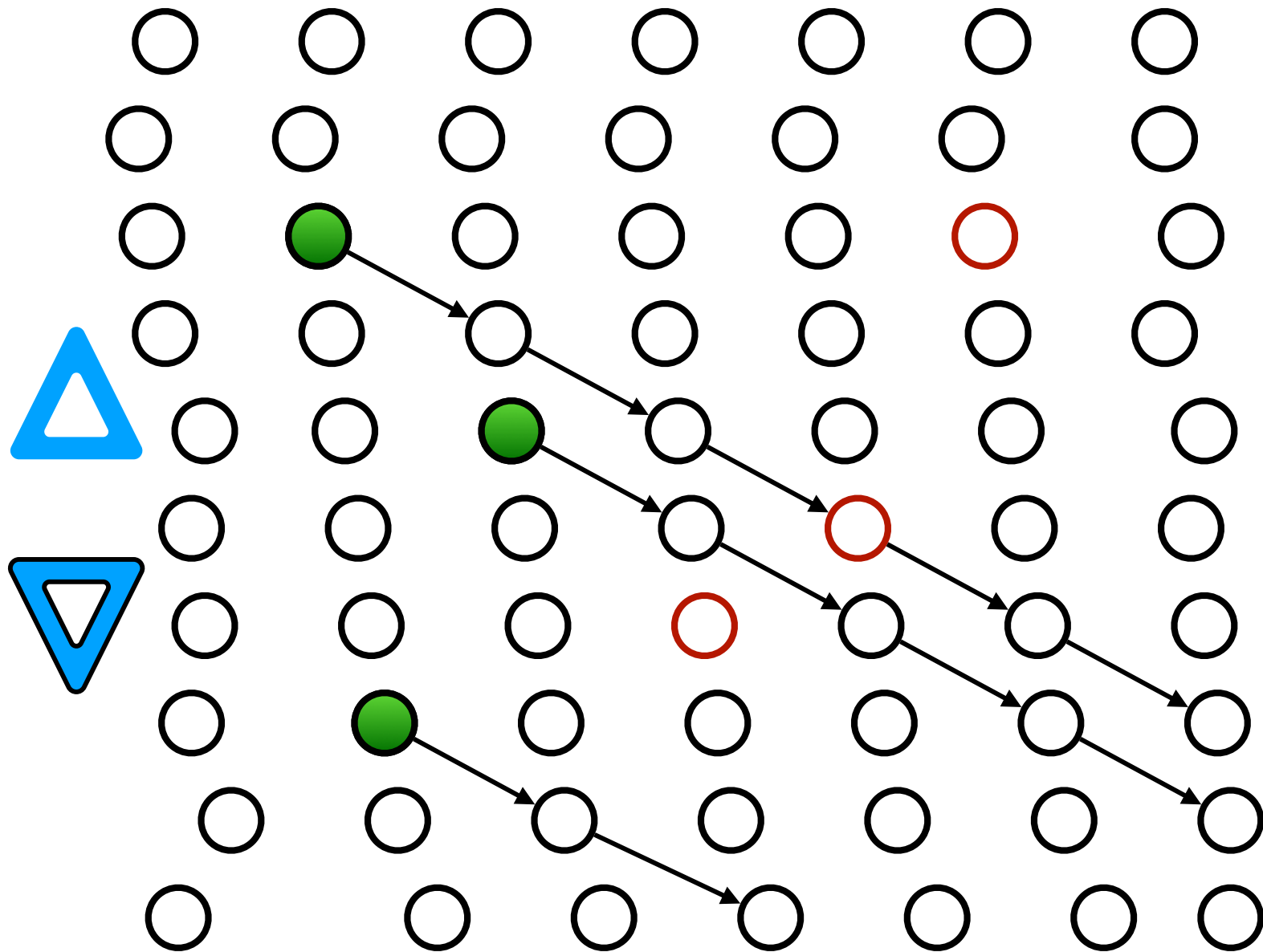


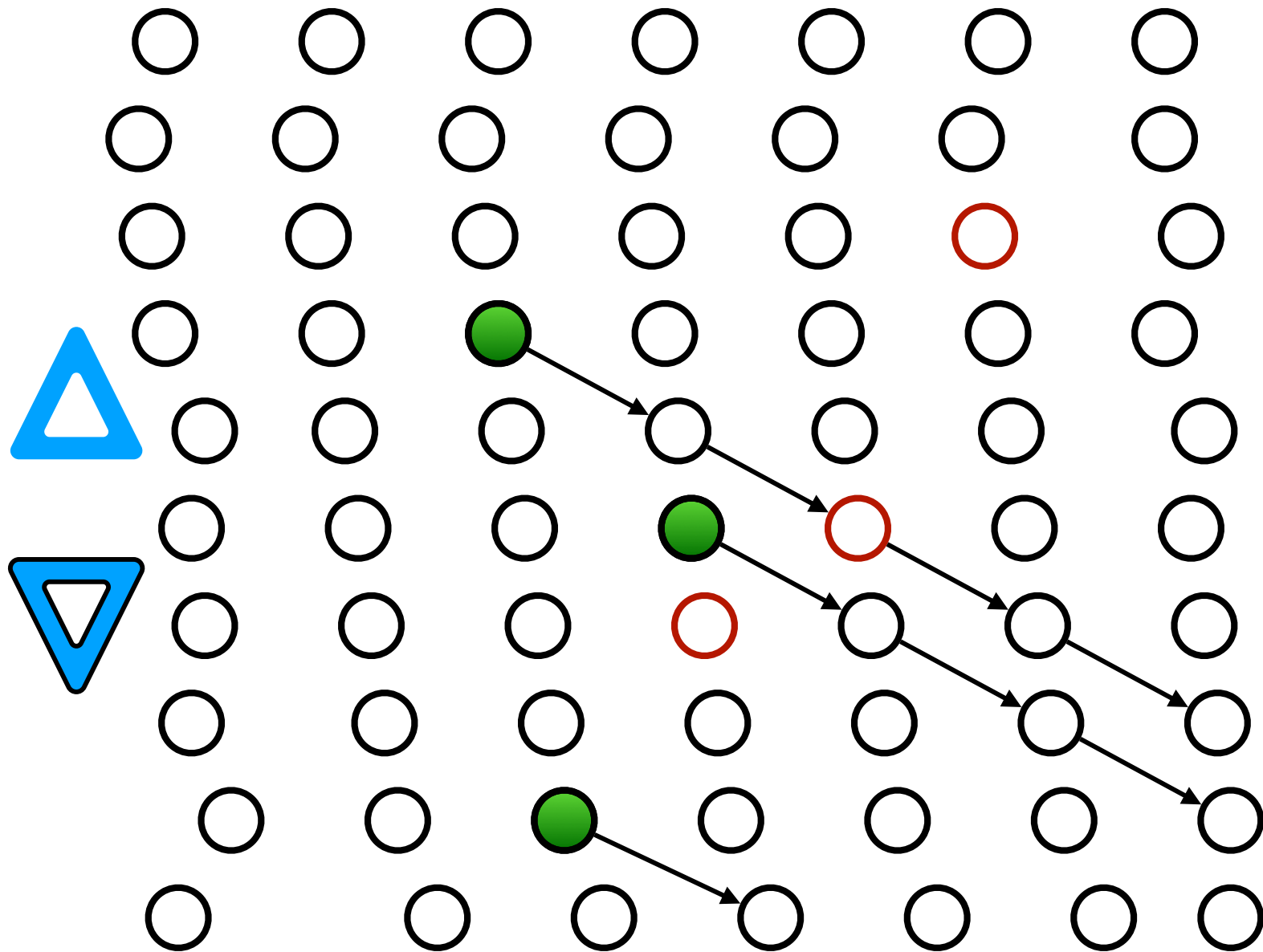


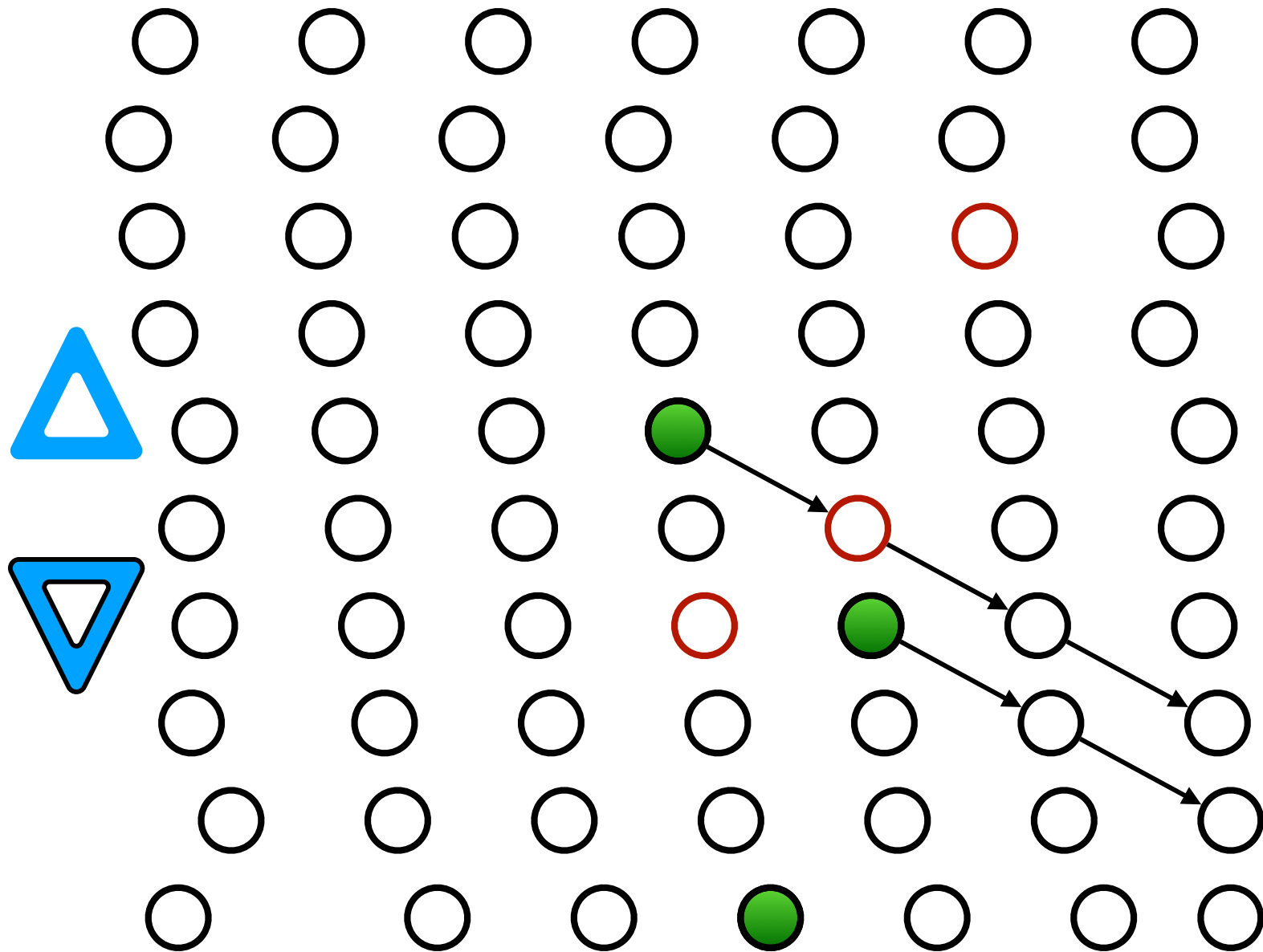


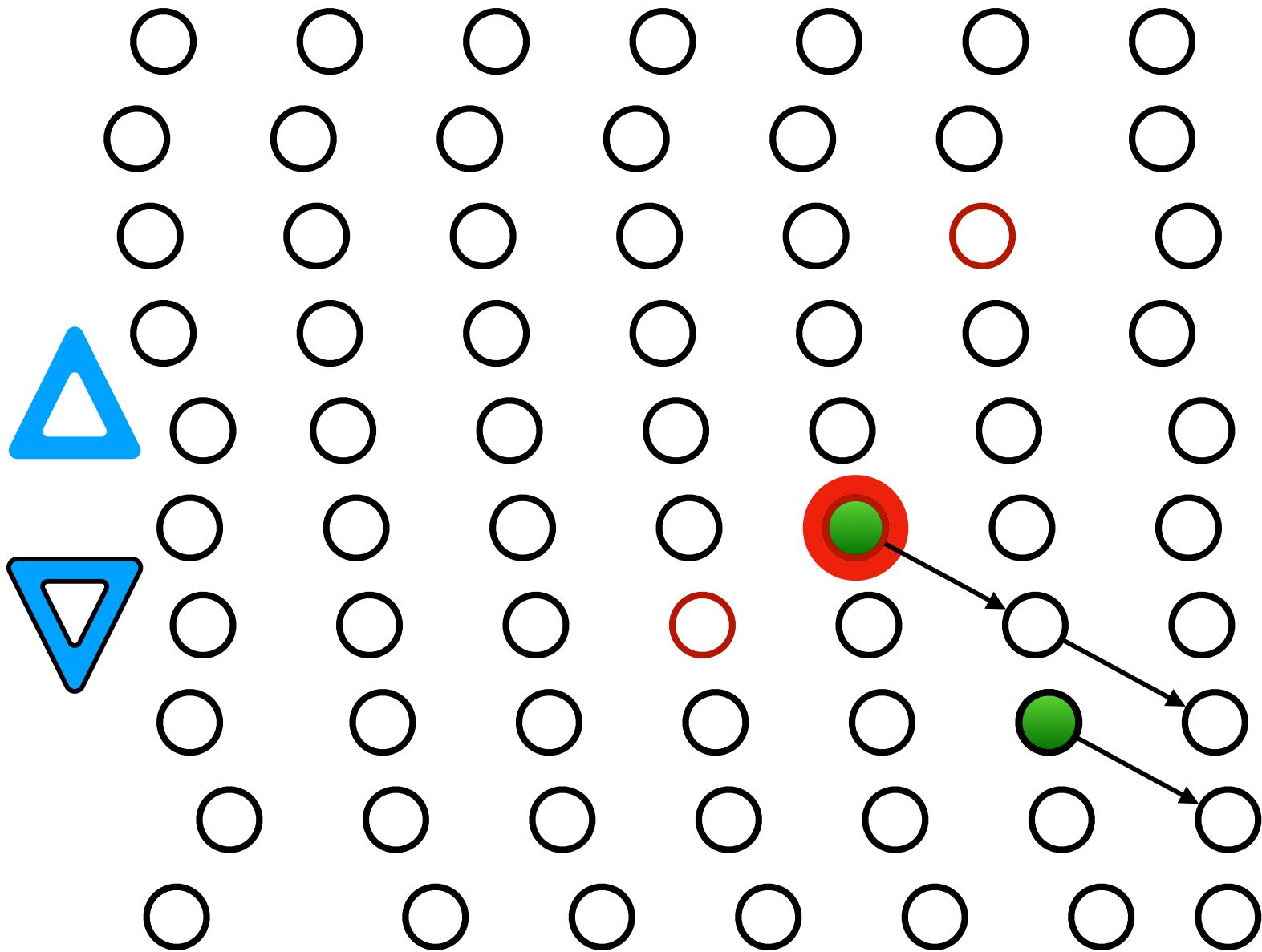


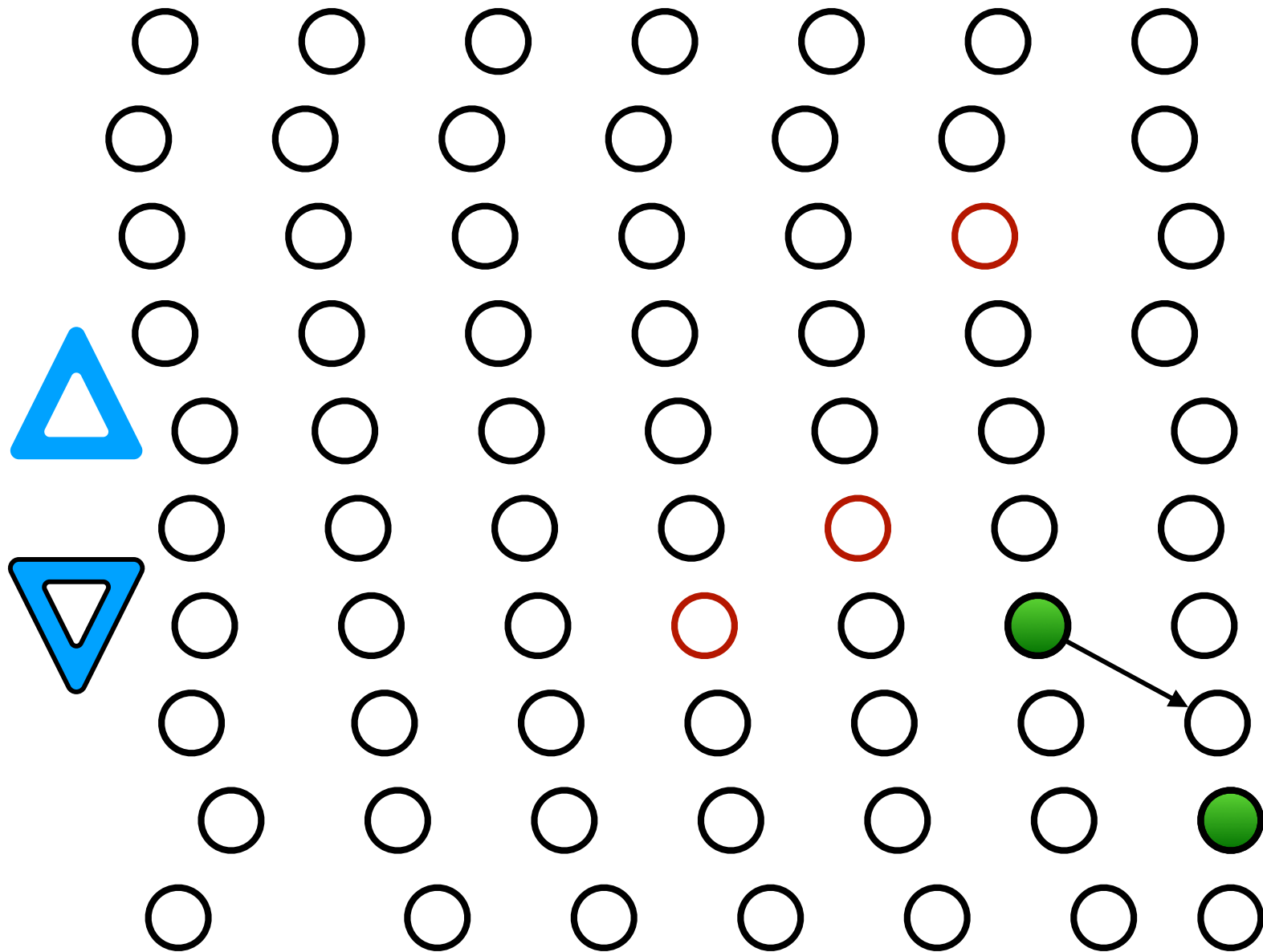


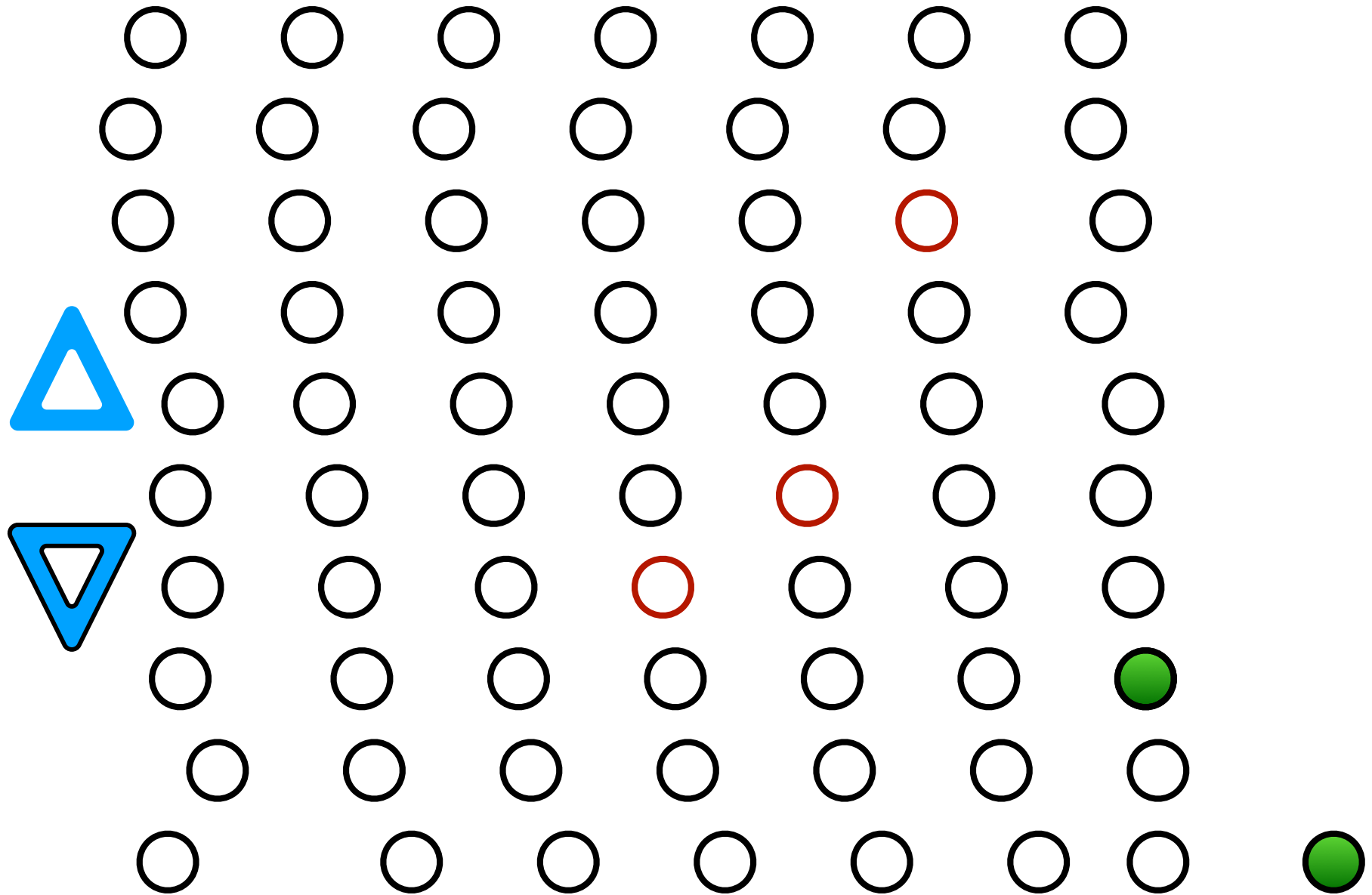


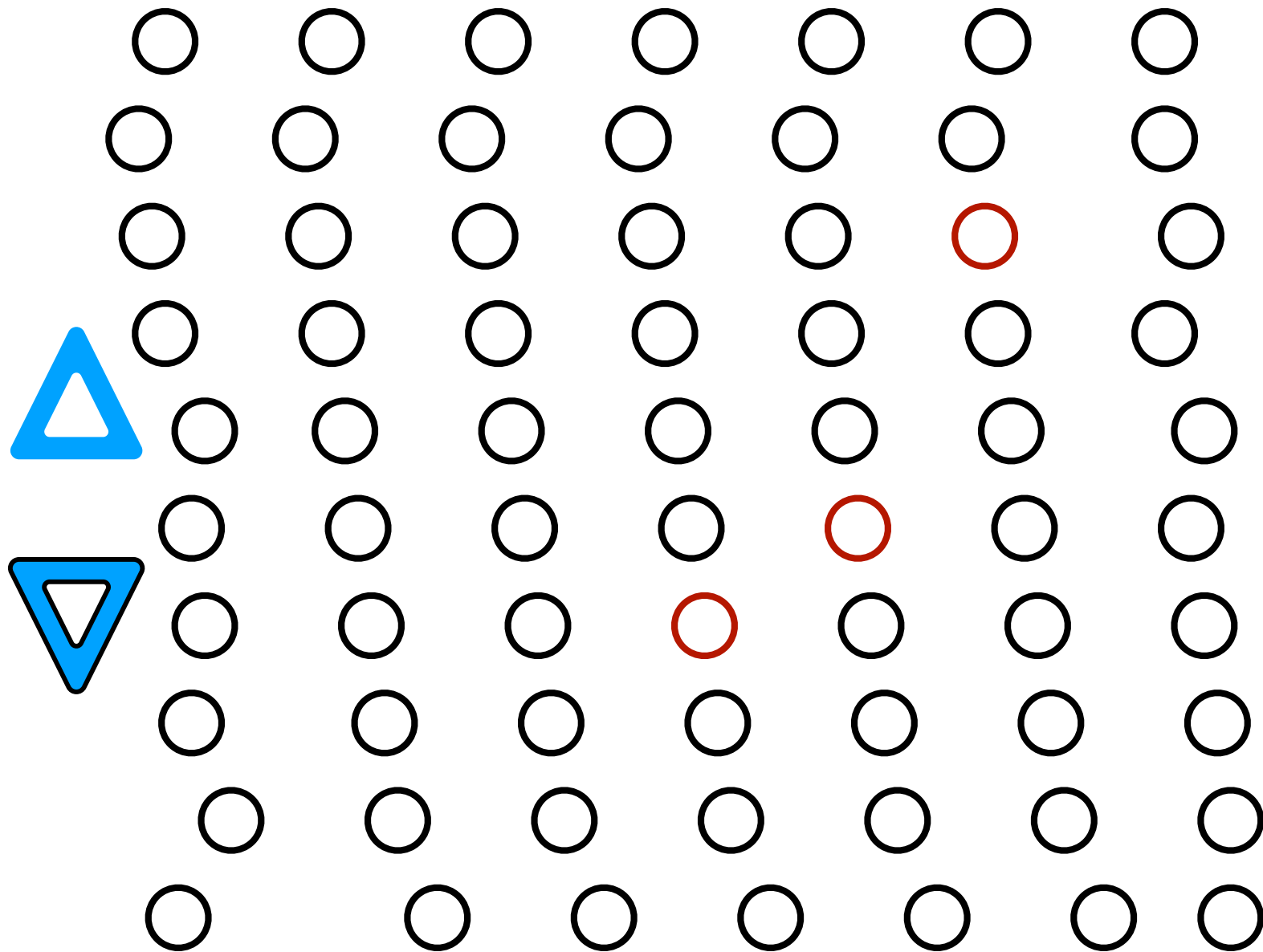


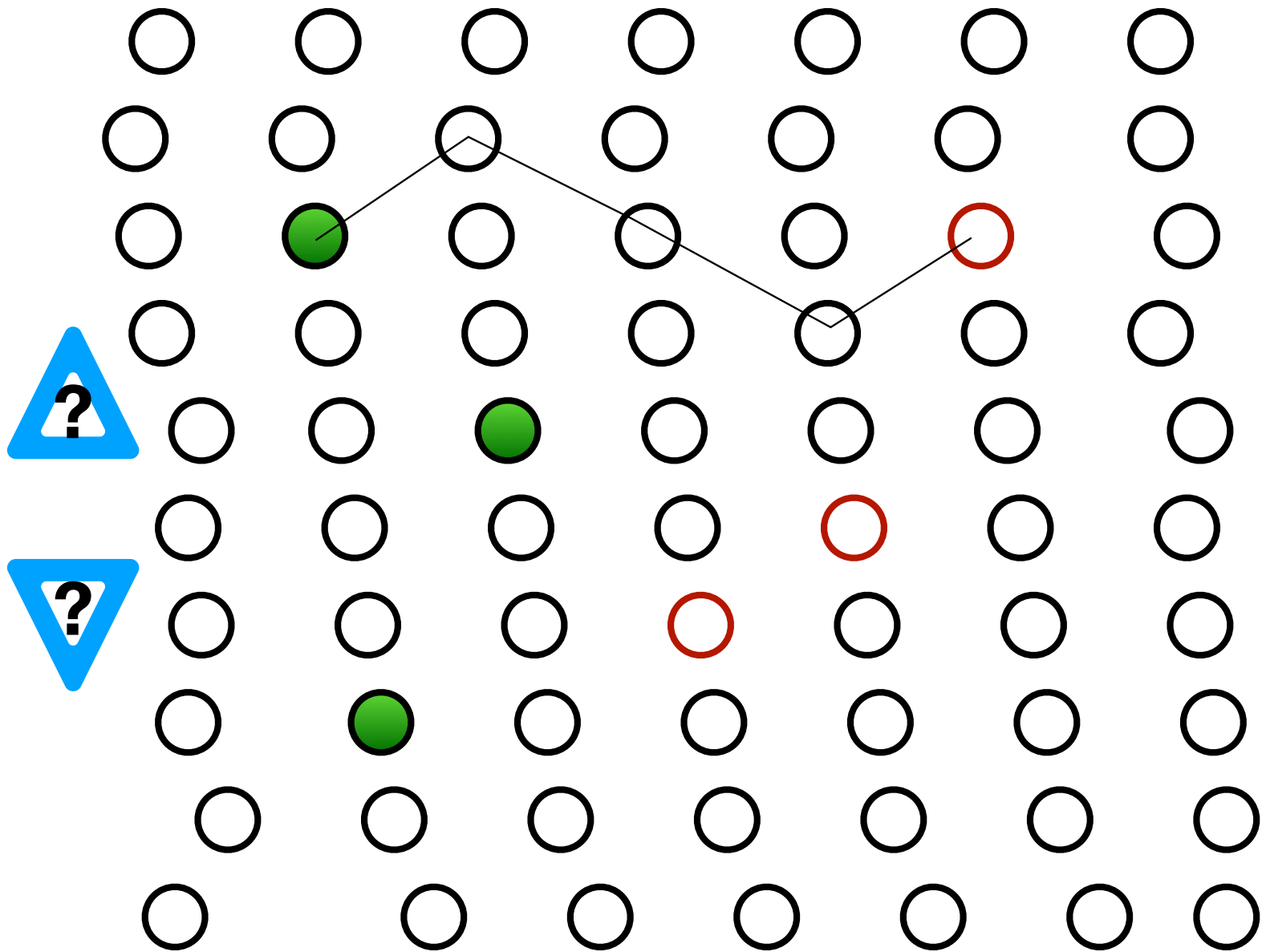


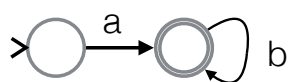
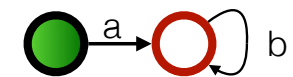




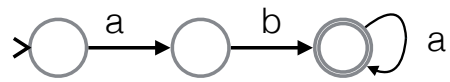




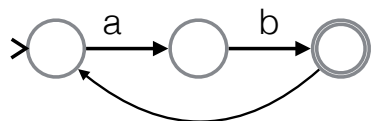




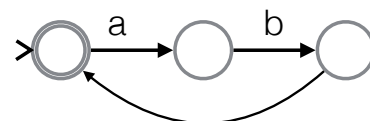
ab^*



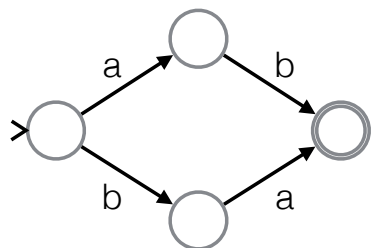
aba^*



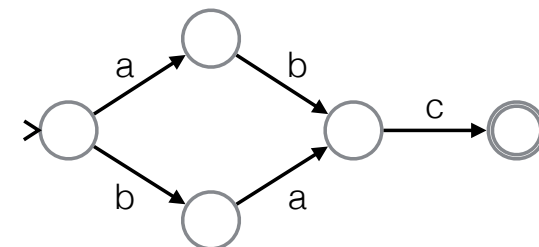
$ab(aab)^*$



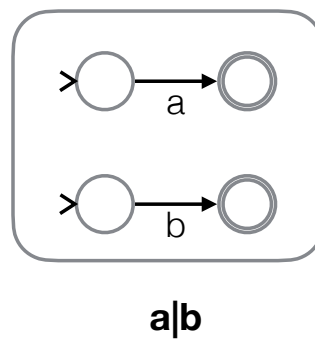
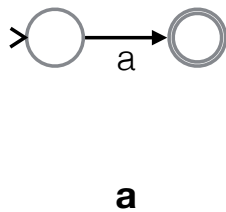
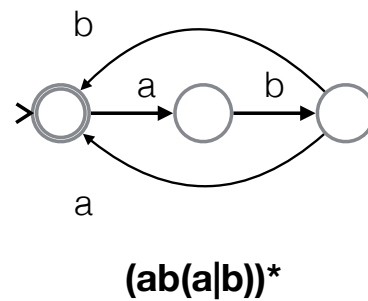
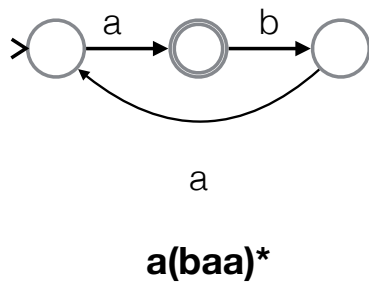
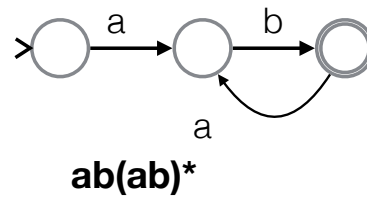
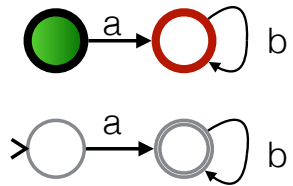
$(aba)^*$



$ab|ba$



$(ab|ba)c$



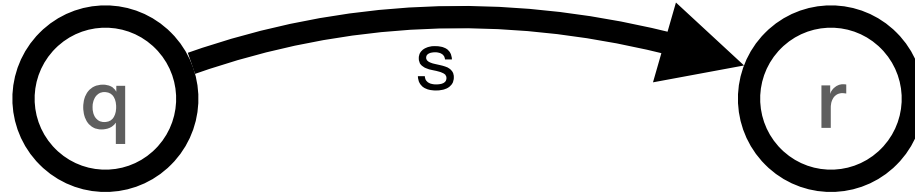
IDEA

a machine consists of lights (states)
and buttons (symbols)
some lights are special (final states)
some lights are lit (start states)

when we press a button the lights change according to
some rule

we want to find which sequences of button presses will turn
some accepting state on

IDEA



when we press a button the lights change according to some rule

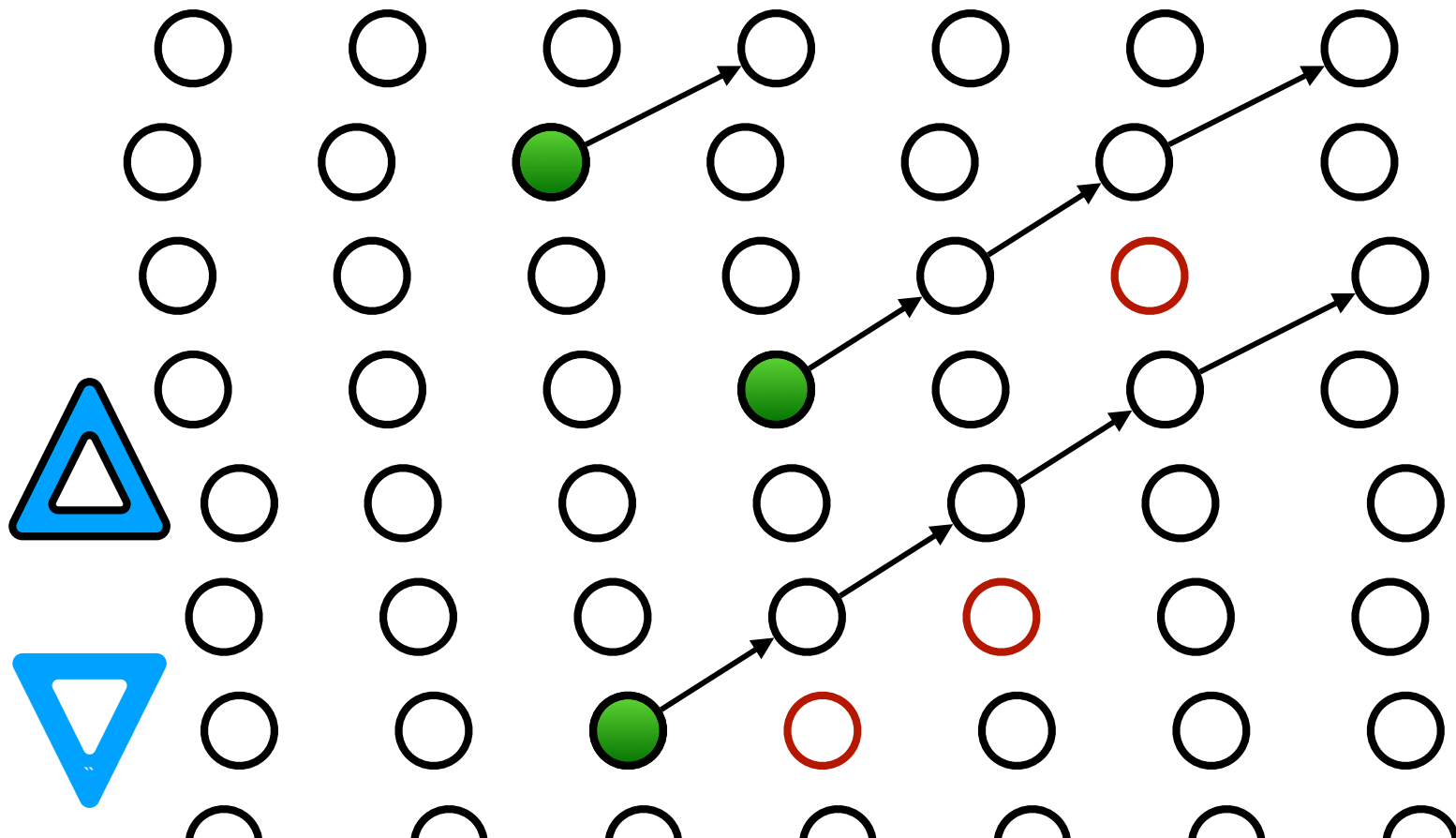
the rule is simple: we have a finite set of transitions
(we draw these as labelled arrows)

r is on after s is pushed iff
there is some transition (q, s, r)
for which q was on before s was pressed

```

type Sym = Char
type Trans q = (q, Sym, q)
-- lift transitions to [q]
next :: (Eq q) => [Trans q] -> Sym -> [q] -> [q]
next trans x ss = [ q' | (q, y, q') <- trans, x == y, q`elem`ss ]

```



```

type Sym = Char
type Trans q = (q, Sym, q)
data FSM q = FSM [q] [Sym] [Trans q] [q] [q] deriving Show

-- lift transitions to [q]
next :: (Eq q) => [Trans q] -> Sym -> [q] -> [q]
next trans x ss = [ q' | (q, y, q') <- trans, x == y, q `elem` ss ]

-- apply transitions for symbol x to move the start states
step :: Eq q => FSM q -> Sym -> FSM q
step (FSM qs as ts ss fs) x = FSM qs as ts (next ts x ss) fs

```

A machine has

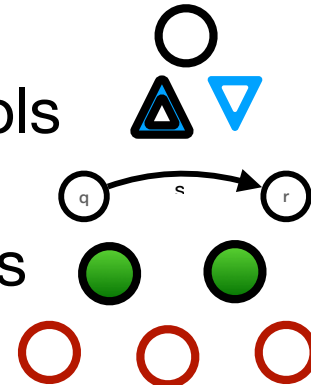
qs a set of states

as an alphabet of symbols

ts a set of transitions

ss a set of starting states

fs a set of final states



Automaton

An *NFA* is represented formally by a **5-tuple**, , consisting of

- a finite **set** of states
- a finite set of **input symbols**
- a transition function
- an *initial* (or *start*) state
- a set of states distinguished as *accepting* (or *final*) states .

not quite the definition we will use

Automaton

A *FSM* is represented formally by a **5-tuple**, , consisting of

- *qs* a finite **set** of states
- *as* a finite set of **symbols**
- *ts* a transition relation - represented as a set of triples
- *ss* a set of *initial* (or *start*) states
- *fs* a set of *accepting* (or *final*) states

```

type Sym = Char
type Trans q = (q, Sym, q)
data FSM q = FSM [q] [Sym] [Trans q] [q] [q] deriving Show

-- lift transitions to [q]
next :: (Eq q) => [Trans q] -> Sym -> [q] -> [q]
next trans x ss = [ q' | (q, y, q') <- trans, x == y, q `elem` ss ]

-- apply transitions for symbol x to move the start states
step :: Eq q => FSM q -> Sym -> FSM q
step (FSM qs as ts ss fs) x = FSM qs as ts (next ts x ss) fs

accepts :: (Eq q) => FSM q -> String -> Bool
accepts (FSM qs as ts ss fs) "" = or [ q `elem` ss | q <- fs ]
accepts fsm (x : xs) = accepts (step fsm x) xs

trace :: Eq q => FSM q -> [Sym] -> [[q]]
trace      (FSM _ _ _ ss _) []      = [ss]
trace fsm@(FSM _ _ _ ss _) (x:xs) = ss : trace (step fsm x) xs

```

Traffic Light Signals



RED means 'Stop'. Wait behind the stop line on the carriageway



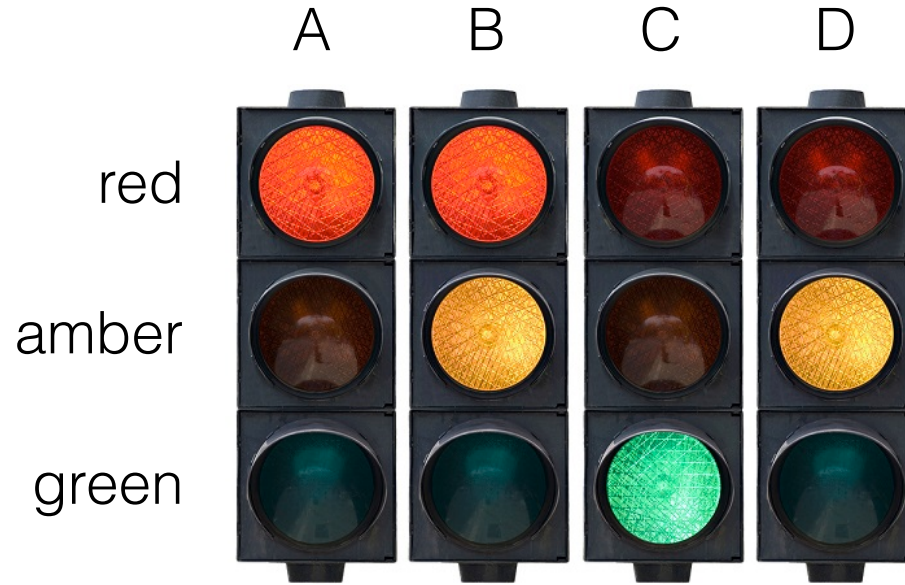
RED AND AMBER also means 'Stop'. Do not pass through or start until GREEN shows



GREEN means you may go on if the way is clear. Take special care if you intend to turn left or right and give way to pedestrians who are crossing



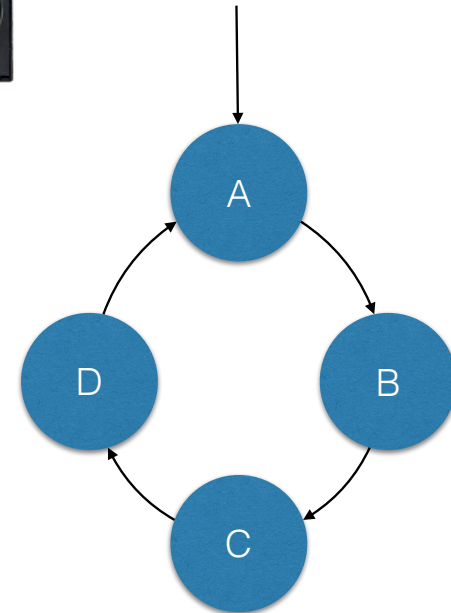
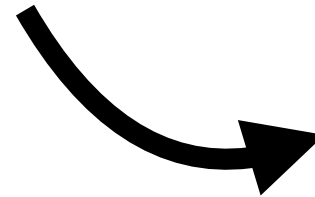
AMBER means 'Stop' at the stop line. You may go on only if the AMBER appears after you have crossed the stop line or are so close to it that to pull up might cause an accident

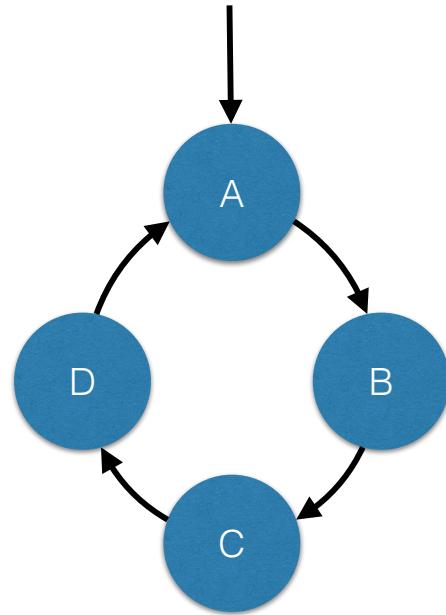


logic & computation

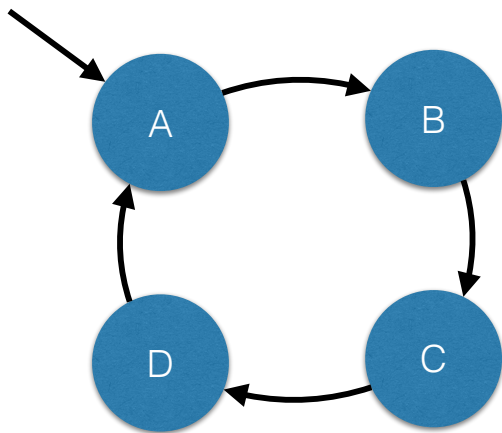


red iff A or B
amber iff B or D
green iff C

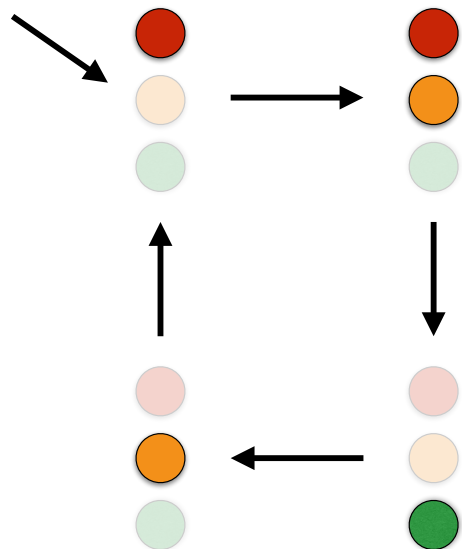




























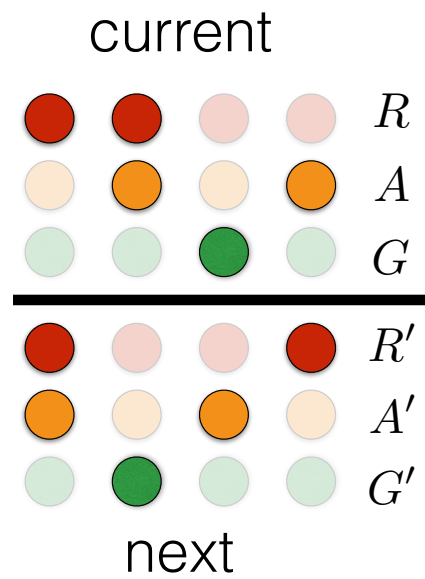
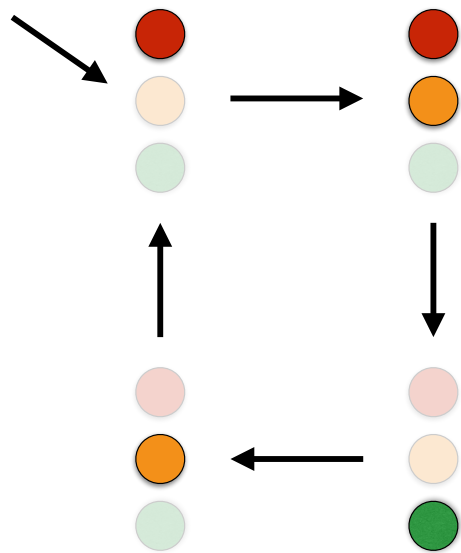
current			
A	B	C	D
<hr/>			
B	C	D	A
next			



current			
A	B	C	D
<hr/>			
B	C	D	A
next			

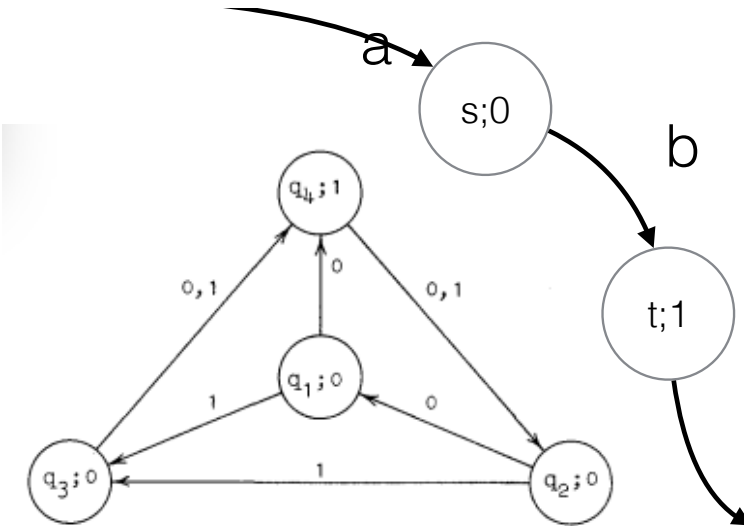


current				
				R
				A
				G
<hr/>				
				R'
				A'
				G'
next				



Moore machine 1956

Present State			Present State	Present Output
Previous State	Previous Input			
	0	1		
q_1	q_4	q_3	q_1	0
q_2	q_1	q_3	q_2	0
q_3	q_4	q_4	q_3	0
q_4	q_2	q_2	q_4	1



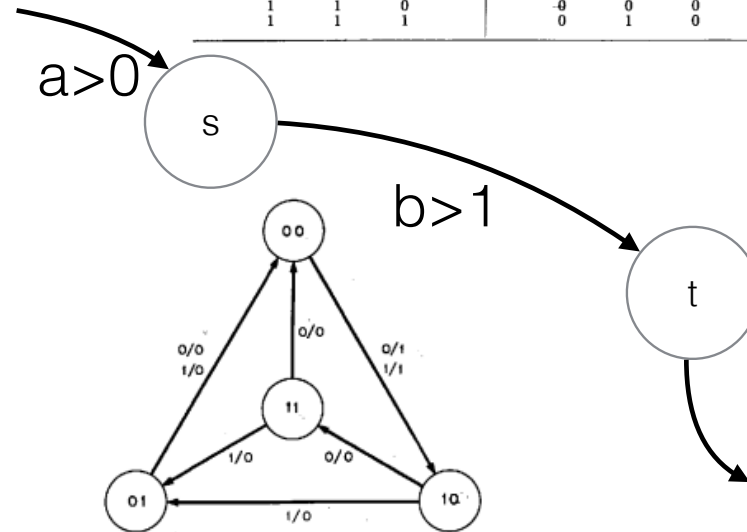
Edward F. Moore 1925-2003

Gedanken - Experiments on Sequential Machines, 1956

http://people.mokk.bme.hu/~kornai/termeszetes/moore_1956.pdf

Mealy machine 1955

q_1	q_2	x	\bar{q}_1	\bar{q}_2	y
0	0	0	1	0	1
0	0	1	1	0	1
0	1	0	0	0	0
0	1	1	0	0	0
1	0	0	1	1	0
1	0	1	0	1	0
1	1	0	0	0	0
1	1	1	0	1	0



George H. Mealy 1927-2010

A Method for
Synthesizing Sequential Circuits 1955

<http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6771467>

Finite State Machine concepts proved valuable in language parsing (compilers) and sequential circuit design

Moore is less

keep it simple

What yes/no questions about inputs
can be answered by a finite deterministic machine?

possible inputs
 Σ^* = finite sequences

for each input $x \in \Sigma^*$
a yes/no question Q
gives an answer $Q(x)$
 $\{ x \in \Sigma^* \mid Q(x) \} \subseteq \Sigma^*$

yes/no questions
correspond to
subsets of Σ^*

$A \subseteq \Sigma^*$ subset
is $x \in A$ question

simple machines

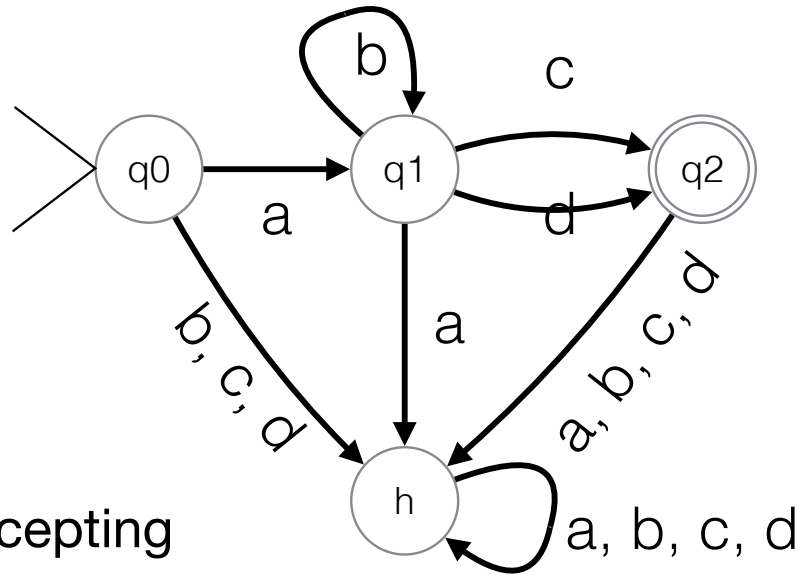
no outputs

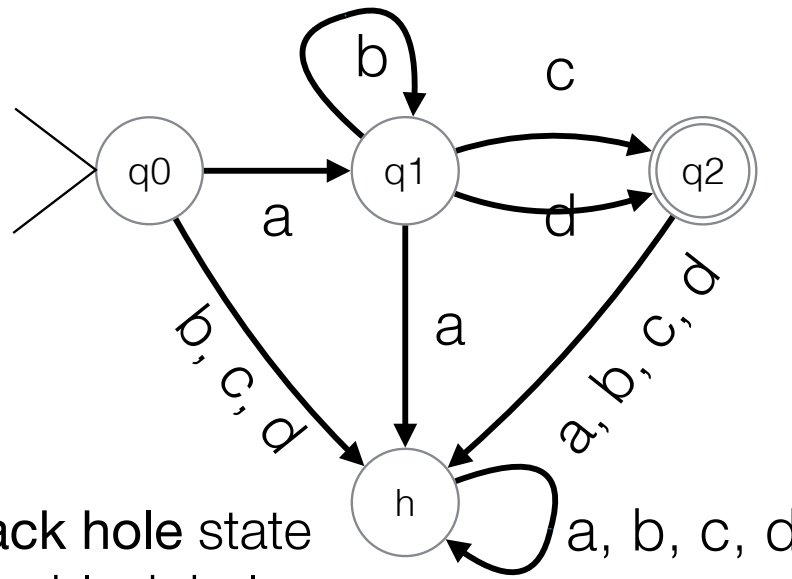
each input sequence
leads to a single state

the answer depends
only on the this state

for some states, yes  accepting

for the rest, no 

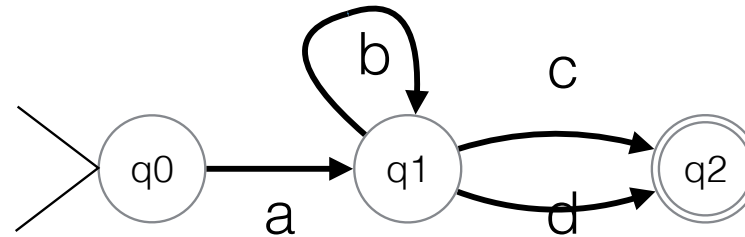




h is a **black hole** state
once in a black hole
we can never escape

omitting the black hole
gives a simpler diagram

still shows all paths
from start to accepting



DFA

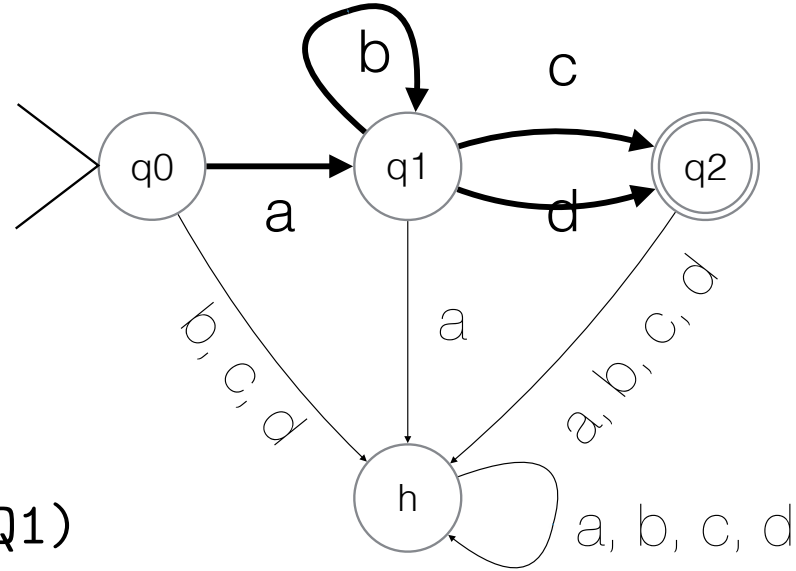
single start state

any number

of accepting states

each (state, input) pair
determines next state

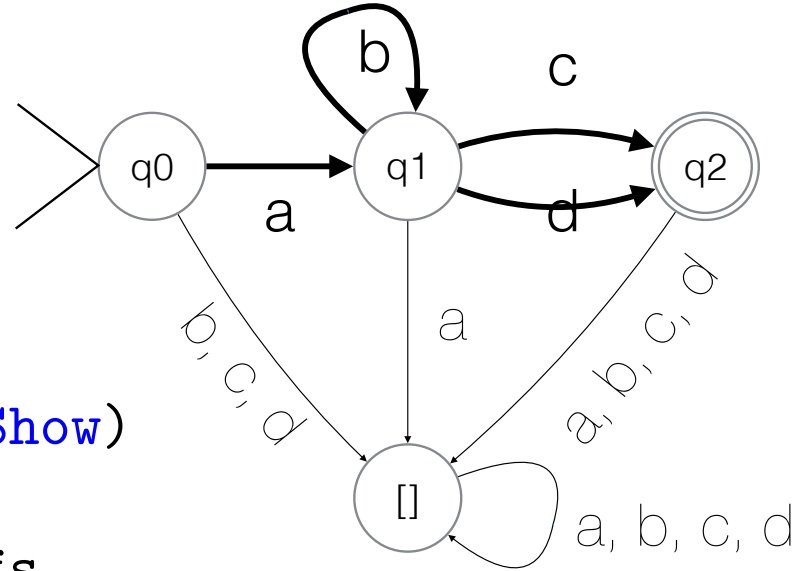
```
ts = [(Q0,a,Q1), (Q1,b,Q1)  
      , (Q1,c,Q2), (Q1,d,Q2)]
```



```
next :: (Eq q) => [Trans q] -> Sym -> [q] -> [q]
next trans x ss = [ q' | (q, y, q') <- trans, x == y, q `elem` ss ]
next ts a [Q0] = [Q1]
next ts b [Q1] = [Q1]
next ts c [Q1] = [Q2]
next ts d [Q1] = [Q2]
next ts _ _ = [] -- black hole
```

Always, at most one *state is lit*

DFA



```
data EG = Q0|Q1|Q2
      deriving (Eq,Show)
[a,b,c,d] = "abcd"
eg = FSM qs as ts ss fs
  where
    qs = [Q0,Q1,Q2]
    as = [a,b,c,d]
    ts = [(Q0,a,Q1),(Q1,b,Q1),(Q1,c,Q2),(Q1,d,Q2)]
    ss = [Q0]
    fs = [Q2]
```

```
data EG = Q0|Q1|Q2 deriving (Eq,Show)
```

```
[a,b,c,d] = "abcd"
```

```
eg = FSM qs as ts ss fs
```

```
where
```

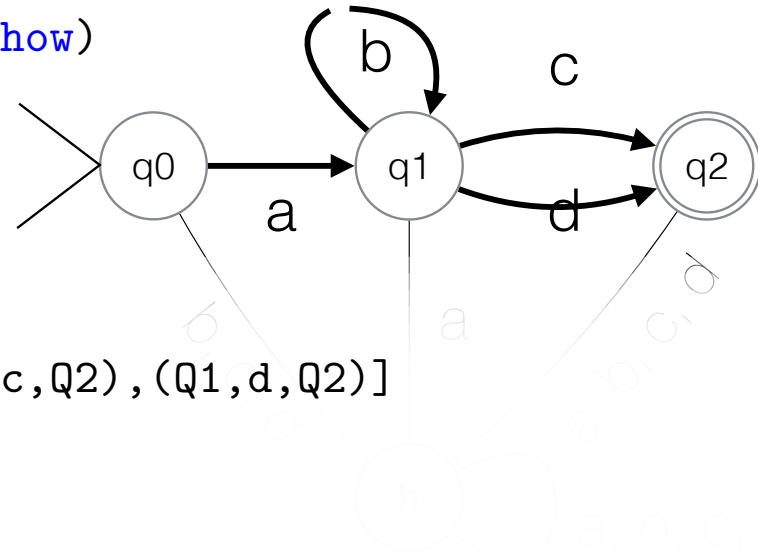
```
  qs = [Q0,Q1,Q2]
```

```
  as = [a,b,c,d]
```

```
  ts = [(Q0,a,Q1),(Q1,b,Q1),(Q1,c,Q2),(Q1,d,Q2)]
```

```
  ss = [Q0]
```

```
  fs = [Q2]
```



```
trace      (FSM _ _ _ ss _) []      = [ss]
```

```
trace fsm@(FSM _ _ _ ss _) (x:xs) = ss : trace (step fsm x) xs
```

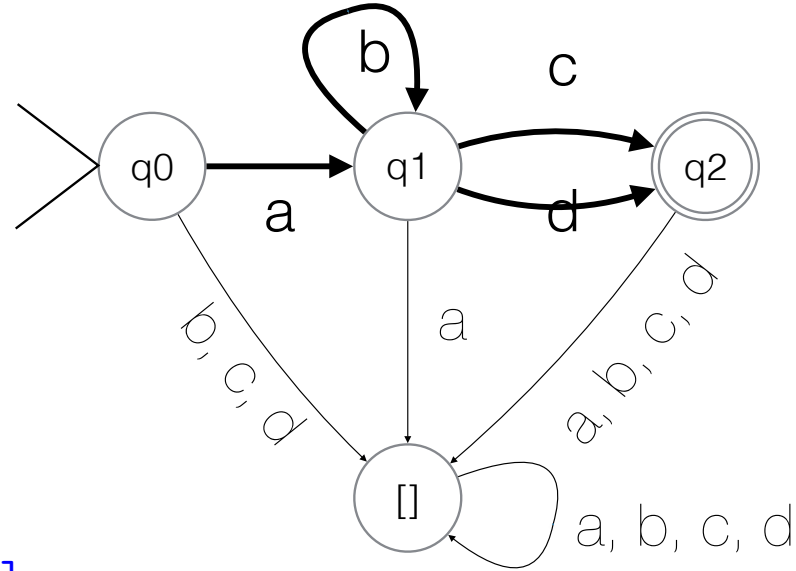
```
> trace eg "abbc"
```

```
[[Q0],[Q1],[Q1],[Q1],[Q2]]
```

```
> trace eg "abbcd"
```

```
[[Q0],[Q1],[Q1],[Q1],[Q2],[]]
```

DFA



```
isDFA :: Eq q => FSM q -> Bool
```

```
isDFA (FSM qs as ts ss fs) =
```

```
  (length ss == 1)
```

```
  &&
```

```
  and[ r == q' | (q, a, q') <- ts, r <- qs, (q, a, r) `elem` ts ]
```