



Early Reinforcement Learning

Learning Objectives

- Understand the History of Reinforcement Learning
 - Value Iteration
 - Policy Iteration
 - TD-Learning
 - Q-Learning

Agenda

History Overview

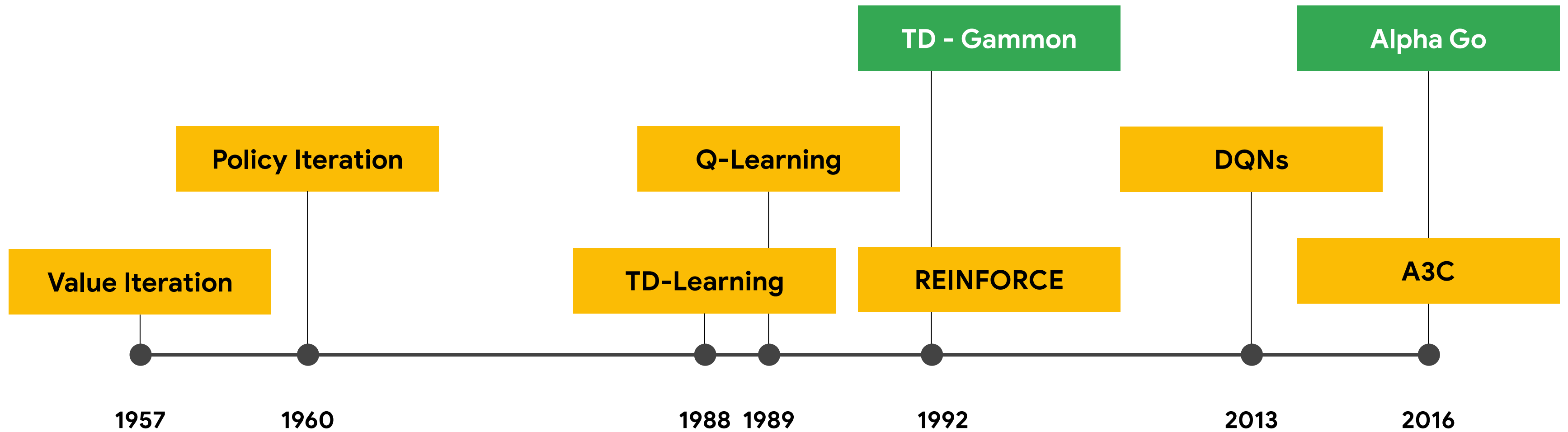
Value Iteration

Policy Iteration

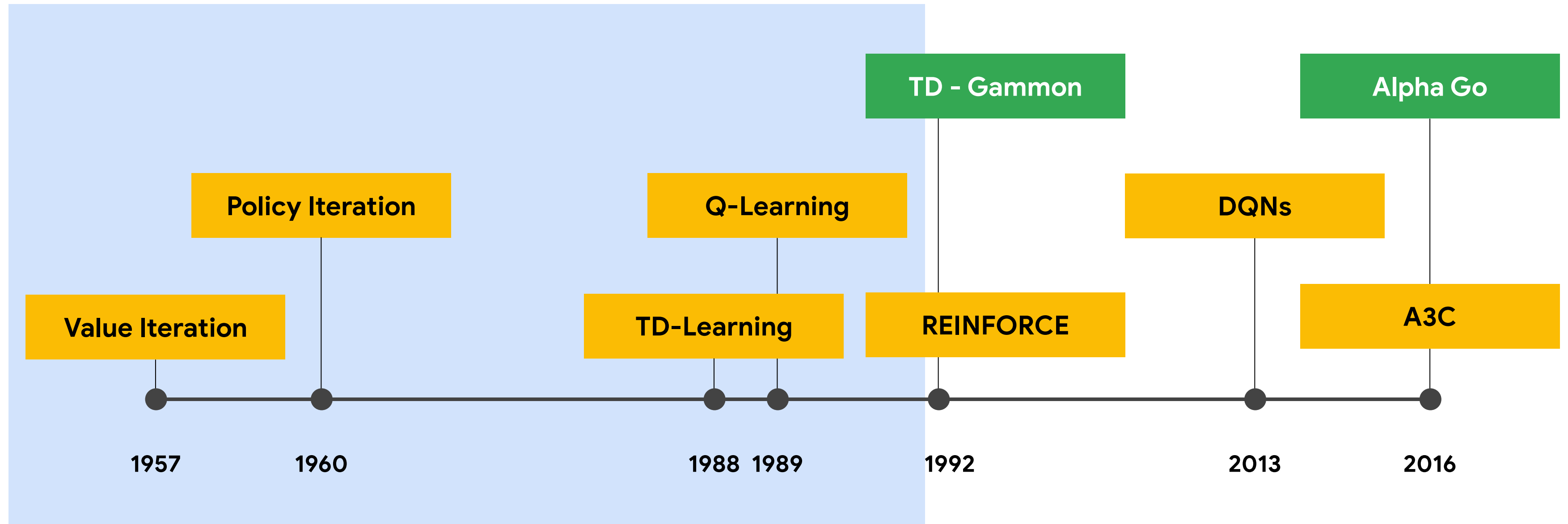
TD(Lambda)

Q-Learning

An RL Timeline



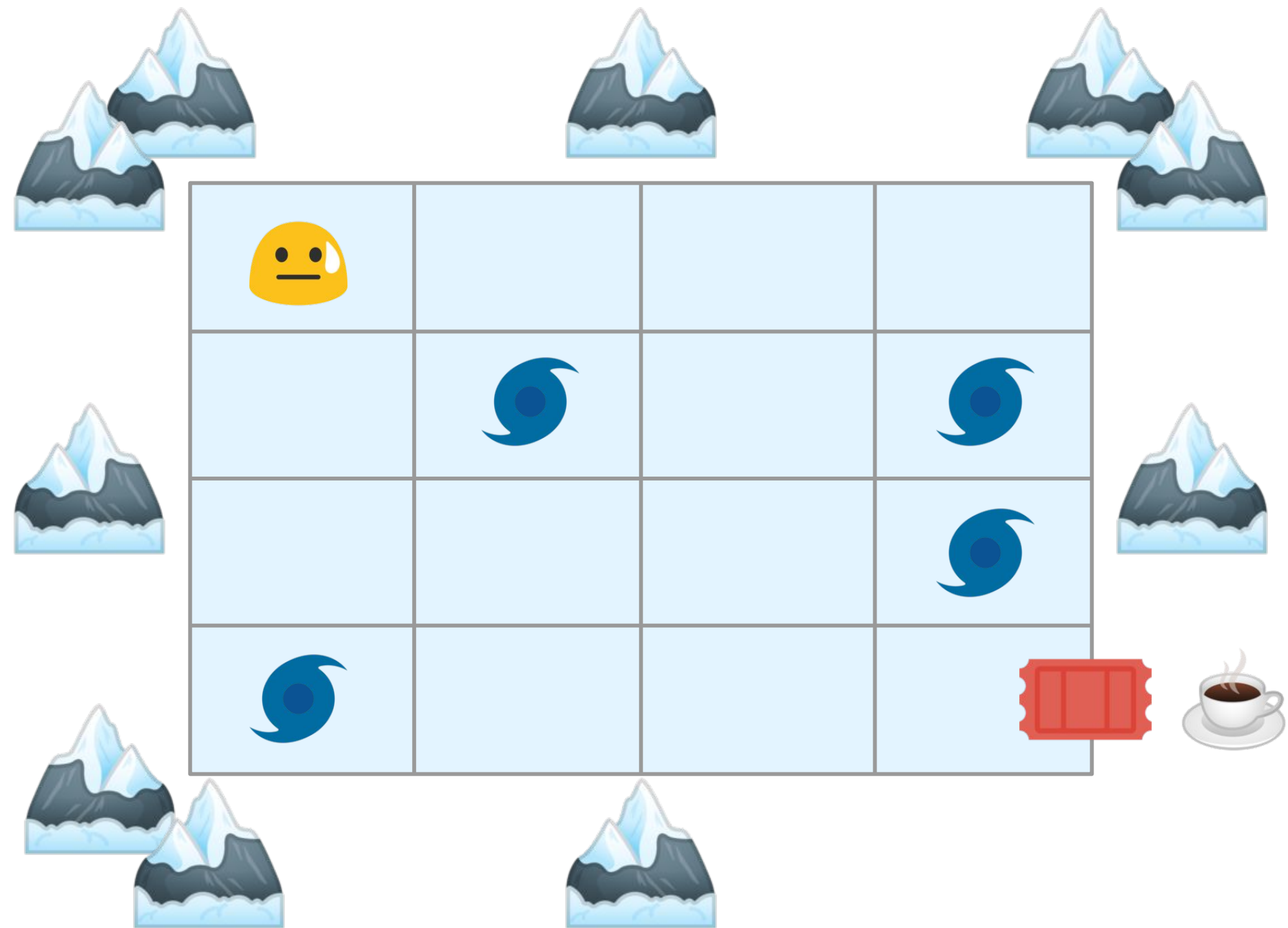
An RL Timeline



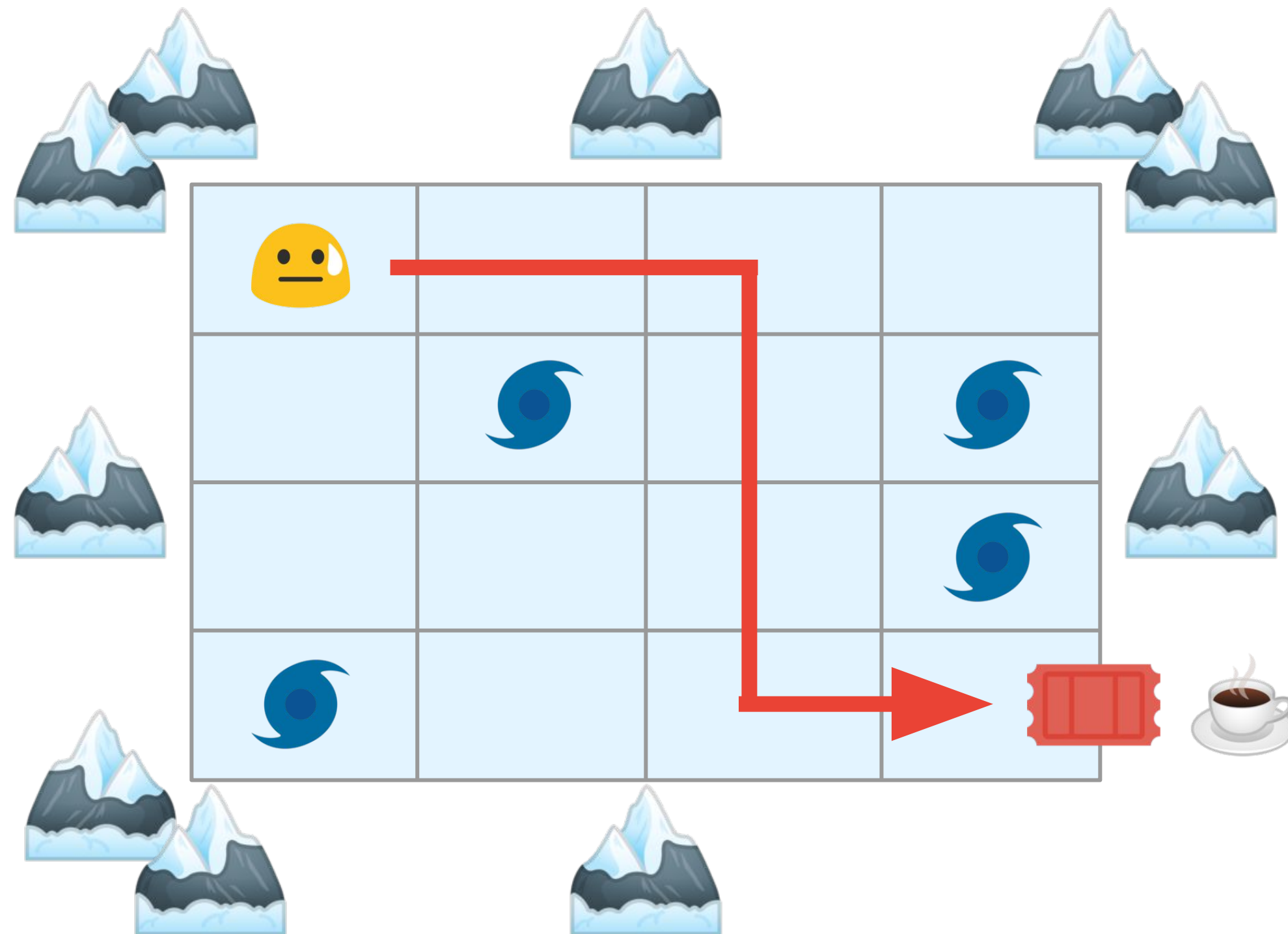
A Simple Story



Frozen Lake



Frozen Lake



Agenda

History Overview

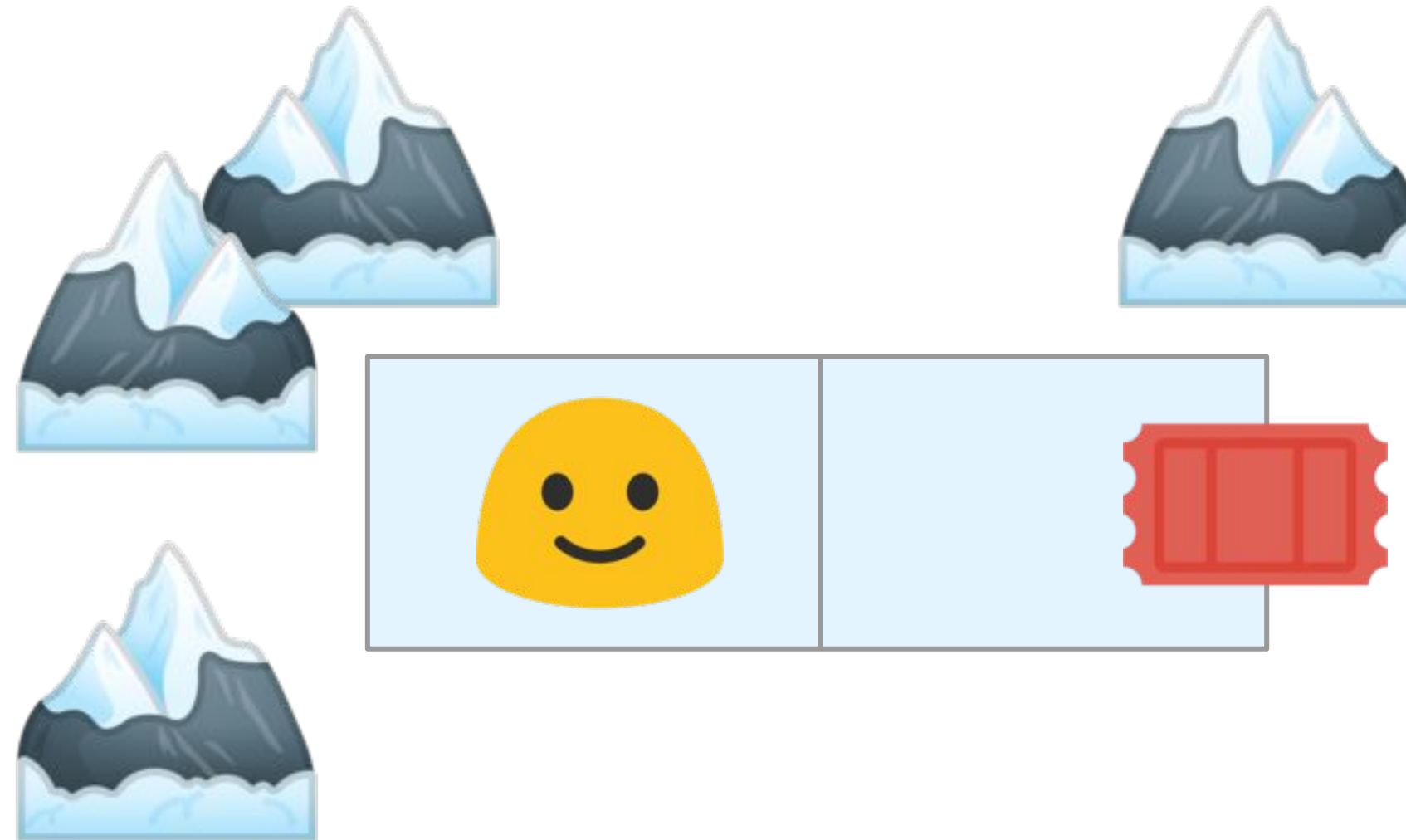
Value Iteration

Policy Iteration

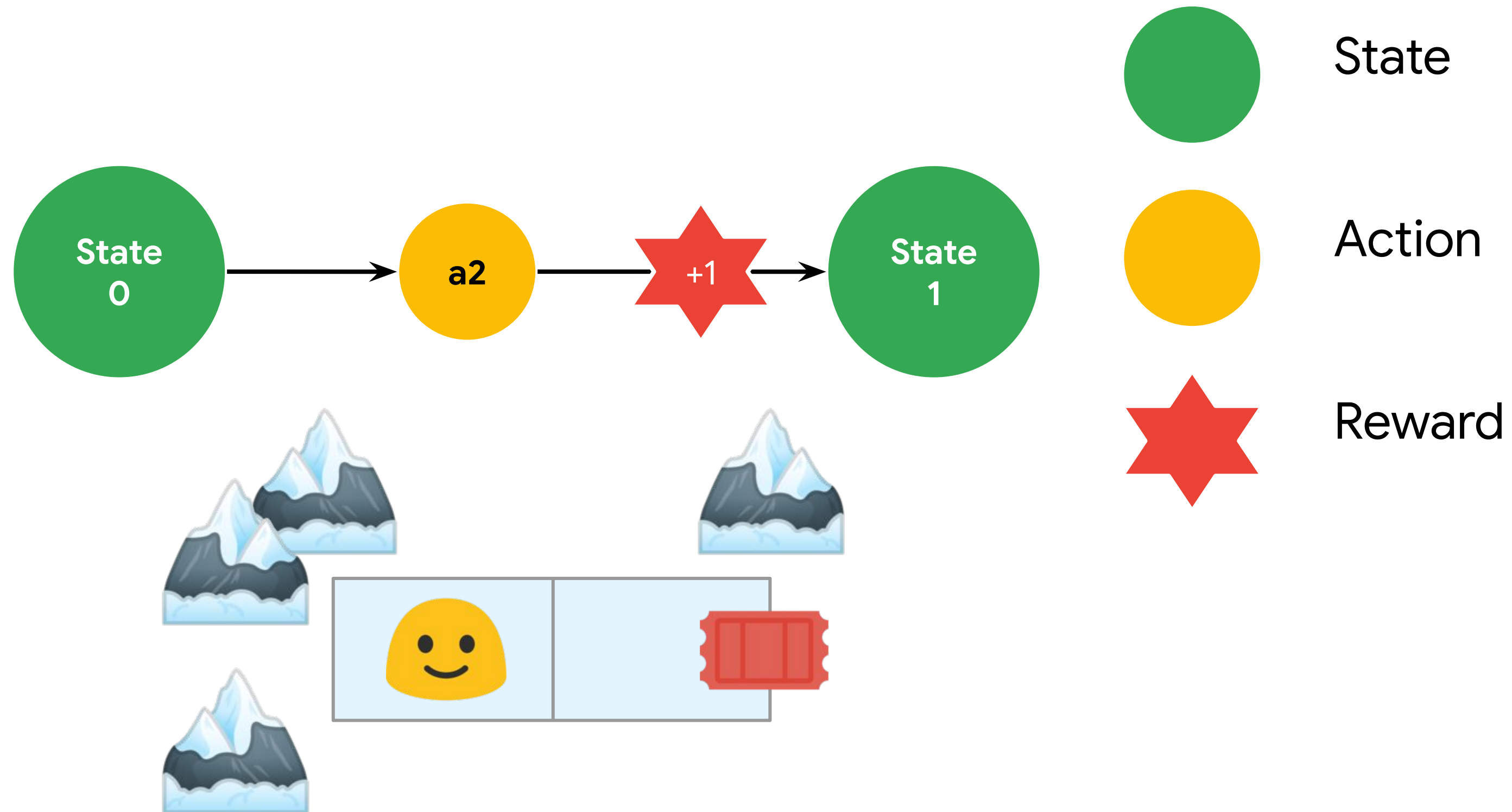
TD(Lambda)

Q-Learning

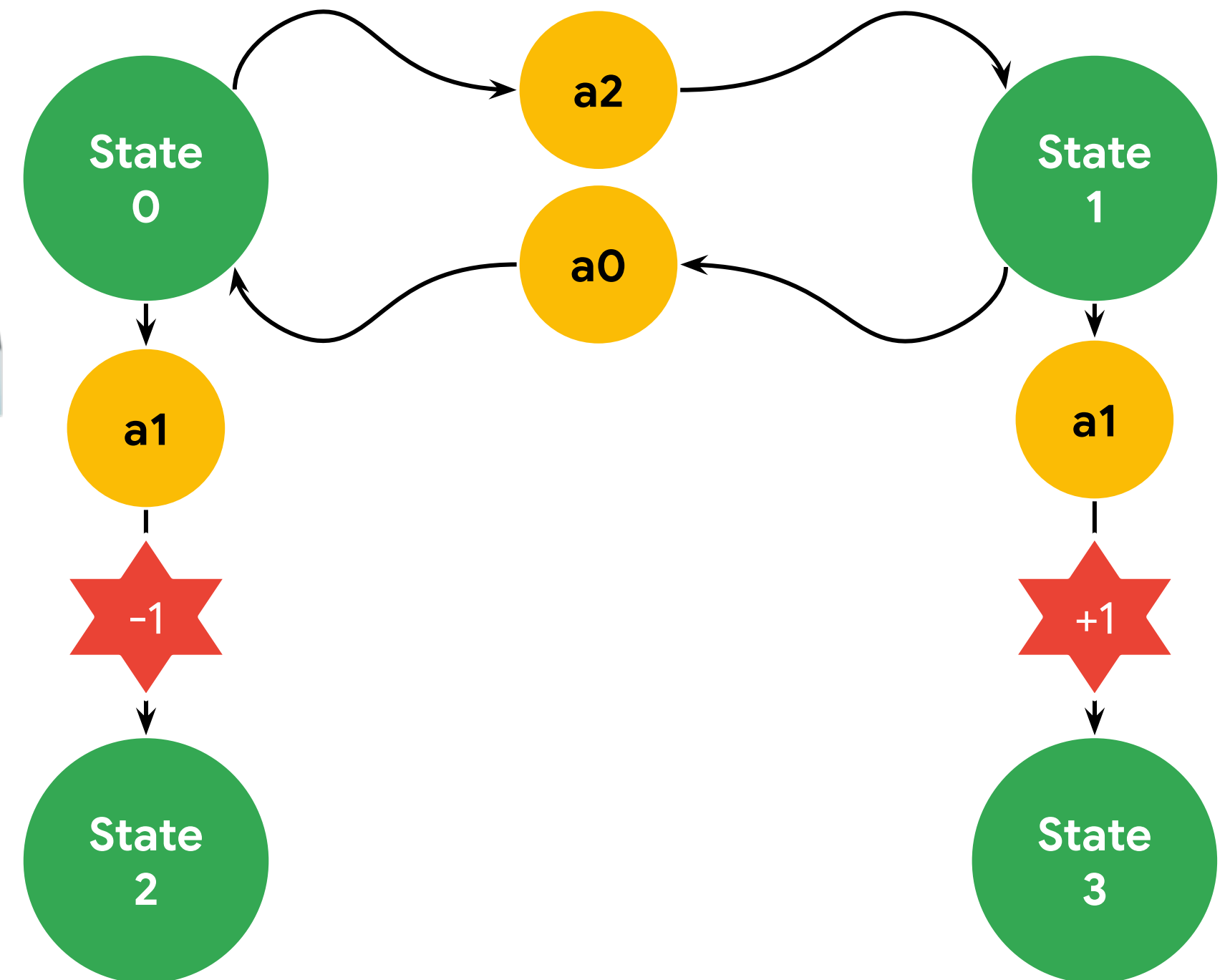
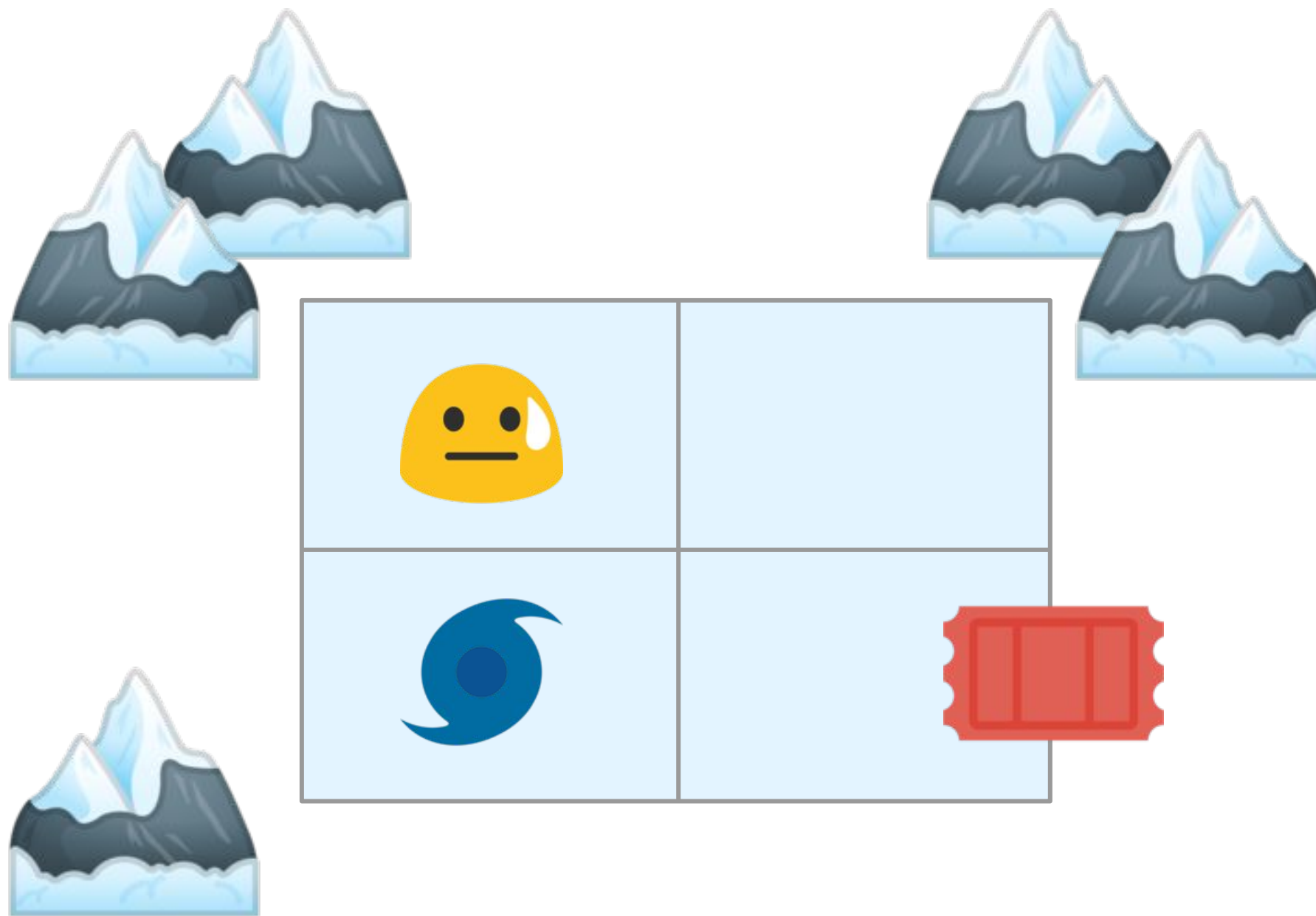
A Simpler Lake



Markov Decision Process (MDP)



A Simpler Lake



Bellman Equation

$$\underline{V(s)} = \frac{\text{Rewards received}}{\quad} + \frac{\gamma V(s')}{\quad}$$

Value of the current state

Discounted future state

The Discount Factor (γ)



γ	Today	Tomorrow	2 days from now	3 days from now	4 days from now
1	\$100	\$100	\$100	\$100	\$100

The Discount Factor (γ)



γ	Today	Tomorrow	2 days from now	3 days from now	4 days from now
1	\$100	\$100	\$100	\$100	\$100
.5	\$100	\$50	\$25	\$12.5	\$6.25

The Discount Factor (γ)

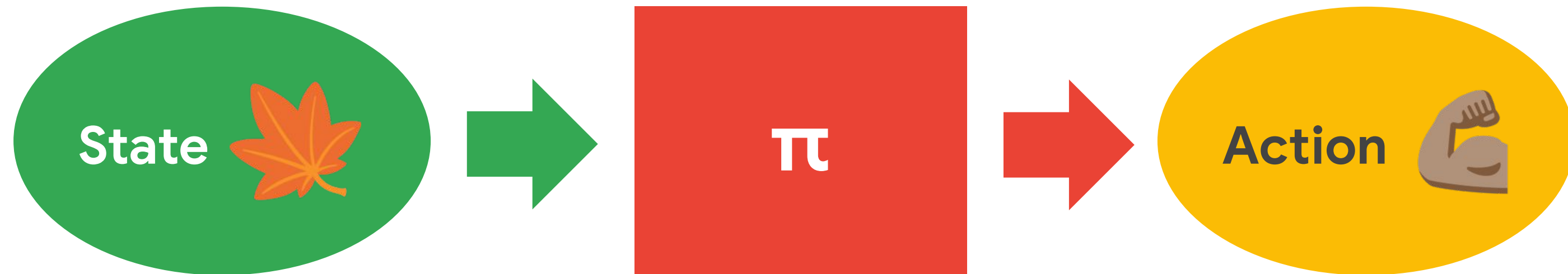


γ	Today	Tomorrow	2 days from now	3 days from now	4 days from now
1	\$100	\$100	\$100	\$100	\$100
.5	\$100	\$50	\$25	\$12.5	\$6.25
0	\$100	\$0	\$0	\$0	\$0

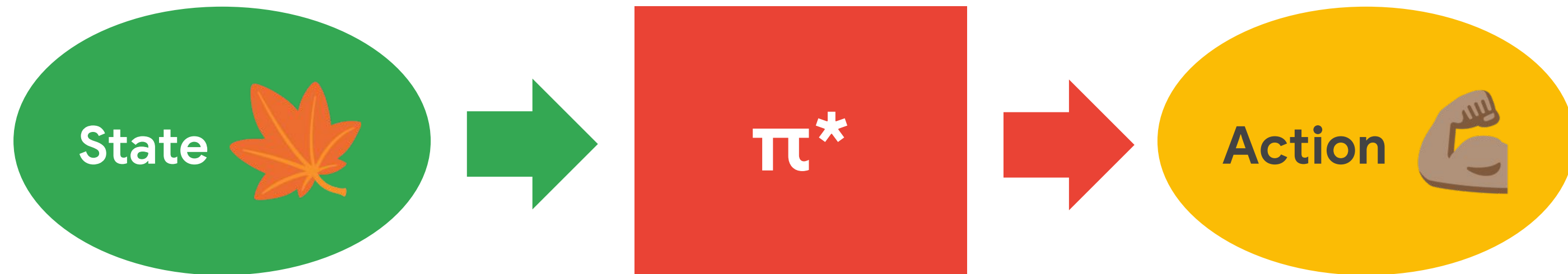
Bellman Equation

$$V(s) = R(s, a) + \gamma V(s')$$

The Policy



The Policy

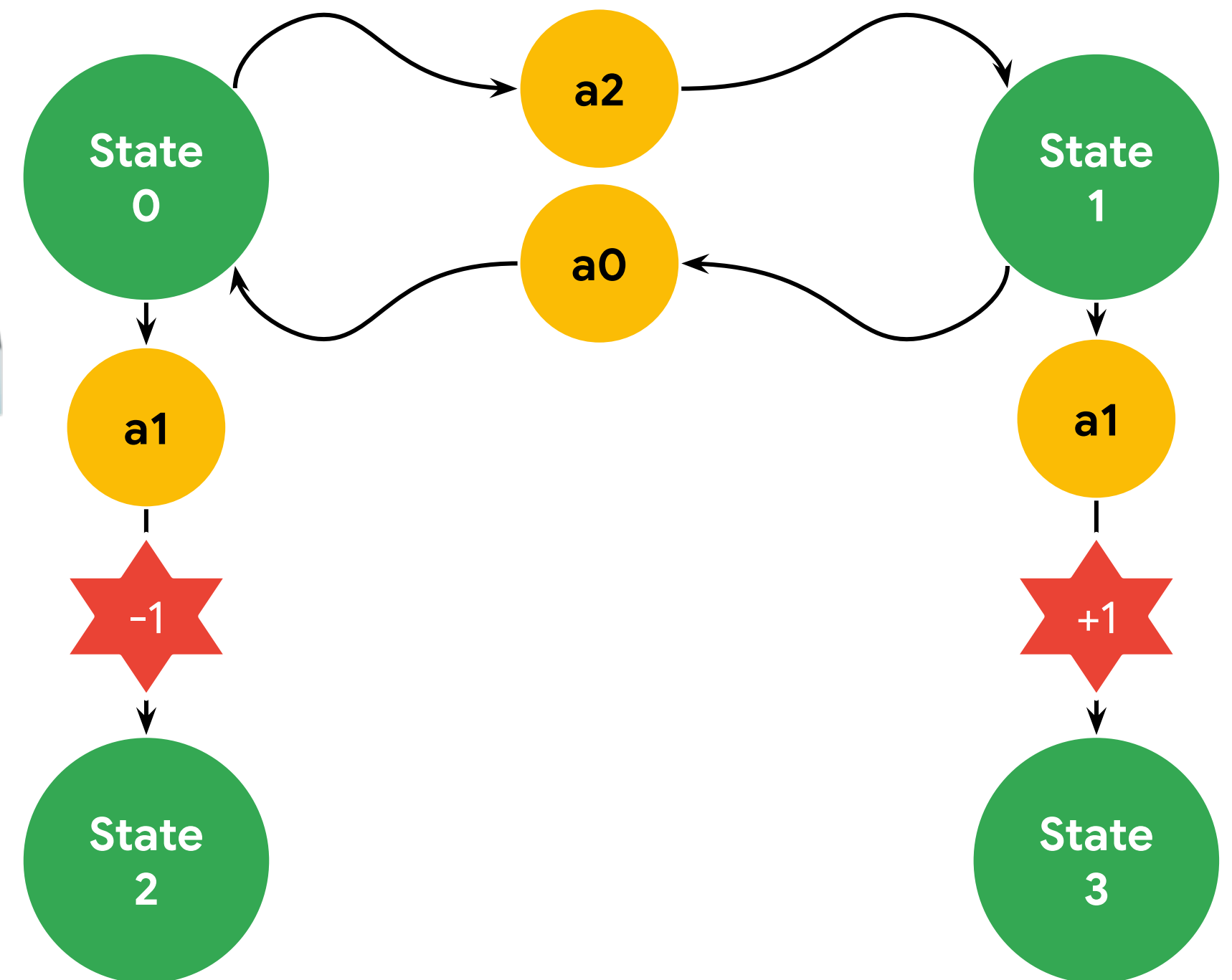
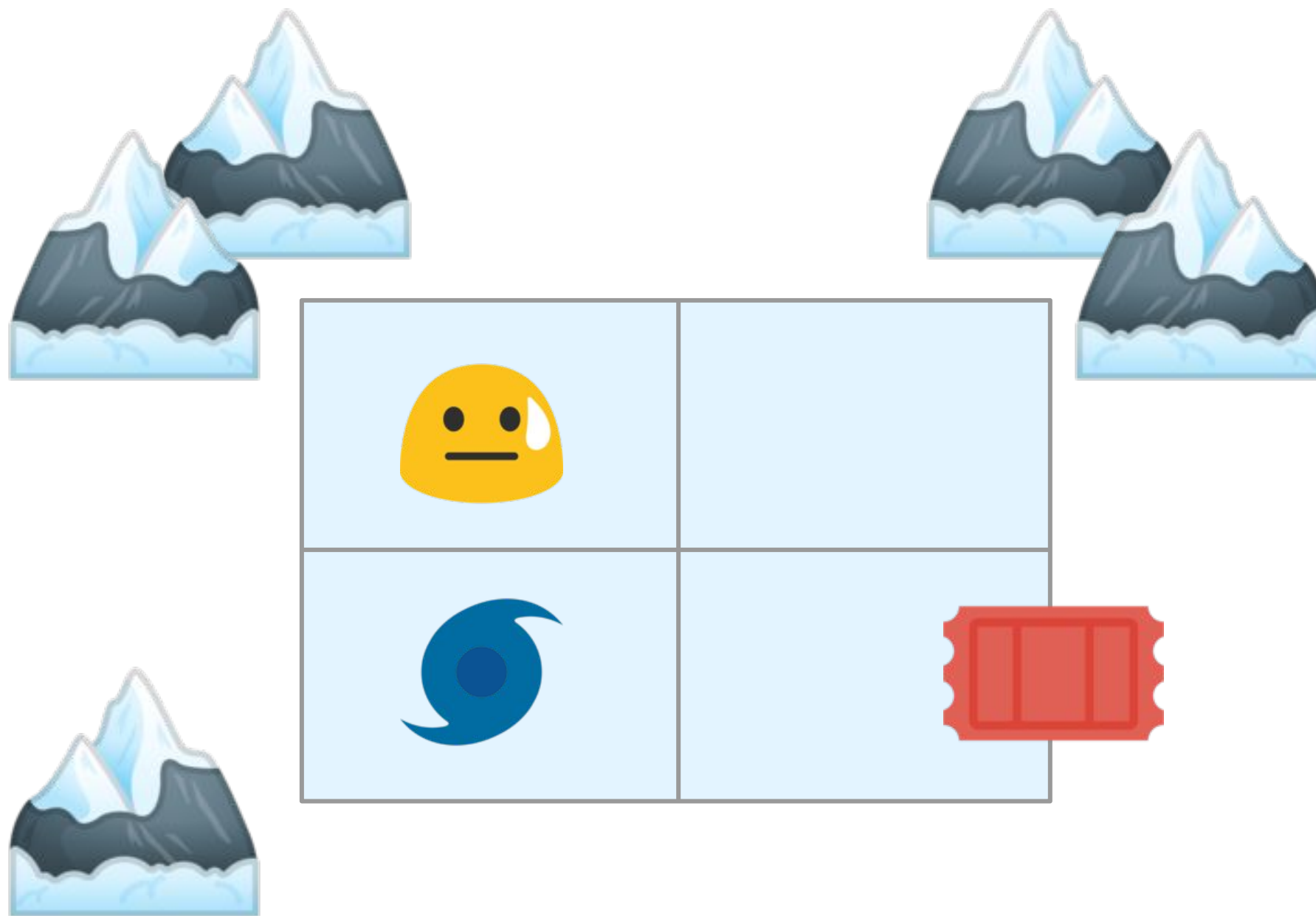


Bellman Equation

$$\overline{V^{\pi^*}}(s) = \underline{\max_a} \{R(s, a) + \gamma \overline{V^{\pi^*}}(s')\}$$

—— = new addition

Simple Lake Value



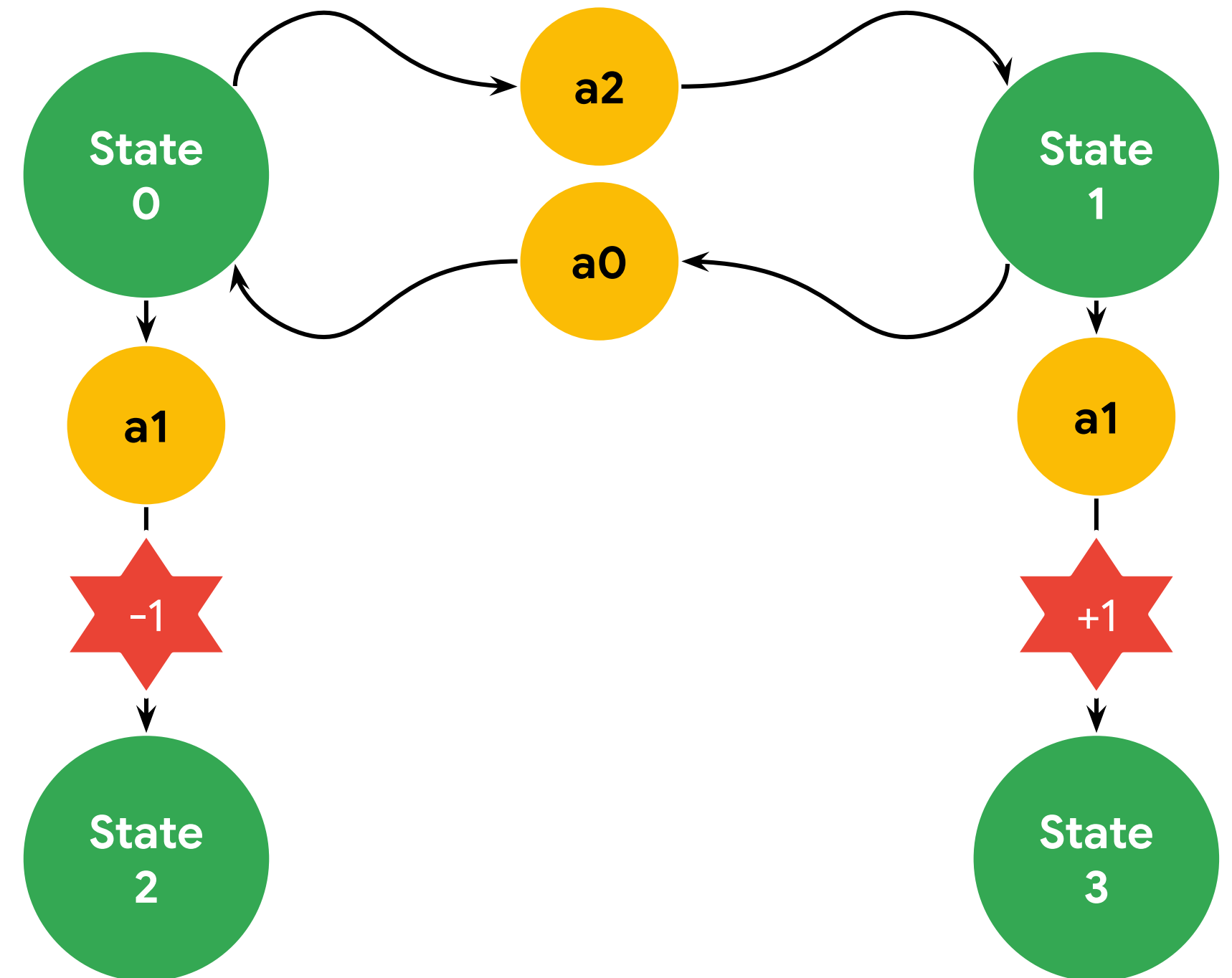
Simple Lake Value

State Map	
0	1
2	3

Current Value	
0	0
0	0

Policy Map	
?	?
-	-

Prime Value	
0	0
0	0



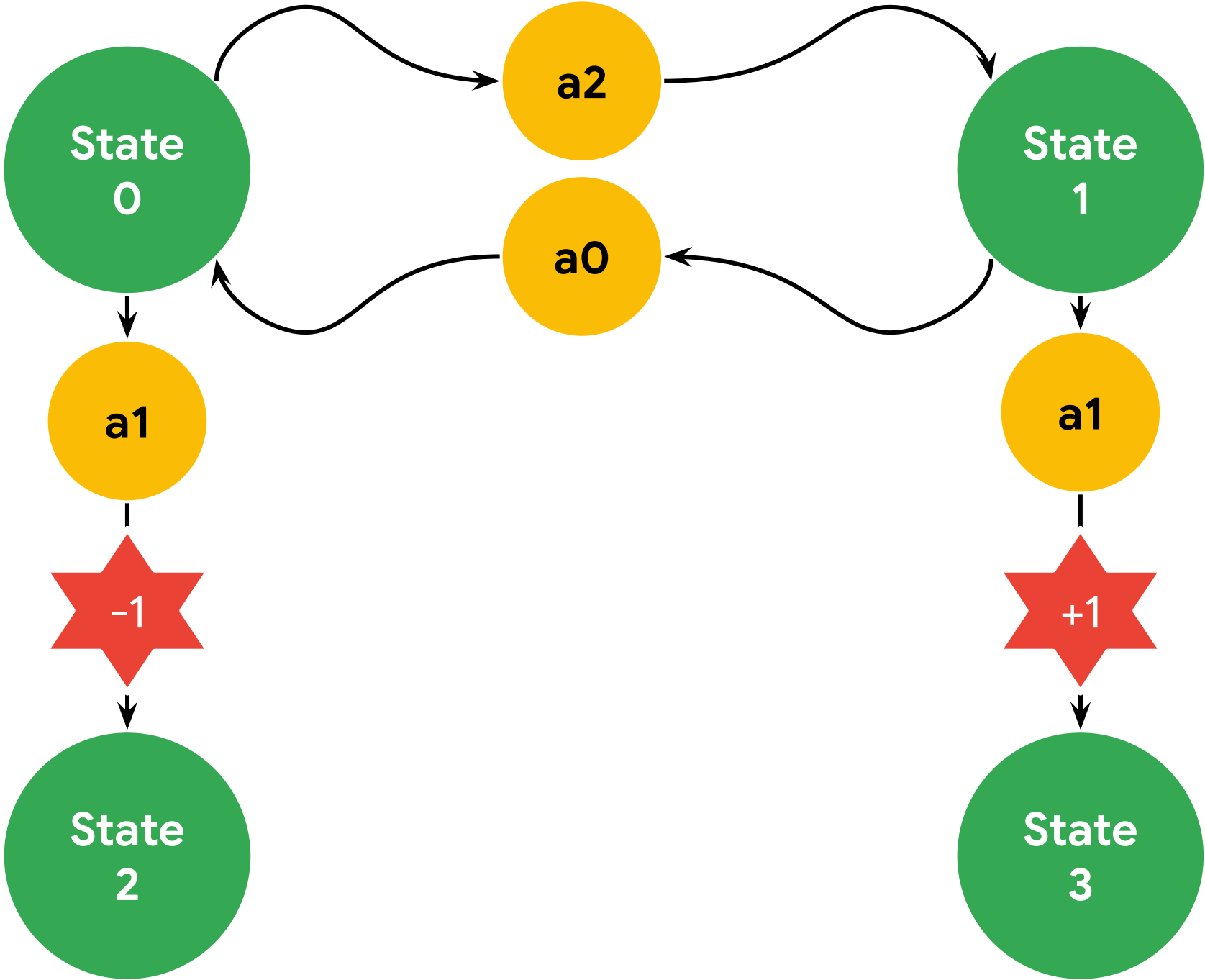
Simple Lake Value

State Map	
<u>0</u>	1
2	3

Current Value	
0	0
0	0

Policy Map	
?	?
-	-

Prime Value	
0	0
0	0



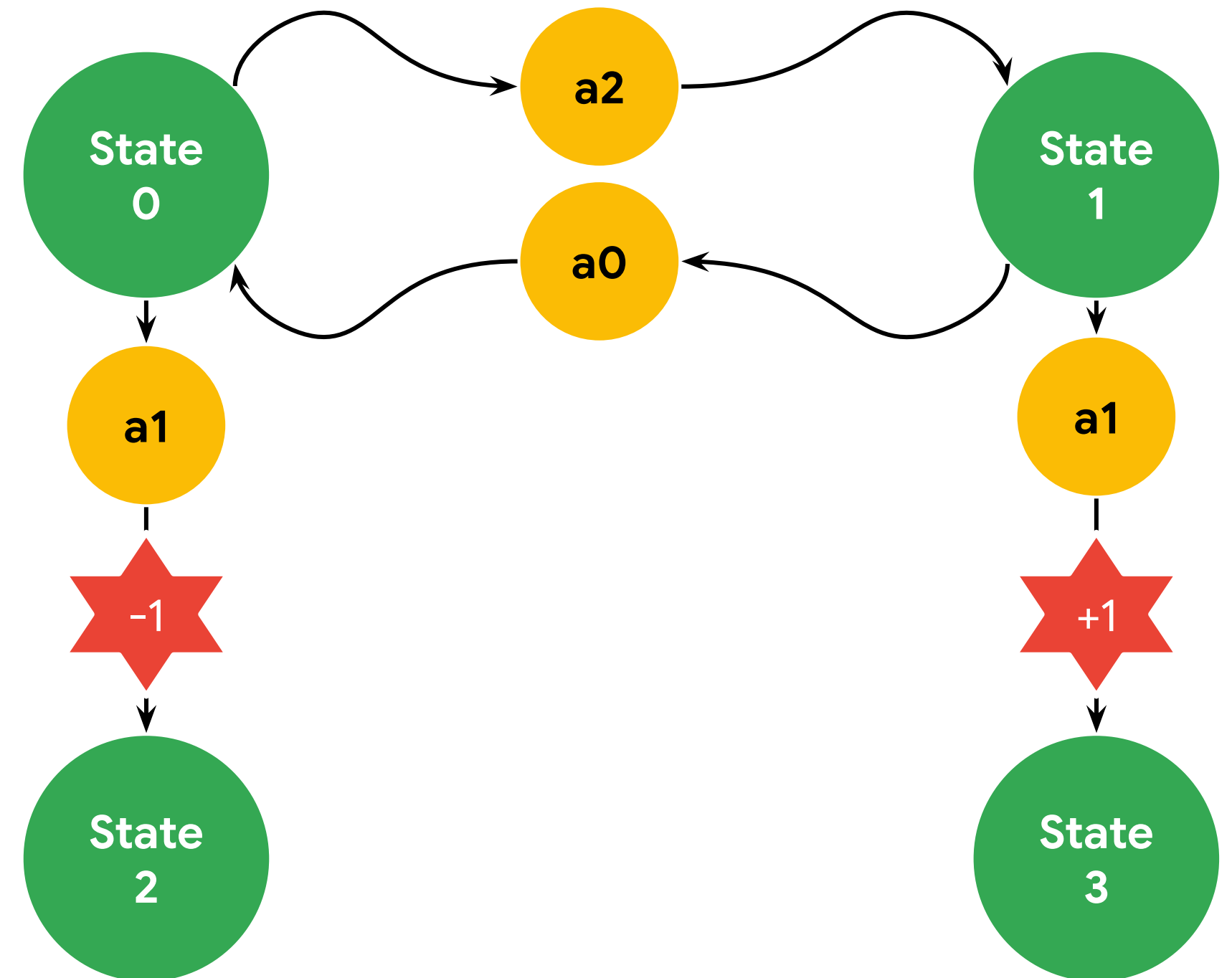
Simple Lake Value

State Map	
<u>0</u>	1
2	3

Current Value	
0	<u>0</u>
<u>0</u>	0

Policy Map	
?	?
-	-

Prime Value	
0	0
0	0



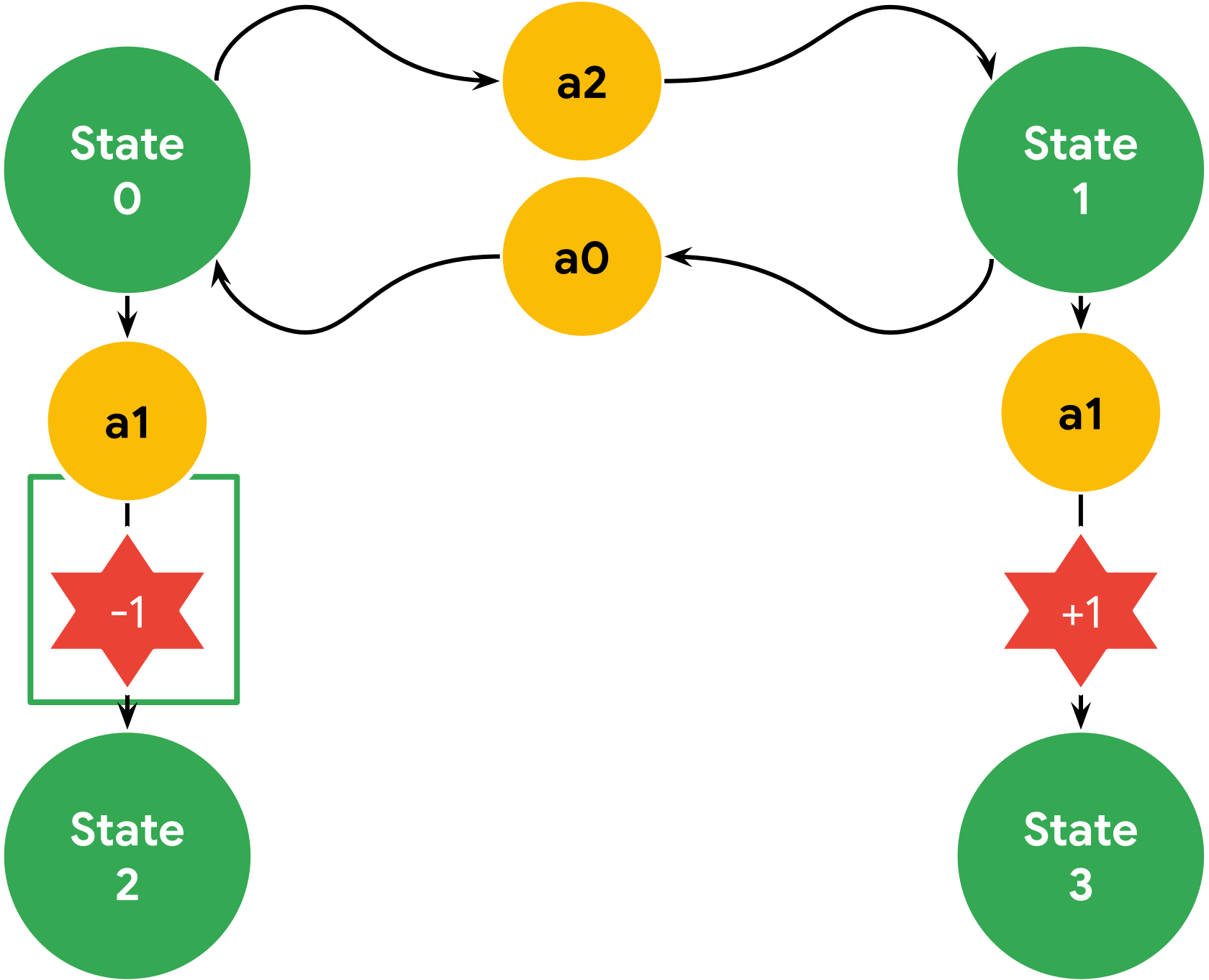
Simple Lake Value

State Map	
<u>0</u>	1
2	3

Current Value	
0	<u>0</u>
<u>0</u>	0

Policy Map	
a2	?
-	-

Prime Value	
<u>0</u>	0
0	0



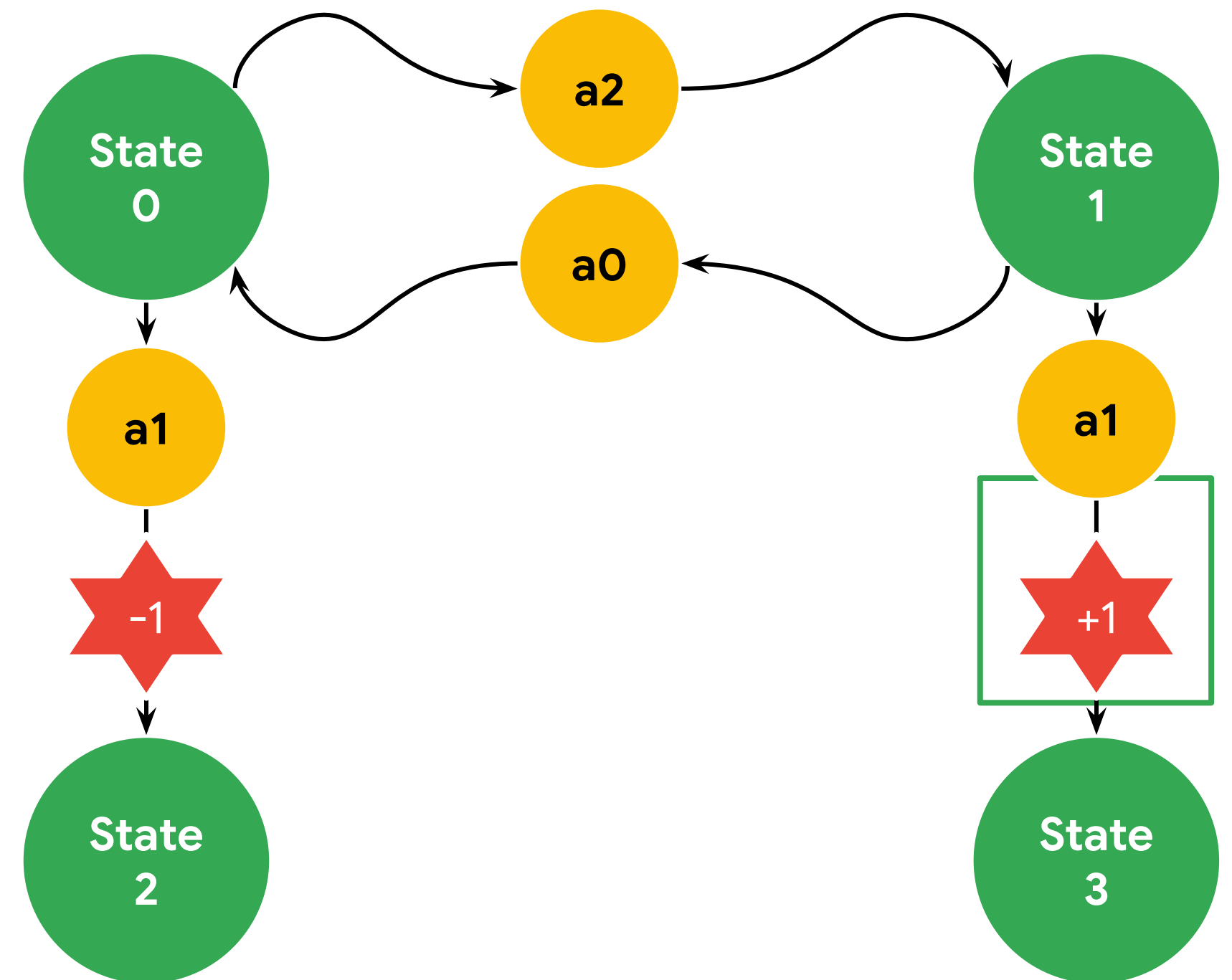
Simple Lake Value

State Map	
0	1
2	3

Current Value	
<u>0</u>	0
0	<u>0</u>

Policy Map	
a2	a1
-	-

Prime Value	
0	1
0	0



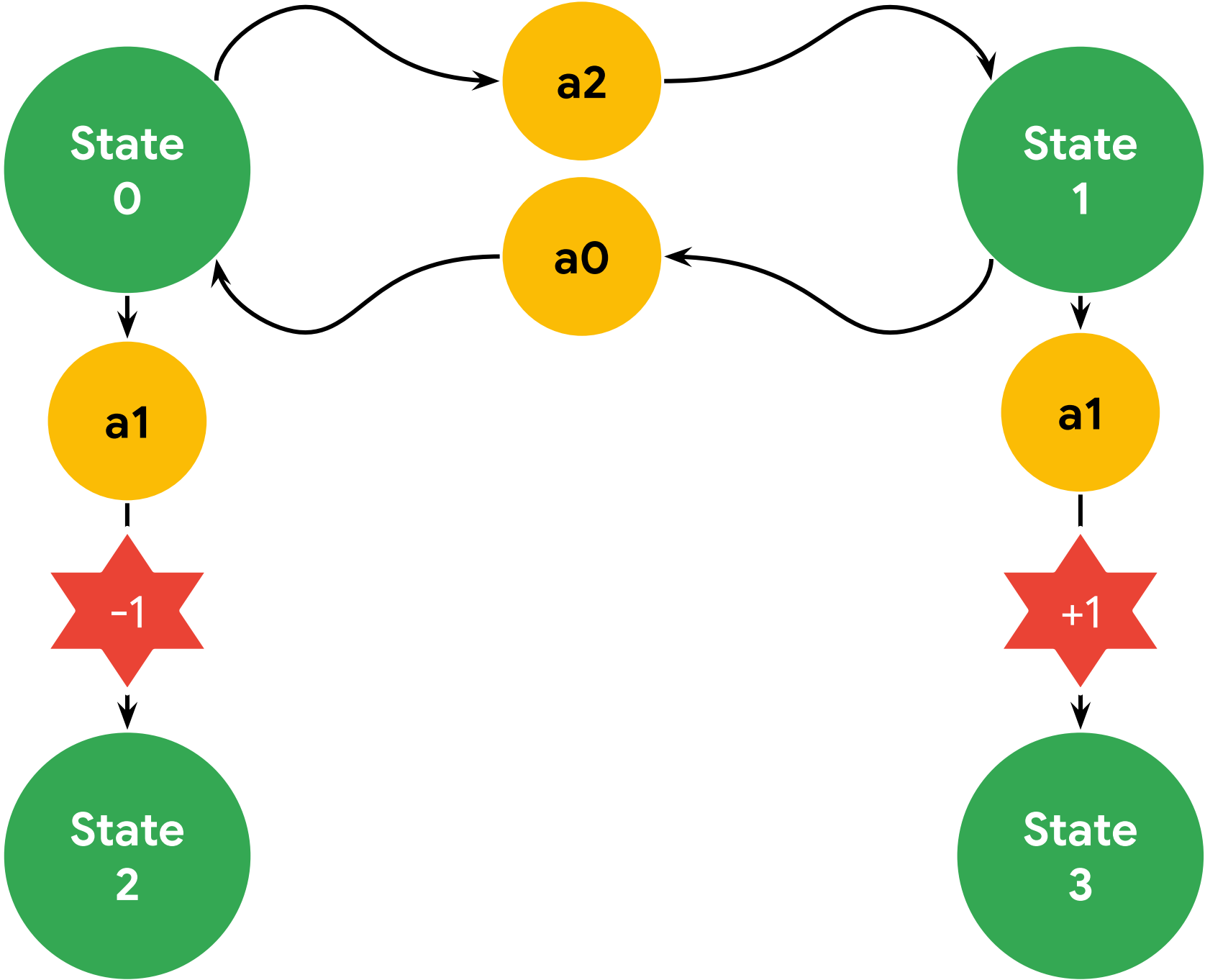
Simple Lake Value

State Map	
0	1
<u>2</u>	<u>3</u>

Current Value	
0	0
0	0

Policy Map	
?	?
-	-

Prime Value	
0	1
<u>0</u>	<u>0</u>



Simple Lake Value

State Map	
0	1
2	3

Current Value	
0	0
0	0

Policy Map	
a2	a1
-	-

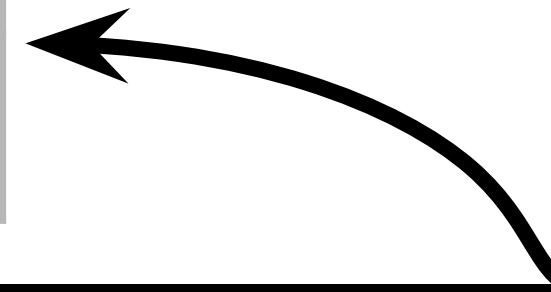
Prime Value	
0	1
0	0

$$\gamma = .9$$


Simple Lake Value (more accurate)

State Map	
0	1
2	3

Current Value	
0	0
0	0



Policy Map	
a2	a1
-	-

Prime Value	
0	1
0	0

×

$$\gamma = .9$$

+

Rewards	
0	1
0	0

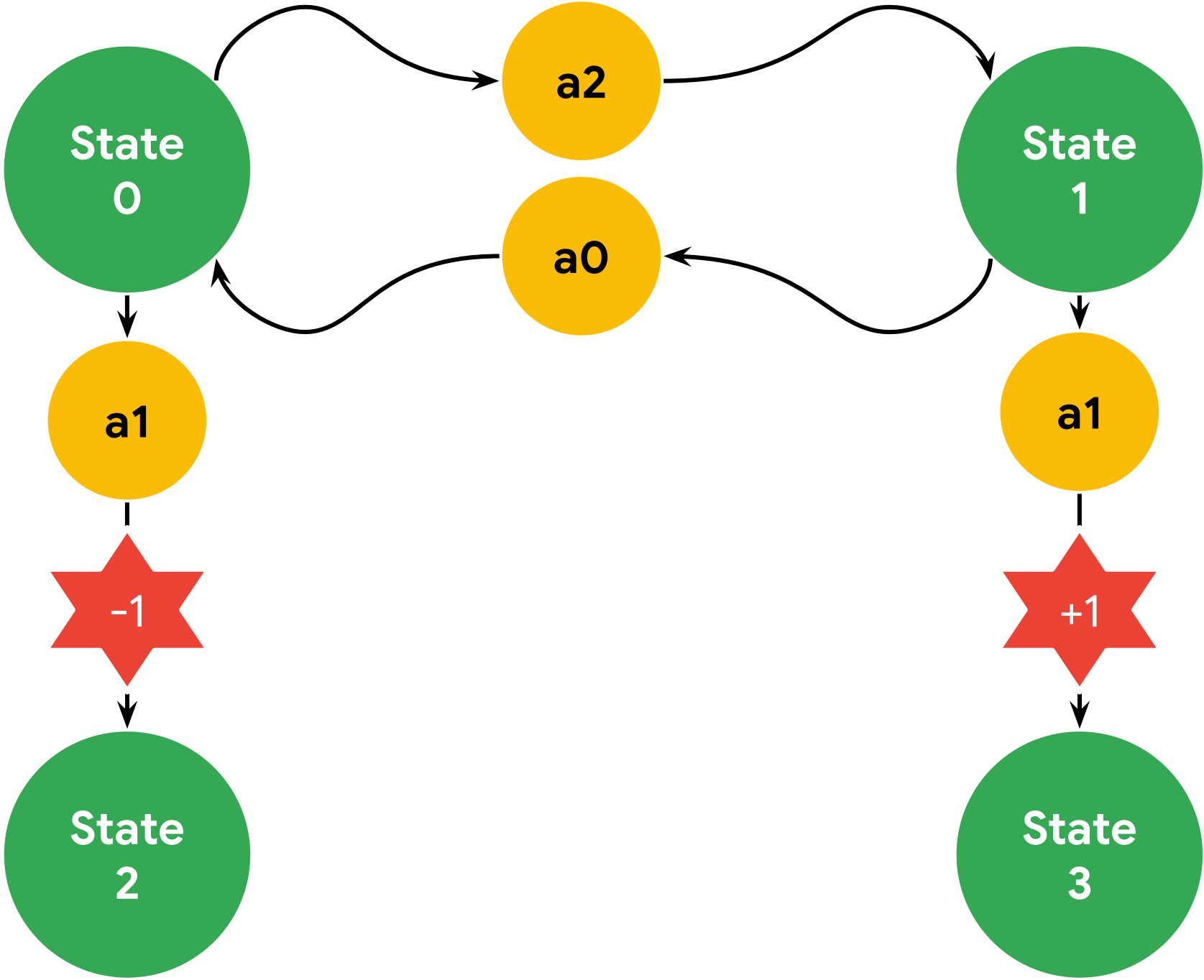
Simple Lake Value (1 iteration)

State Map	
0	1
2	3

Current Value	
0	<u>.9</u>
0	0

Policy Map	
a2	a1
-	-

Prime Value	
0	0
0	0



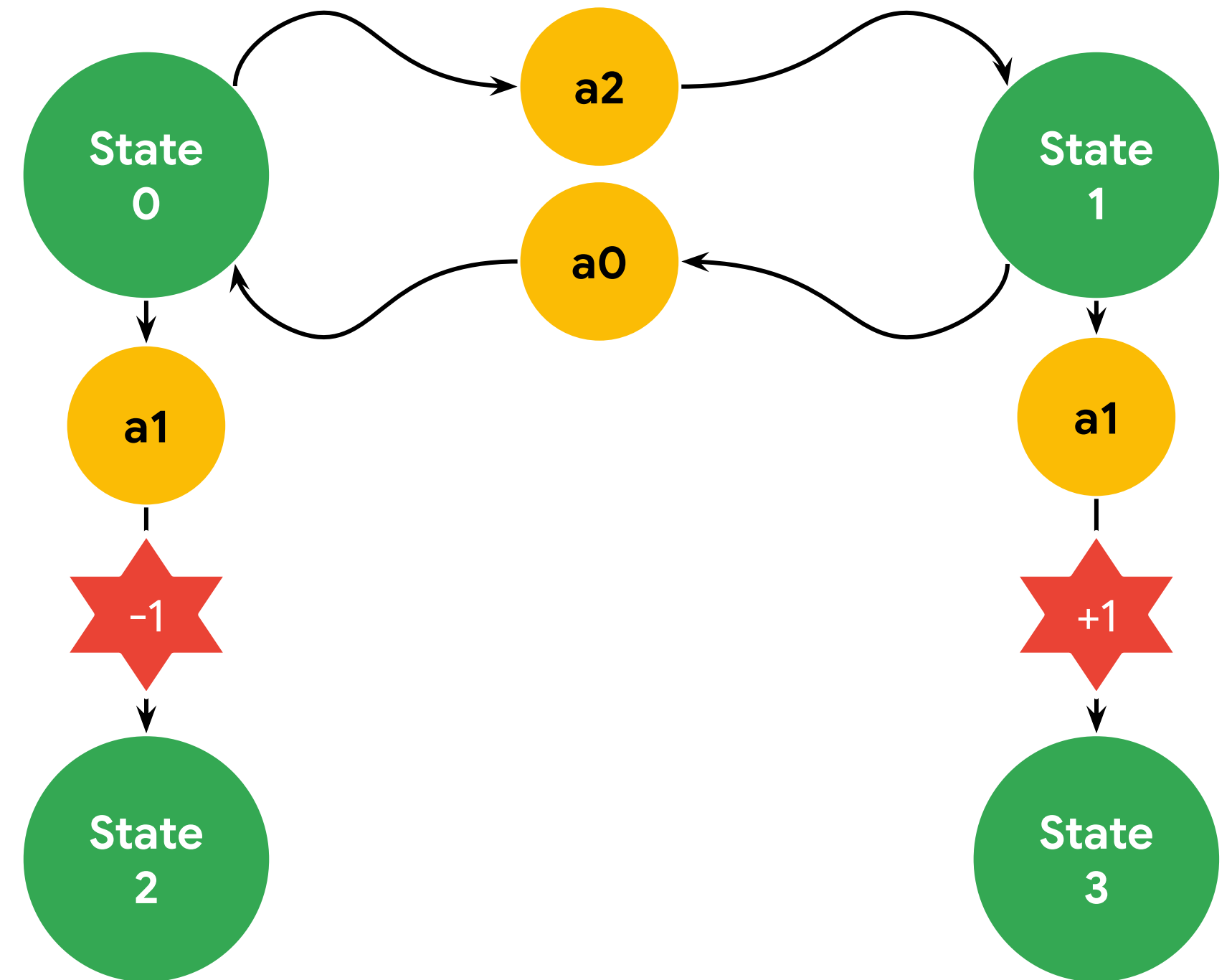
Simple Lake Value (2 and 3 iterations)

State Map	
0	1
2	3

Current Value	
<u>.81</u>	.9
0	0

Policy Map	
a2	a1
-	-

Prime Value	
0	0
0	0



Value Iteration Code

```
LAKE = np.array([[0, 0, 0, 0],
                 [0, -1, 0, -1],
                 [0, 0, 0, -1],
                 [-1, 0, 0, 1]])

LAKE_WIDTH = len(LAKE[0])
LAKE_HEIGHT = len(LAKE)

DISCOUNT = .9 # Change me to be a value between 0 and 1.
DELTA = .0001 # I must be sufficiently small.
current_values = np.zeros_like(LAKE)

while change > DELTA:
    prime_values, policies = iterate_value(current_values)
    old_values = np.copy(current_values)
    current_values = DISCOUNT * prime_values
    change = np.sum(np.abs(old_values - current_values))
```


Value Iteration Code

```
def iterate_value(current_values):  
    """Finds the future state values for an array of current states.  
  
    Args:  
        current_values (int array): the value of current states.  
  
    Returns:  
        prime_values (int array): The value of states based on future states.  
        policies (int array): The recommended action to take in a state.  
    """  
    prime_values = []  
    policies = []  
  
    for state in STATE_RANGE:  
        value, policy = get_max_neighbor(state, current_values)  
        prime_values.append(value)  
        policies.append(policy)  
  
    prime_values = np.array(prime_values).reshape((LAKE_HEIGHT, LAKE_WIDTH))  
    return prime_values, policies
```

Value Iteration Code

Lake			
0	0	0	0
0	-1	0	-1
0	0	0	-1
-1	0	0	1

Iteration 6			
<u>.53</u>	.59	.66	.59
.59	0	.73	0
.66	.73	.81	0
0	.81	.9	0

Optimal Policy			
1	2	1	0
1	-	1	-
2	1	1	-
-	2	2	-

Agenda

History Overview

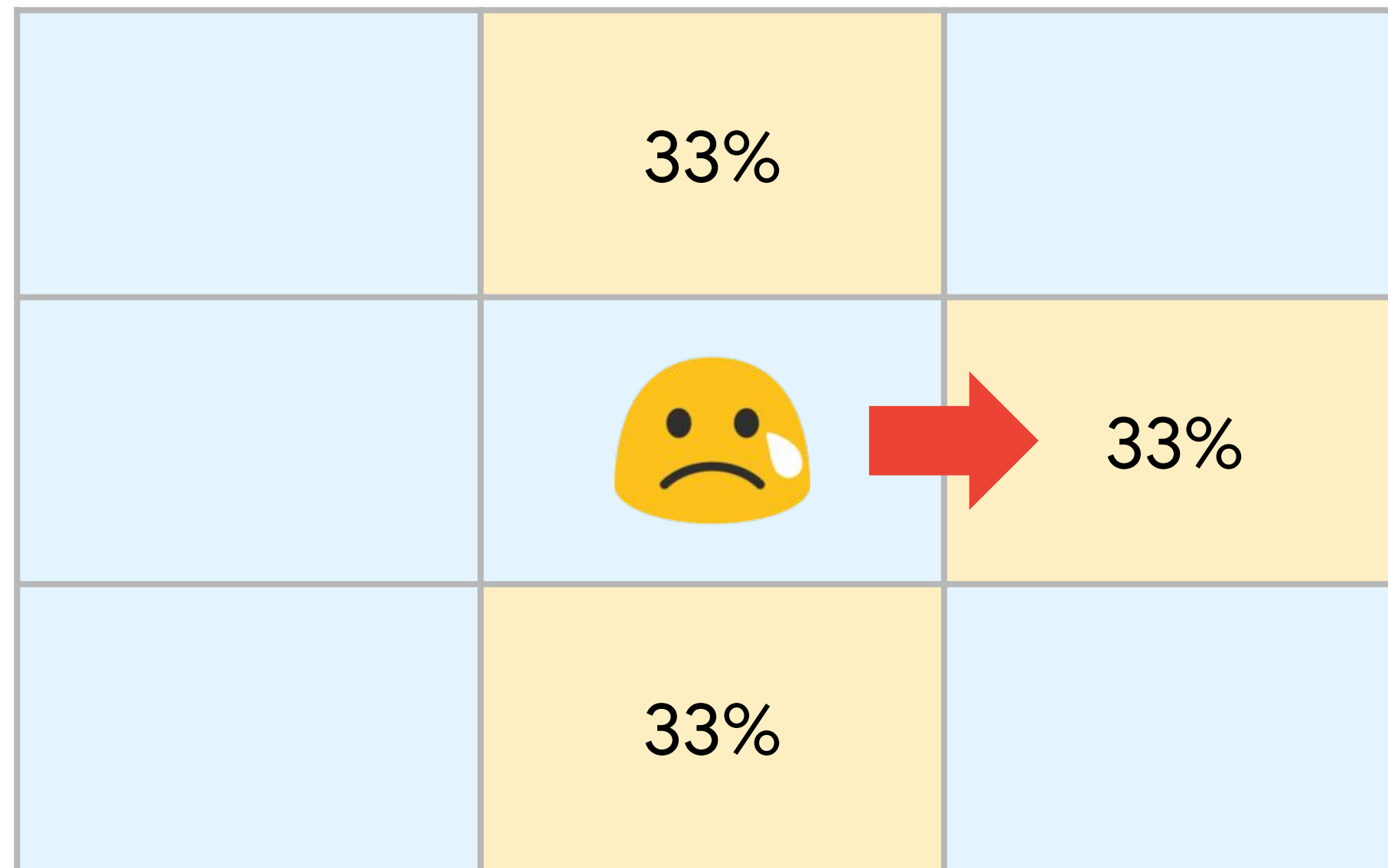
Value Iteration

Policy Iteration

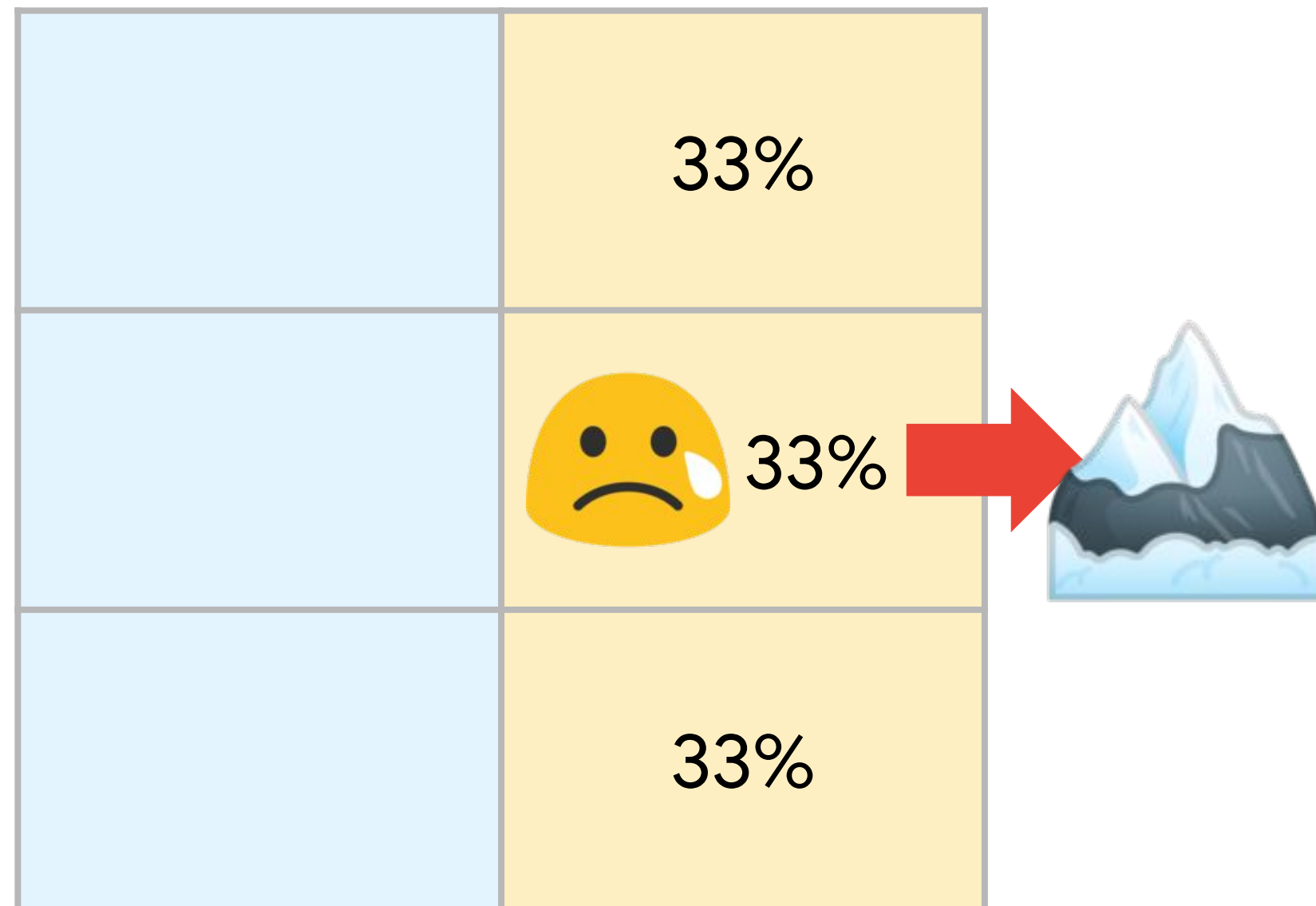
TD(Lambda)

Q-Learning

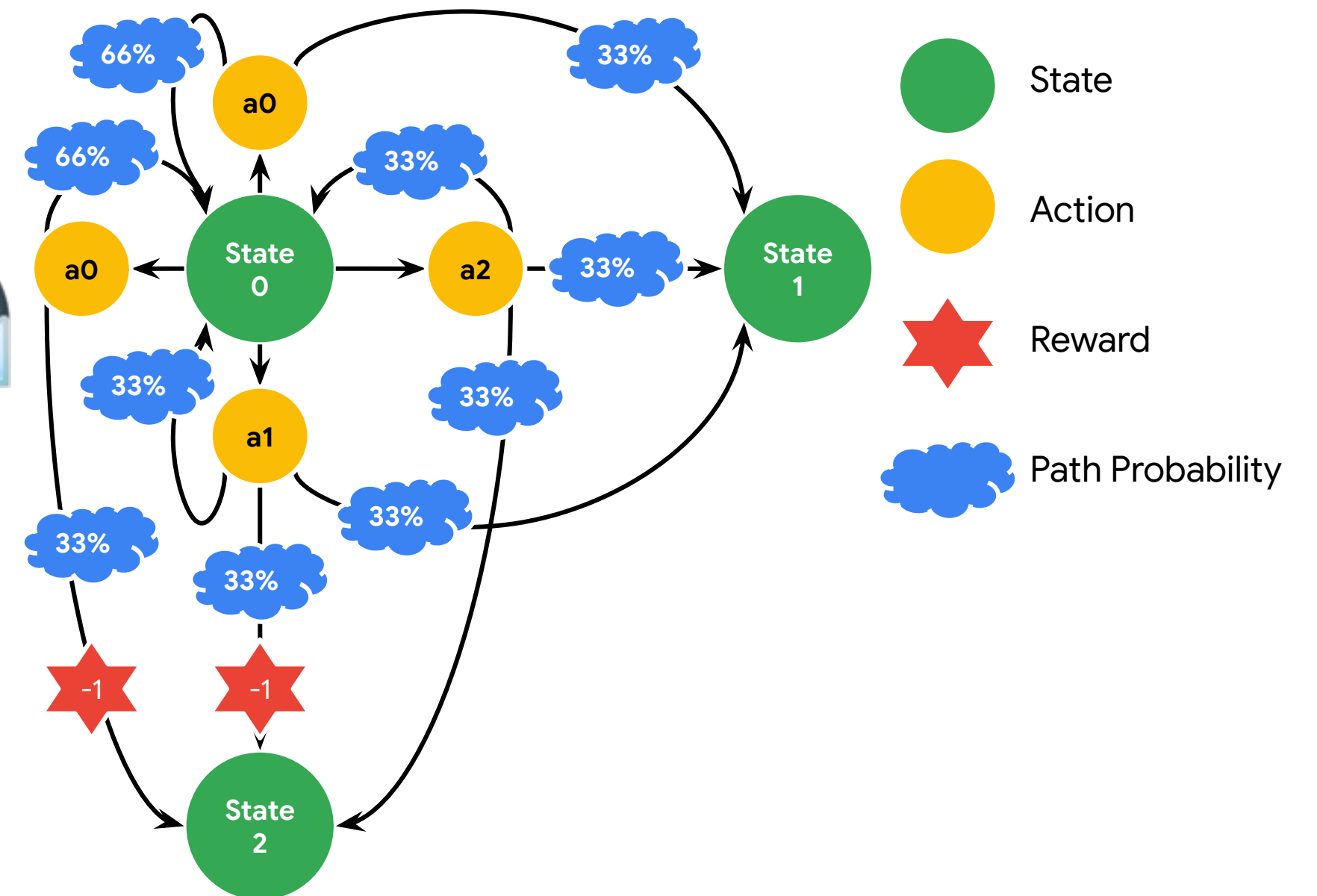
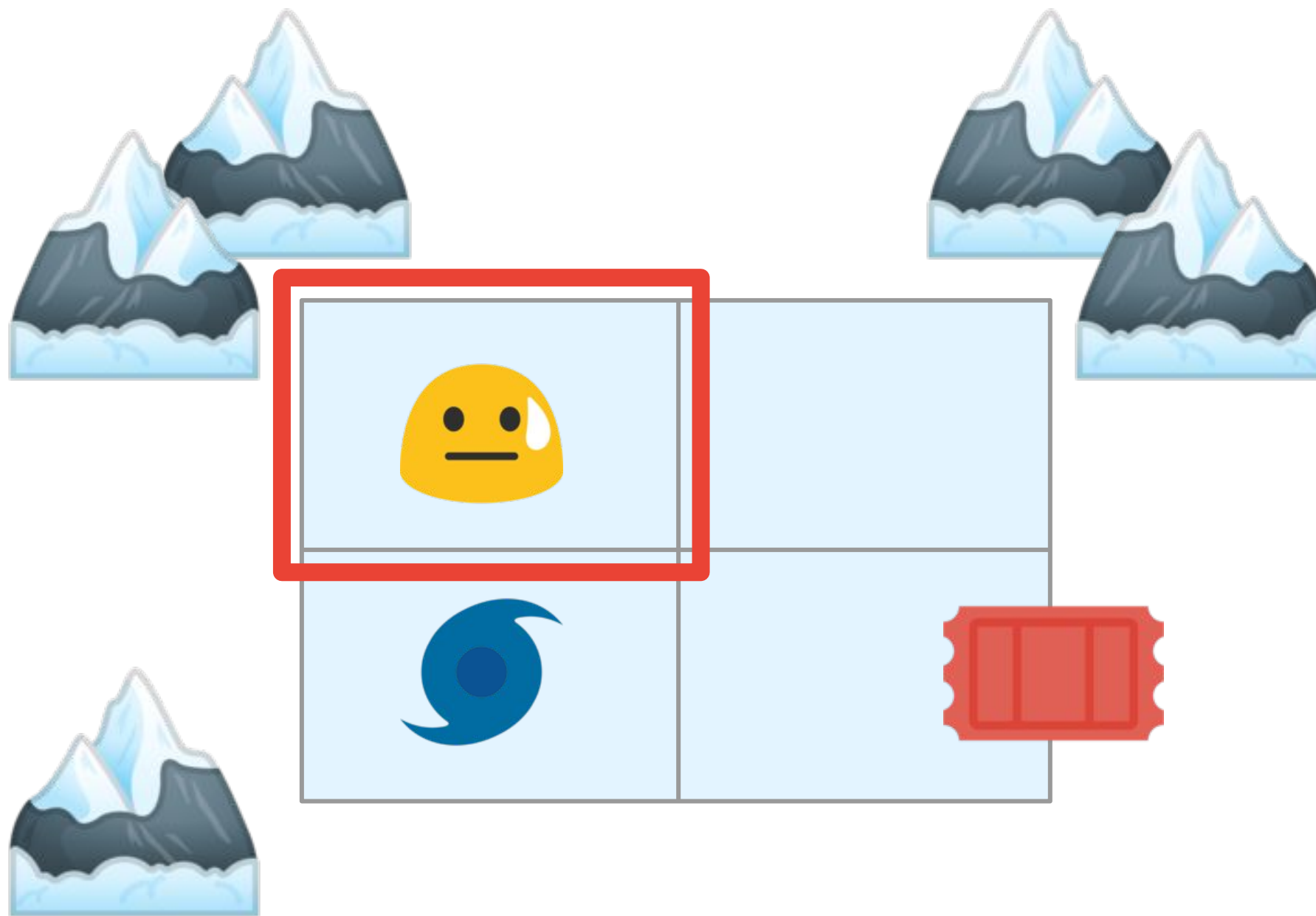
Probabilities and Slipping



Probabilities and Slipping



Slippery Simple Lake



Bellman Equation

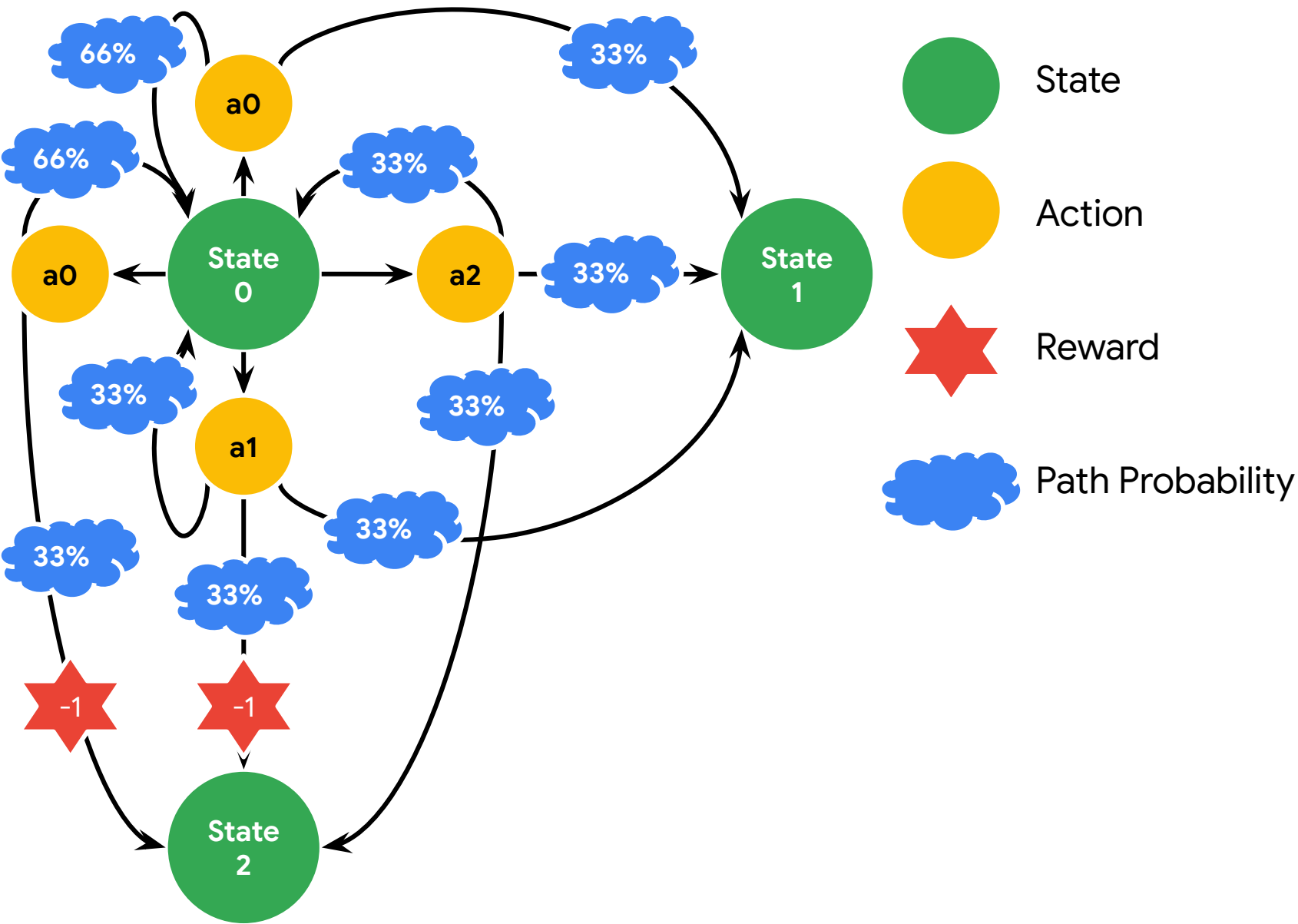
$$V^{\pi^*}(s) = \max_a \left\{ R(s, a) + \gamma \sum_{\underline{s'}} P(s'|s, a) V^{\pi^*}(s') \right\}$$

 = new addition

Weighting State Prime

$$\sum_{s'} P(s'|s,a) V^{\pi^*}(s')$$

Action	Counter Clockwise	Forward	Clockwise
a0	s2	s0	s0
a1	s1	s2	s0
a2	s0	s1	s2
a3	s0	s0	s1



Weighting State Prime

$$\sum_{s'} P(s'|s, a) V^{\pi^*}(s') = .33 \cdot V(\text{Counter Clockwise}) + .33 \cdot V(\text{Forward}) + .33 \cdot V(\text{Clockwise})$$

Action	Counter Clockwise	Forward	Clockwise	V(Counter Clockwise)	V(Forward)	V(Clockwise)	Weighted Total
a0	s2	s0	s0	-1	0	0	-.33
a1	s1	s2	s0	0	-1	0	-.33
a2	s0	s1	s2	0	0	-1	-.33
a3	s0	s0	s1	0	0	0	0

Value Iteration Complexity

$$O(s^2as')$$

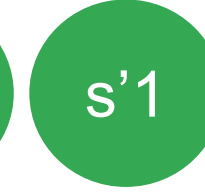
For each state ...



Compare each
action ...



By weighting
each new state ...



Repeat up to the
total number of
states.



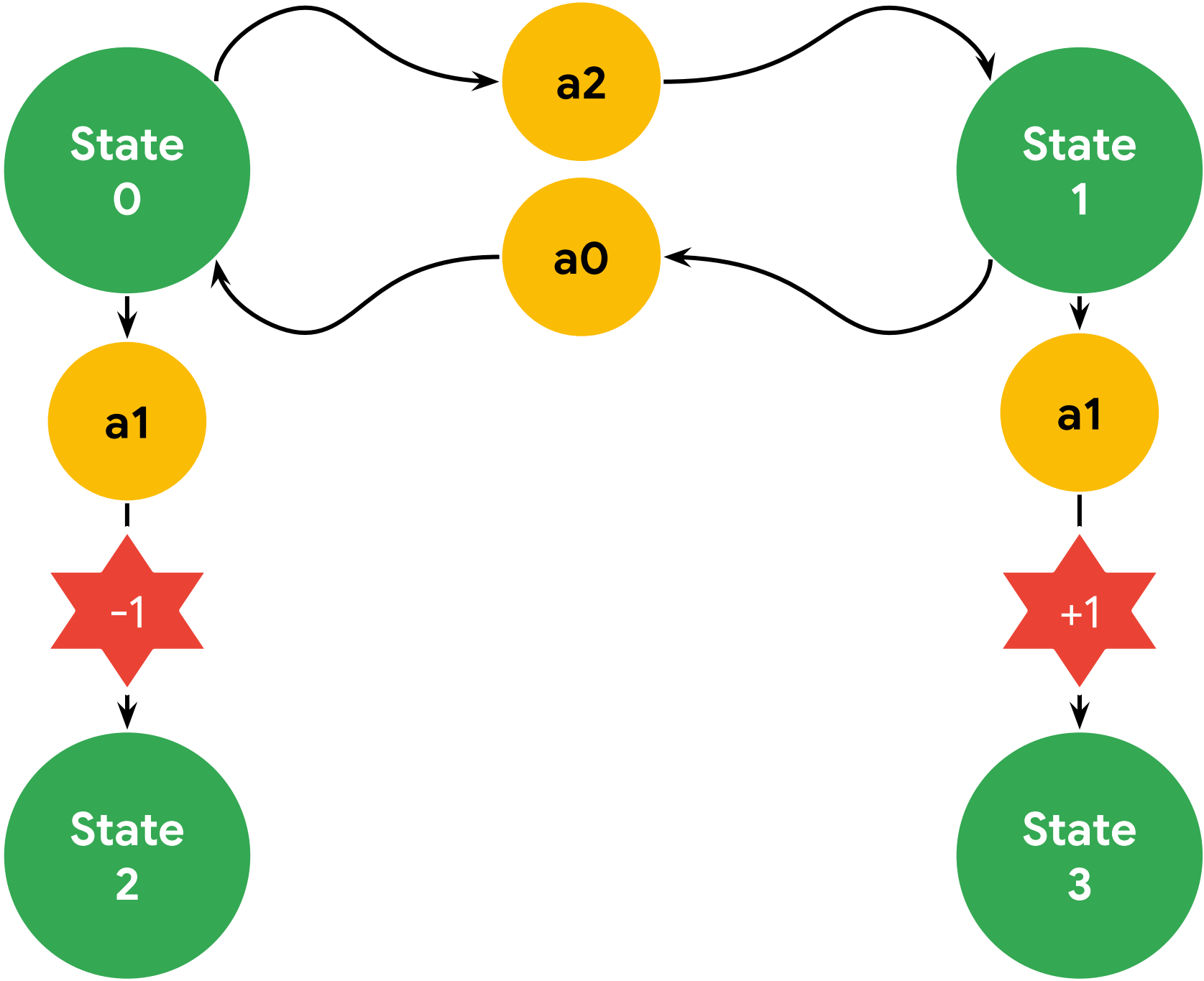
Policy Iteration

State Map	
0	1
2	3

Current Value	
0	0
0	0

Policy Map	
1	1
-	-

Prime Value	
0	0
0	0



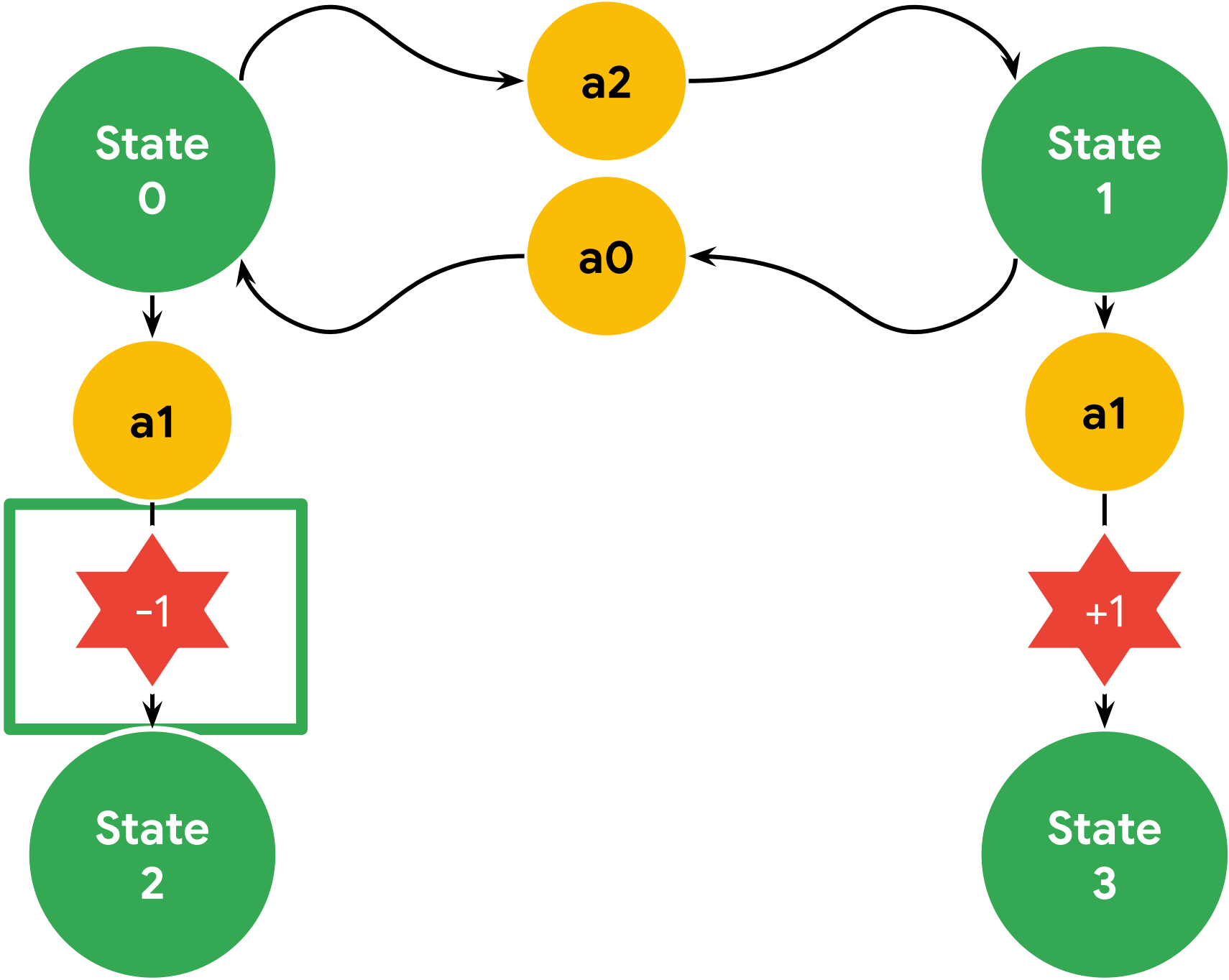
Policy Iteration

State Map	
<u>0</u>	1
2	3

Current Value	
0	0
<u>0</u>	0

Policy Map	
1	1
-	-

Prime Value	
<u>-1</u>	1
0	0



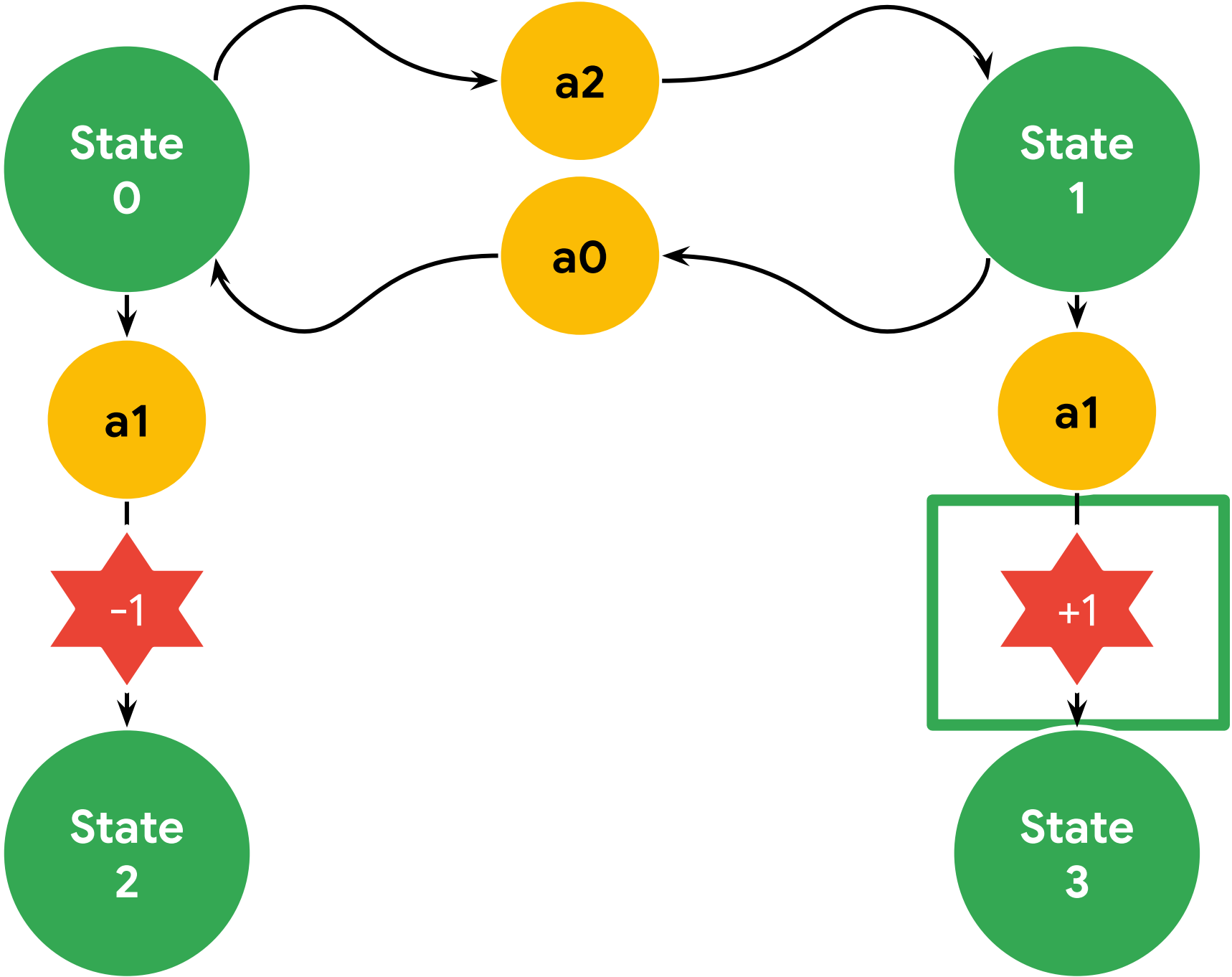
Policy Iteration

State Map	
0	1
2	3

Current Value	
0	0
0	<u>0</u>

Policy Map	
1	1
-	-

Prime Value	
-1	<u>1</u>
0	0



Policy Iteration

State Map	
0	1
2	3

Current Value	
0	0
0	0

Policy Map	
1	1
-	-

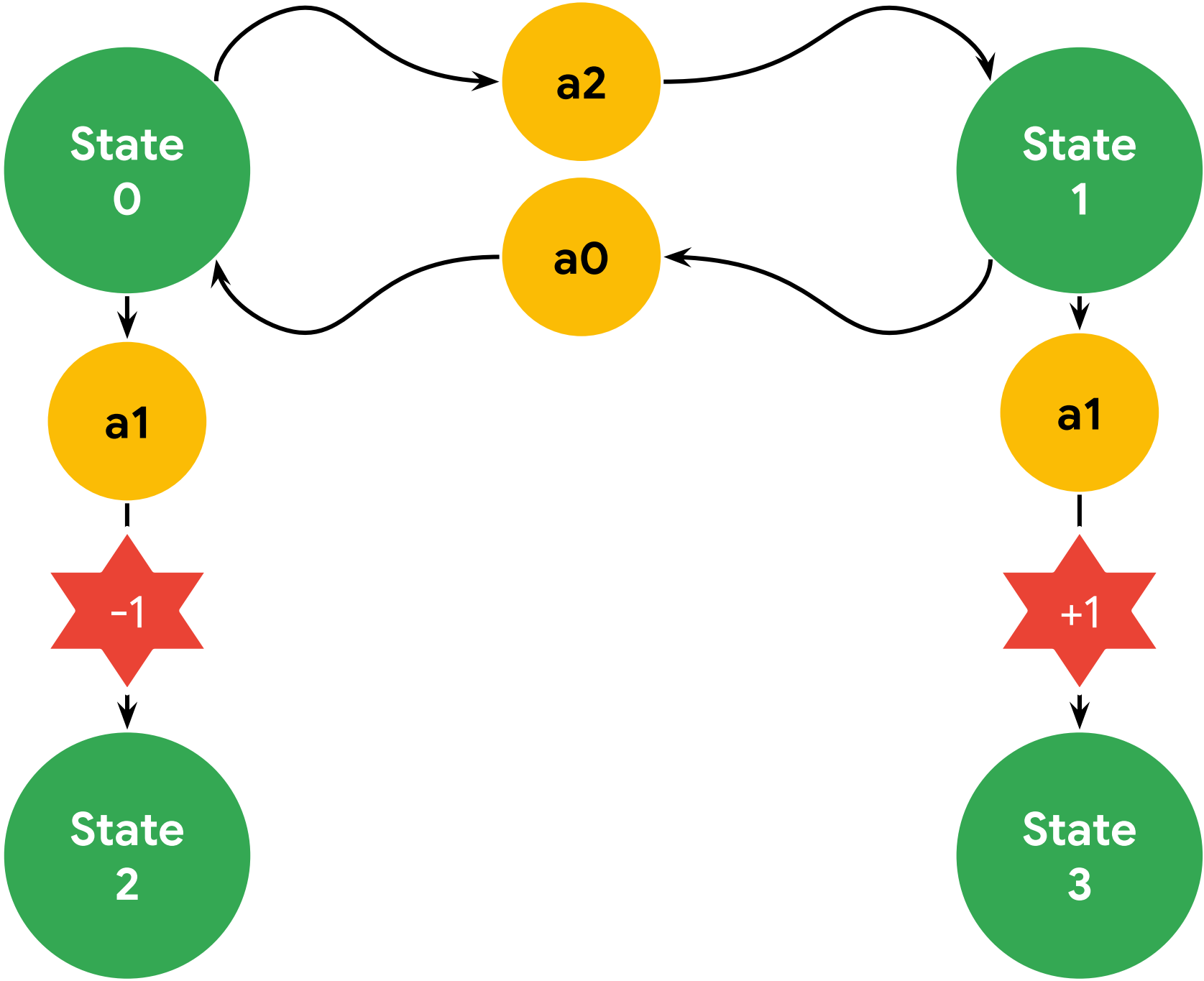
Prime Value	
-1	1
0	0

$$\gamma = .9$$

Policy Iteration

State Map	
0	1
2	3

Current Value	
-.9	.9
0	0



Policy Map	
1	1
-	-

Prime Value	
0	0
0	0

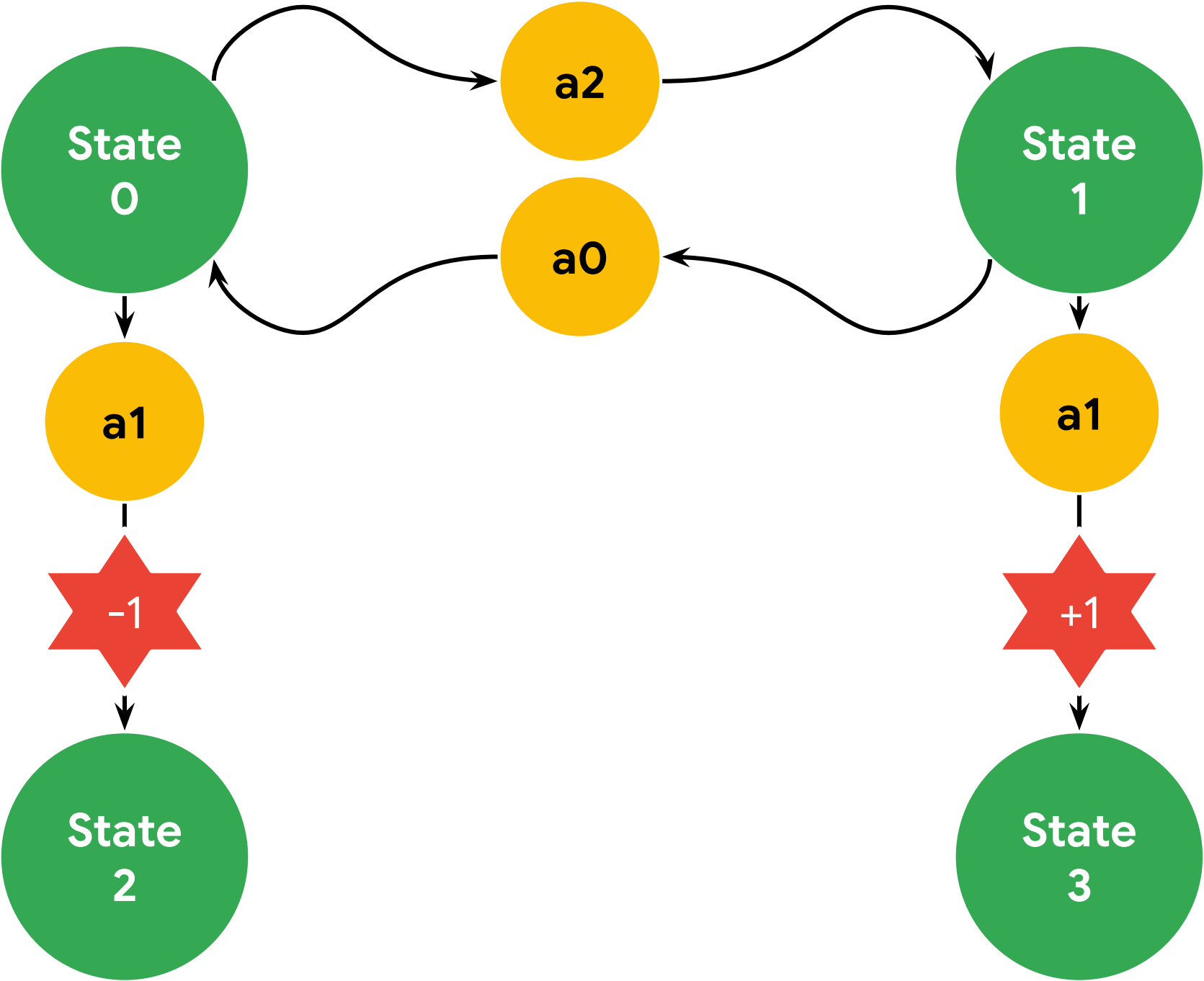
Policy Iteration

State Map	
<u>0</u>	1
2	3

Current Value	
-.9	<u>.9</u>
<u>0</u>	0

Policy Map	
2	1
-	-

Prime Value	
-1	1
0	0



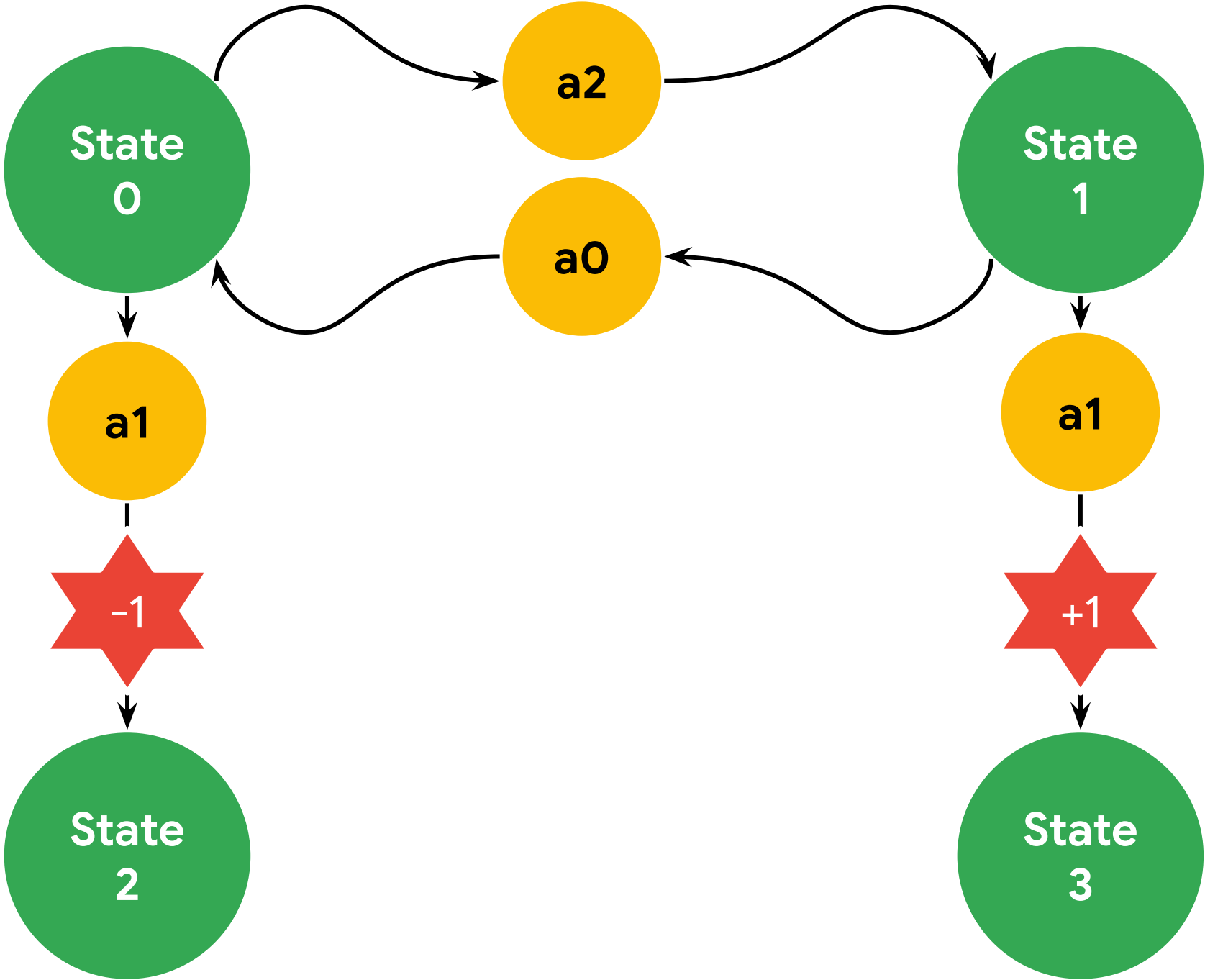
Policy Iteration

State Map	
0	1
2	3

Current Value	
<u>-.9</u>	.9
0	<u>0</u>

Policy Map	
2	1
-	-

Prime Value	
-1	1
0	0



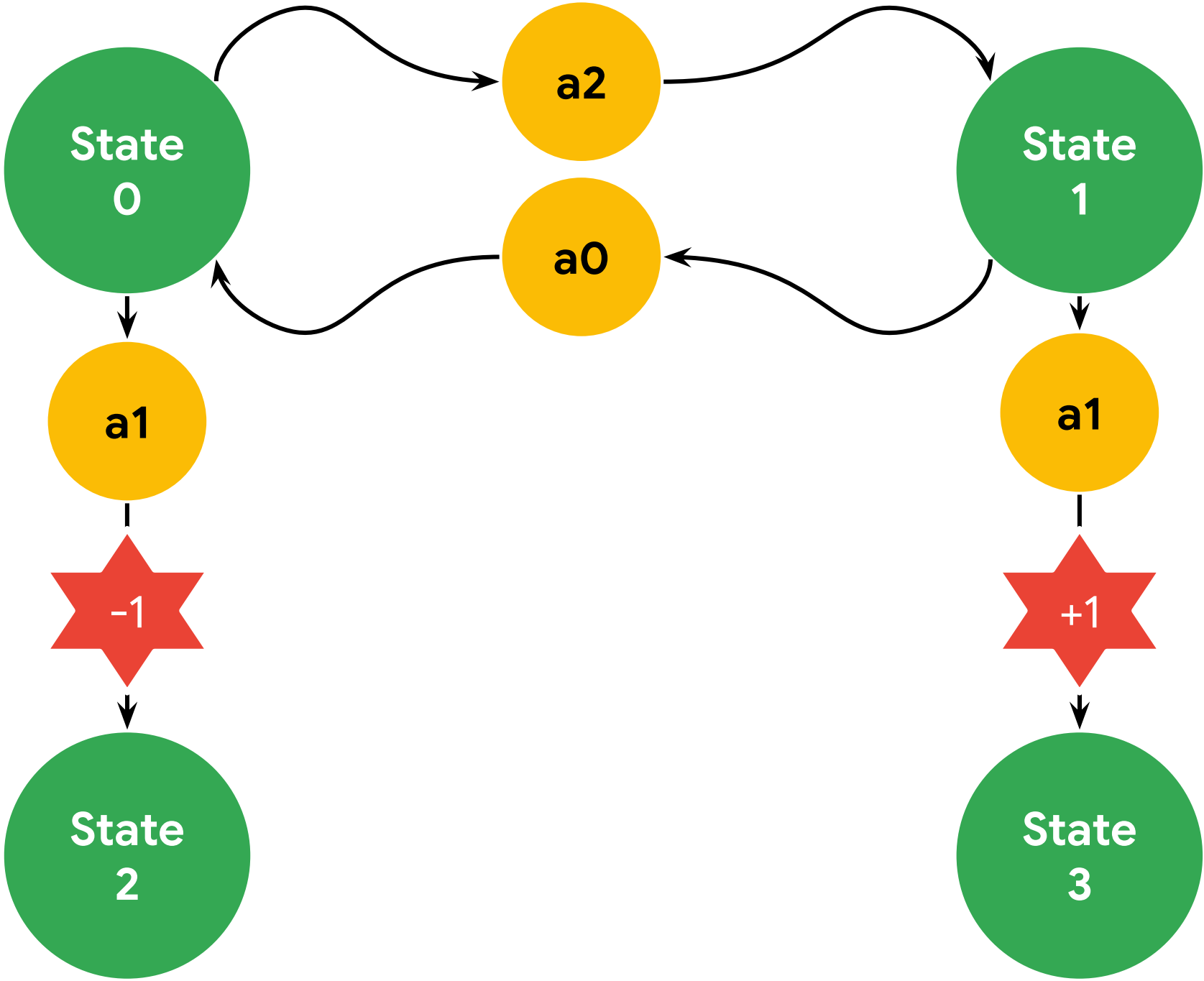
Policy Iteration (Iteration 2)

State Map	
0	1
2	3

Current Value	
<u>.81</u>	.9
0	0

Policy Map	
2	1
-	-

Prime Value	
0	0
0	0



Modified Policy Iteration Code

```
def iterate_policy(current_values, current_policies):  
    """Finds the future state values for an array of current states.  
  
    Args:  
        current_values (int array): the value of current states.  
        current_policies (int array): a list where each cell is the recommended  
            action for the state matching its index.  
  
    Returns:  
        next_values (int array): The value of states based on future states.  
        next_policies (int array): The recommended action to take in a state.  
    """  
    next_values = find_future_values(current_values, current_policies)  
    next_policies = find_best_policy(next_values)  
    return next_values, next_policies
```

Modified Policy Iteration Code

```
def find_future_values(current_values, current_policies):  
    """Finds the next set of future values based on the current policy."""  
    next_values = []  
  
    for state in STATE_RANGE:  
        current_policy = current_policies[state]  
        state_x, state_y = get_state_coordinates(state)  
  
        # If the cell has something other than 0, it's a terminal state.  
        value = LAKE[state_y, state_x]  
        if not value:  
            value = get_neighbor_value(  
                state_x, state_y, current_values, current_policy)  
        next_values.append(value)  
    return np.array(next_values).reshape((LAKE_HEIGHT, LAKE_WIDTH))
```

Modified Policy Iteration Code

```
def find_best_policy(next_values):  
    """Finds the best policy given a value mapping."""  
    next_policies = []  
    for state in STATE_RANGE:  
        state_x, state_y = get_state_coordinates(state)  
  
        # No policy or best value yet  
        max_value = -np.inf  
        best_policy = -1  
  
        if not LAKE[state_y, state_x]:  
            for policy in ACTION_RANGE:  
                neighbor_value = get_neighbor_value(  
                    state_x, state_y, next_values, policy)  
                if neighbor_value > max_value:  
                    max_value = neighbor_value  
                    best_policy = policy  
  
        next_policies.append(best_policy)  
    return next_policies
```

Modified Policy Iteration Complexity

$$O(\underbrace{s^2 s'}_{\text{blue}} + \underbrace{s^2 a s'}_{\text{yellow}})$$

Still need to look
at weighted sum
of future states
to calculate value

Finding the
new policy is
pretty much
the same as
Value Iteration

Value Iteration vs Policy Iteration

Value Iteration				Iteration 7				Optimal Policy			
Lake				.00	.00	.00	.00	0	3	3	3
0	0	0	0	.01	0	-.27	0	0	-	0	-
0	-1	0	-1	.03	.10	.10	0	3	1	0	-
0	0	0	-1	0	.25	.52	0	-	2	1	-
-1	0	0	1	Iteration 4				Optimal Policy			
Policy Iteration				0	0	0	0	0	3	3	3
				0	0	-.3	0	0	-	0	-
				0	.03	.03	0	3	1	0	-
				0	.14	.44	0	-	2	1	-

Value Iteration vs Policy Iteration

Property	Value Iteration	Policy Iteration
Mathematically precise	✓	x
Less iterations	x	✓
Less computation per iteration	✓	x
Convergence condition	Little change in value	No change in policy

Agenda

History Overview

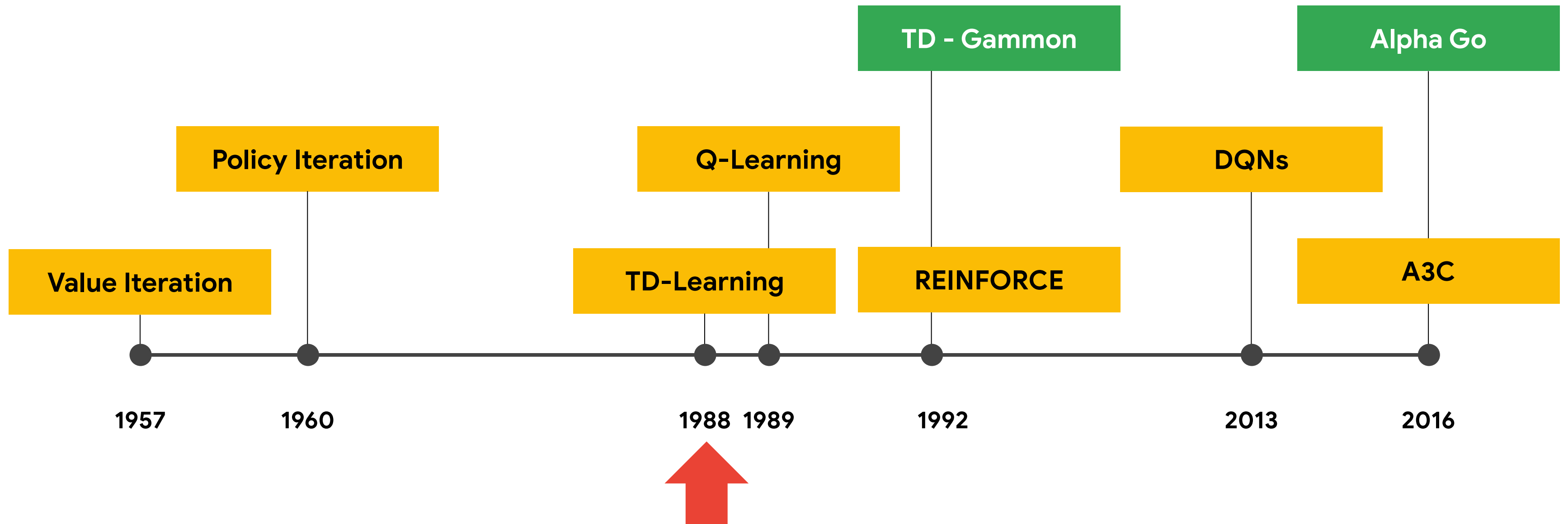
Value Iteration

Policy Iteration

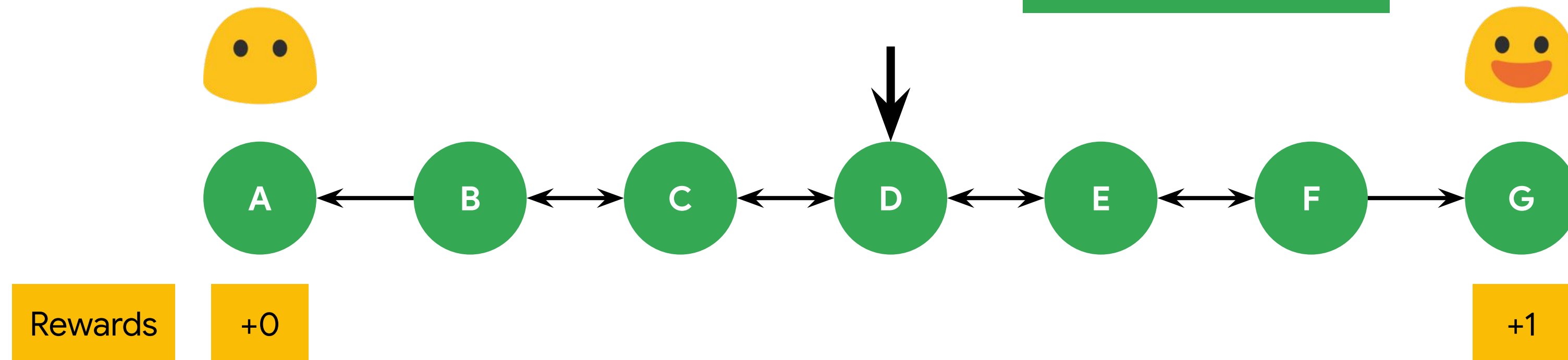
TD(Lambda)

Q-Learning


An RL Timeline




A Random Walk



A Random Walk

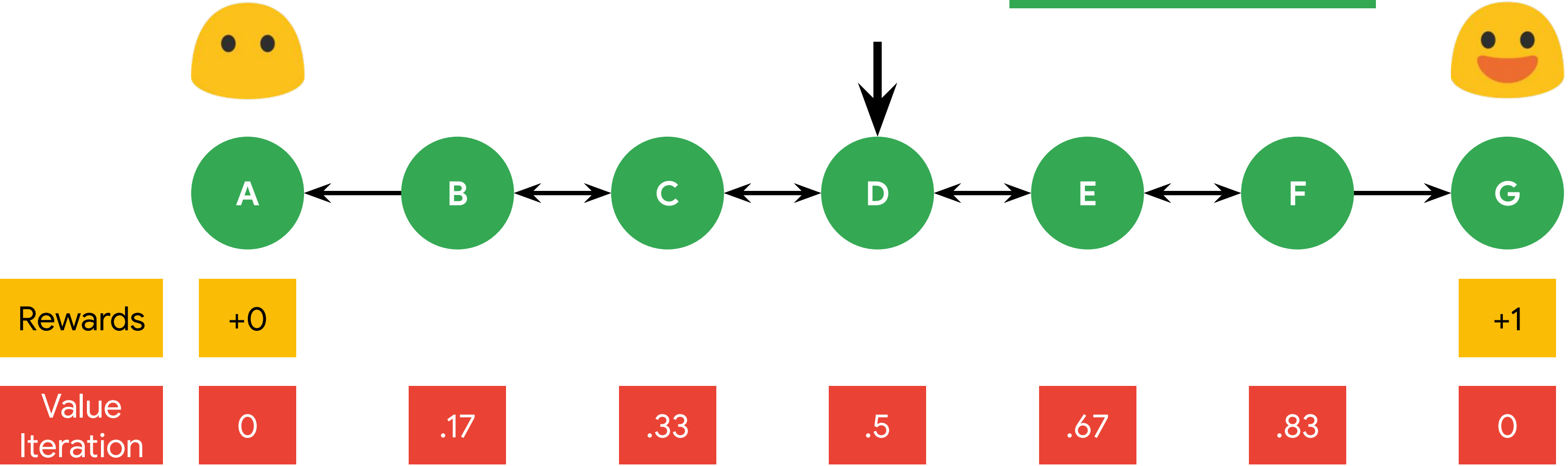


Tails
Left
50%



Heads
Right
50%

$\gamma = 1$



TD(0)

$$\underline{V(s)} = \underline{R(s,a)} + \underline{\gamma V(s')}$$


$$\underline{V(s_{t-1})} = \underline{V(s_{t-1})} + \underline{\alpha_t}(\underline{R(s_{t-1},a)} + \underline{\gamma V(s_t)} - \underline{V(s_{t-1})})$$

New 
Variable!




The Learning Rate

TD(0) Random Walk



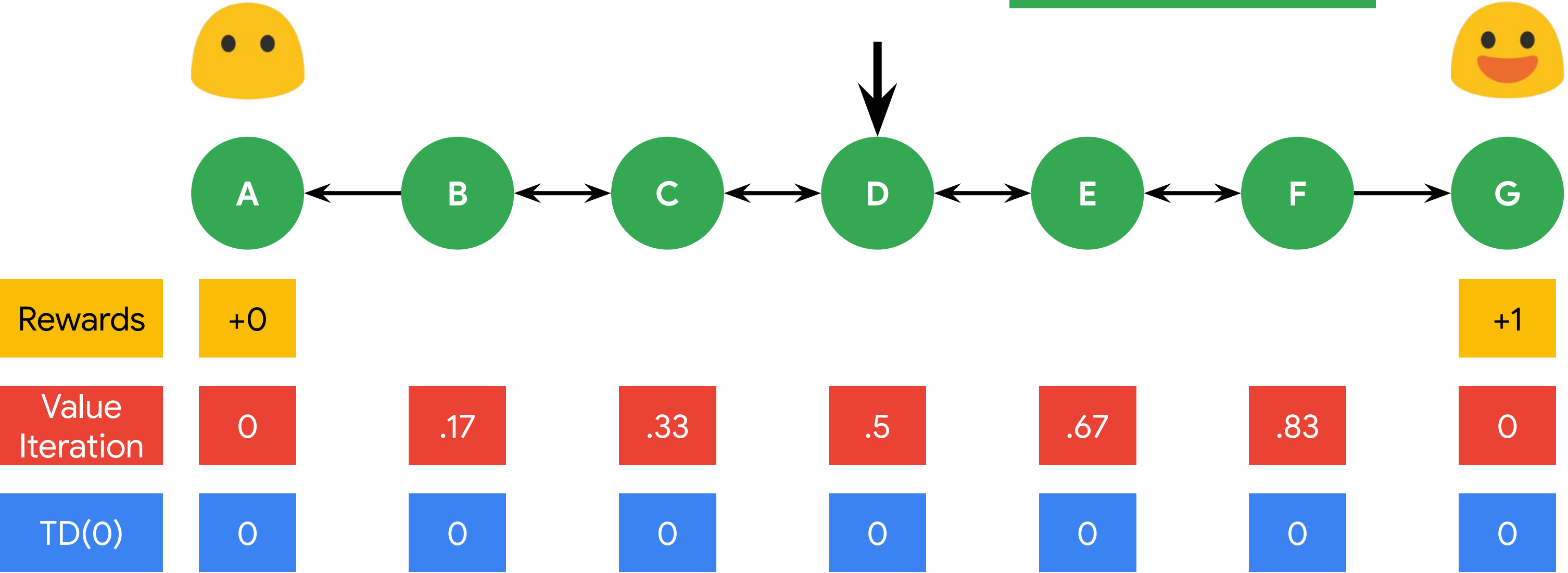
Tails
Left
50%



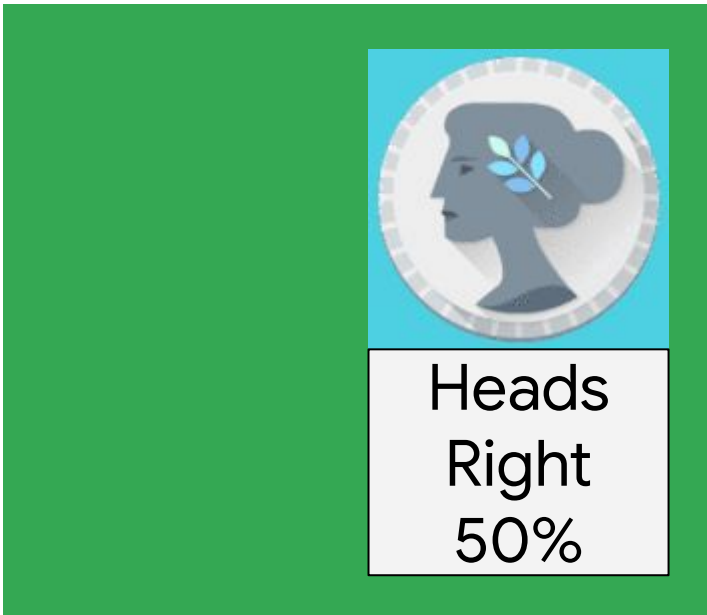
Heads
Right
50%

$\gamma = 1$

$\alpha = .5$

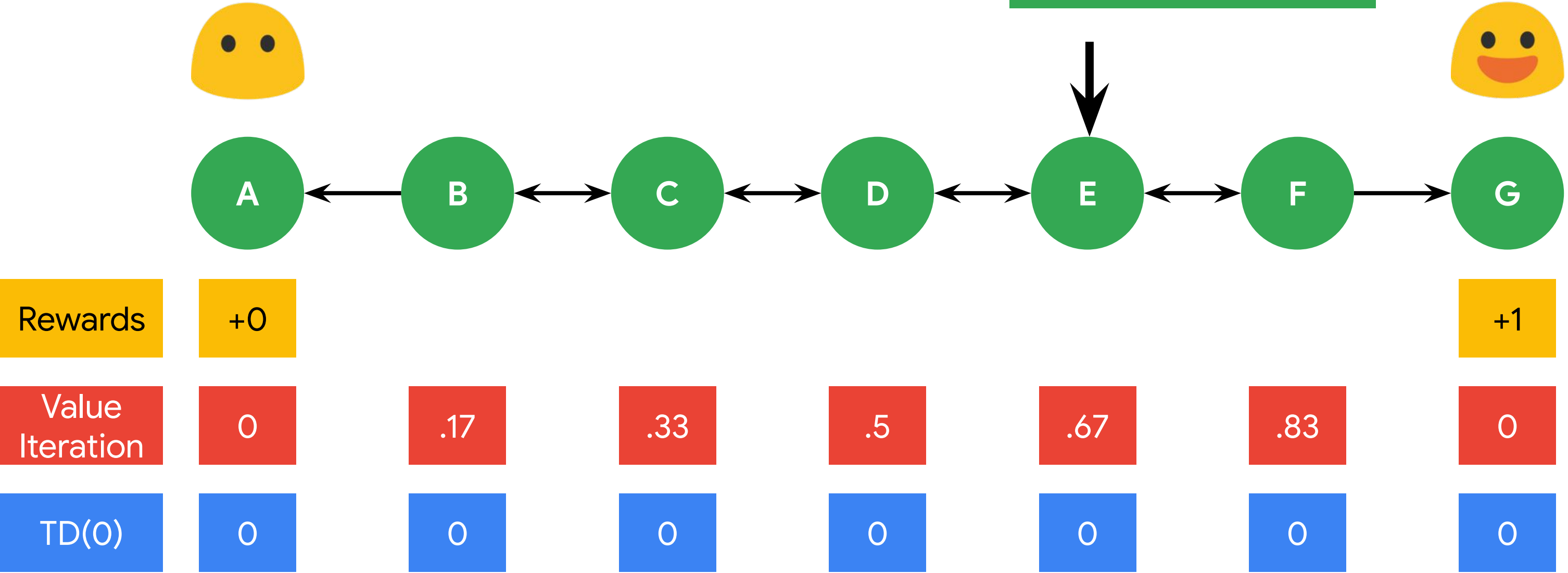


TD(0) Random Walk

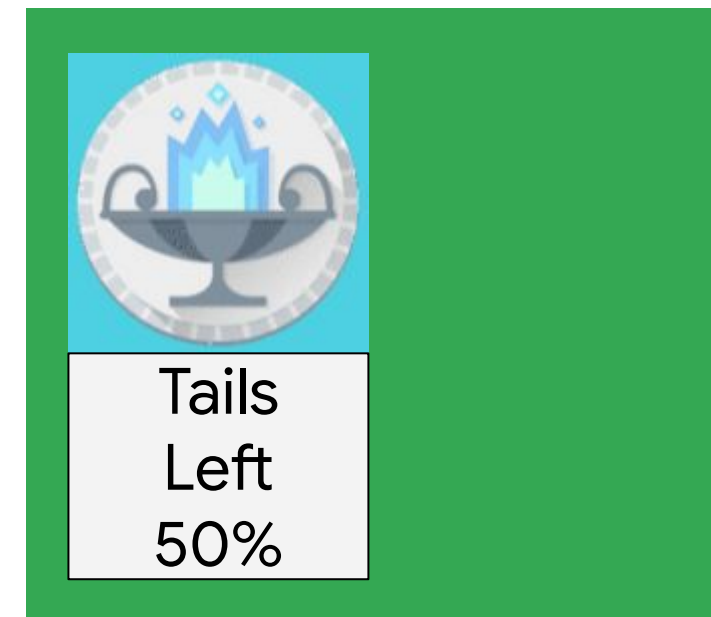


$\gamma = 1$

$\alpha = .5$

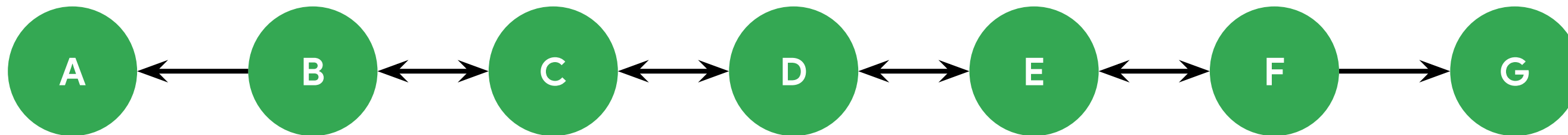


TD(0) Random Walk



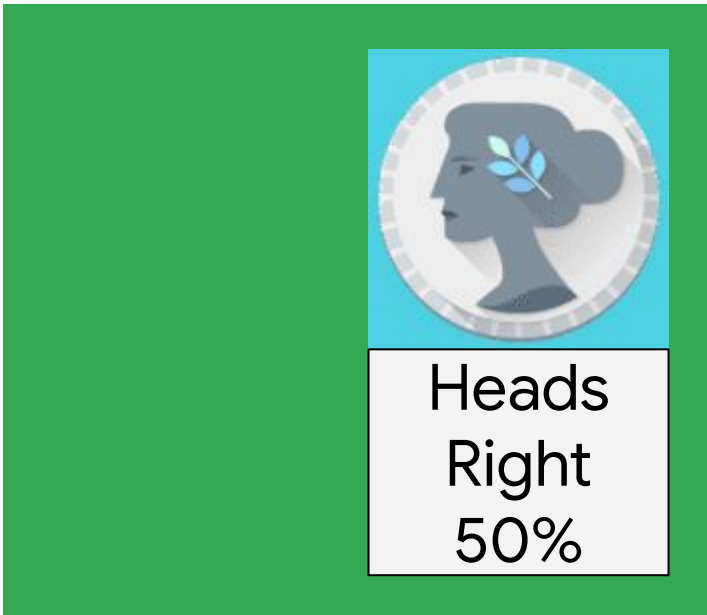
$$\gamma = 1$$

$$\alpha = .5$$



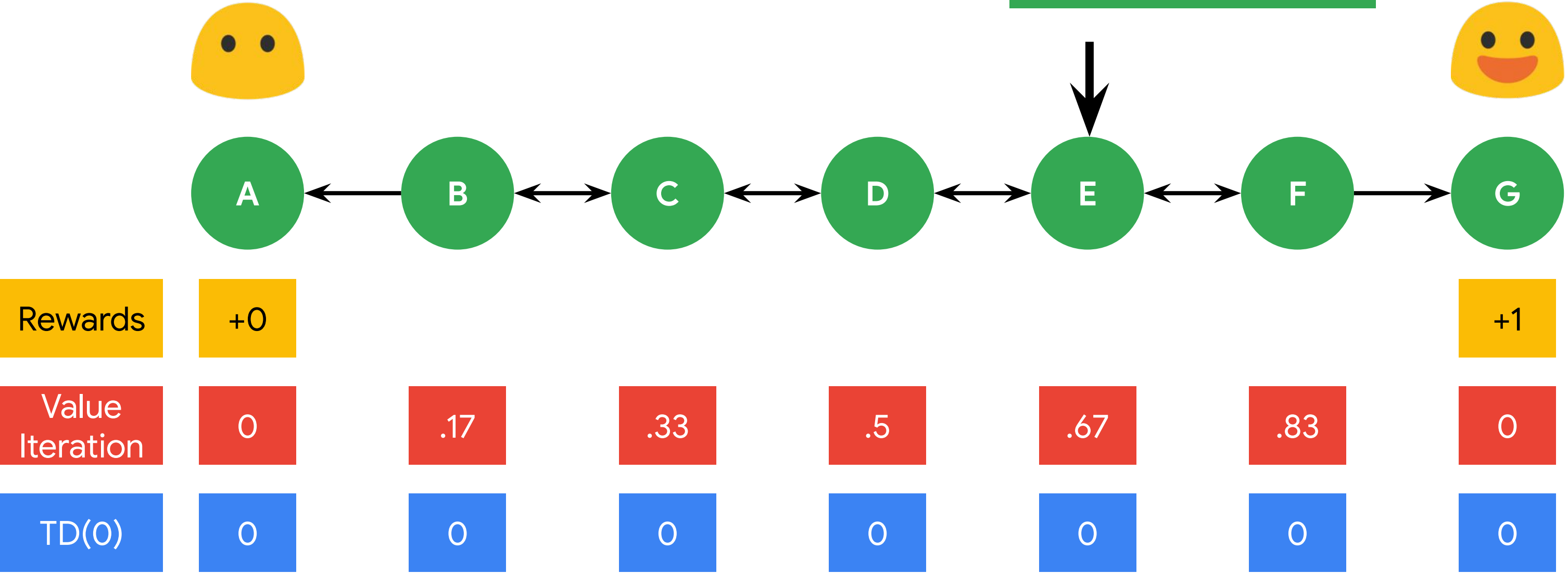
Rewards	+0									+1			
Value Iteration	0		.17		.33		.5		.67		.83		0
TD(0)	0		0		0		0		0		0		0

TD(0) Random Walk

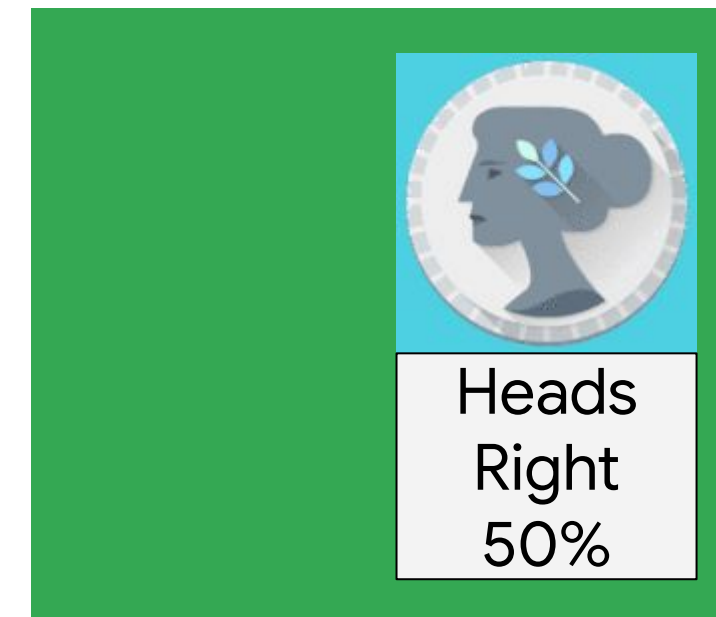


$\gamma = 1$

$\alpha = .5$

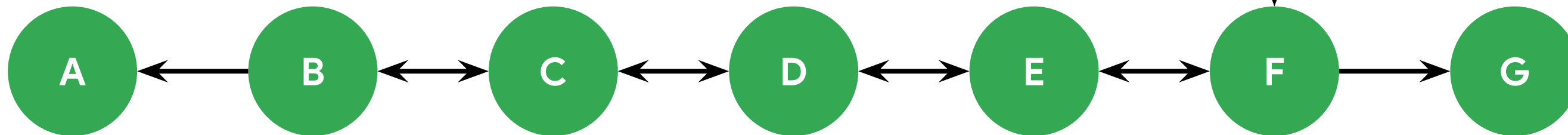


TD(0) Random Walk



$$\gamma = 1$$

$$\alpha = .5$$



Rewards

+0

+1

Value
Iteration

0

.17

.33

.5

.67

.83

0

TD(0)

0

0

0

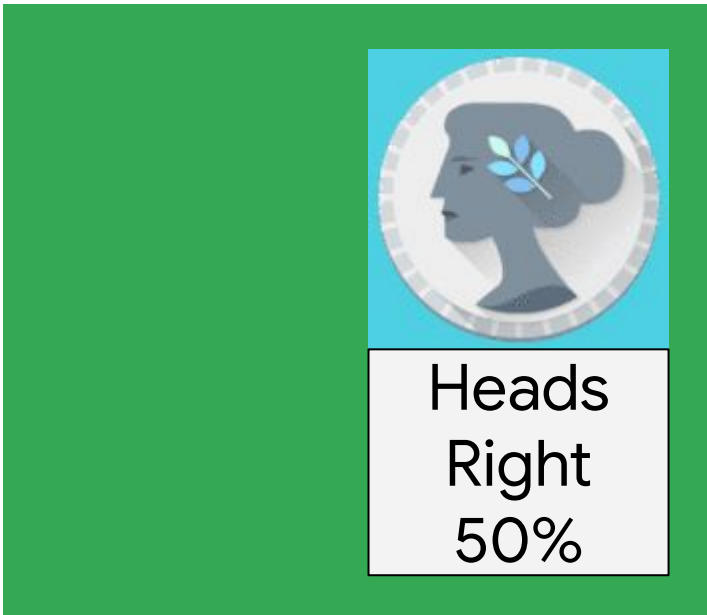
0


0

0

0

TD(0) Random Walk

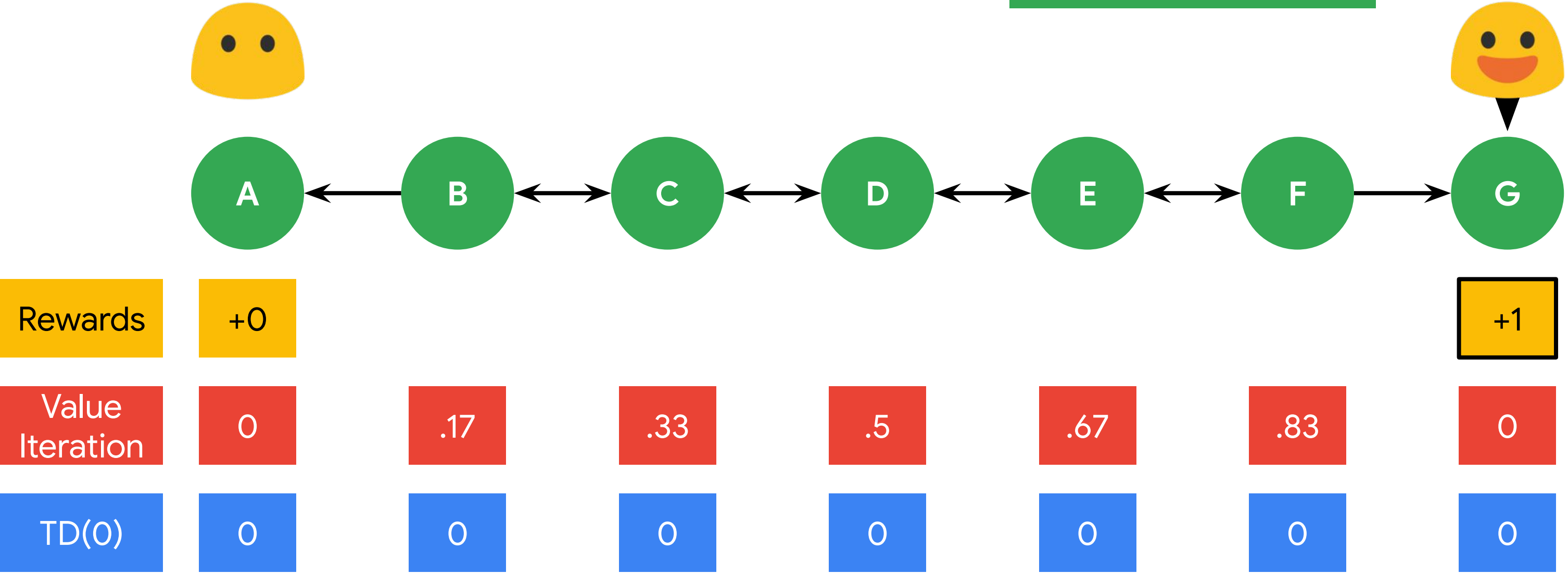




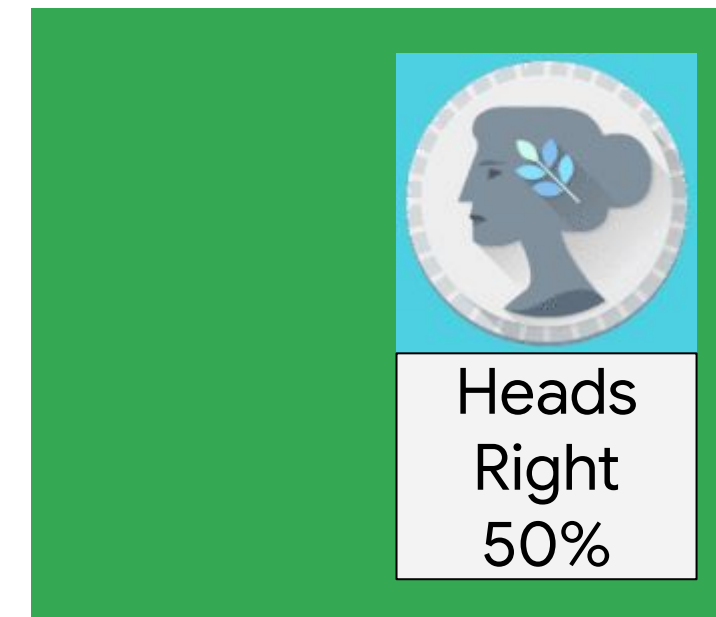
Heads
Right
50%

$\gamma = 1$

$\alpha = .5$

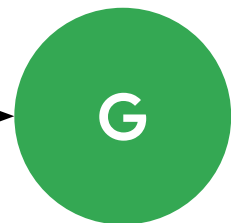
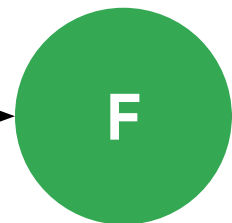
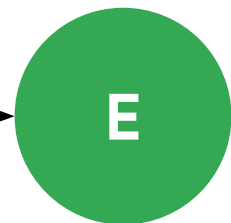
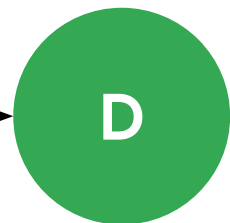
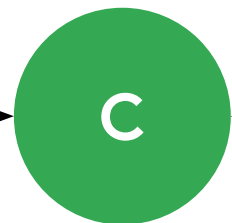
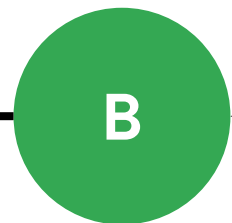


TD(0) Random Walk



$$\gamma = 1$$

$$\alpha = .5$$



$$V(s_{t-1}) = V(s_{t-1}) + \alpha_t (R(s_{t-1}, a) + \gamma V(s_t) - V(s_{t-1}))$$

$$= 0 + .5 (1 + 1 \cdot 0 - 0)$$

TD(0)

0

0

0


0

0


.5

0

TD(0) Random Walk



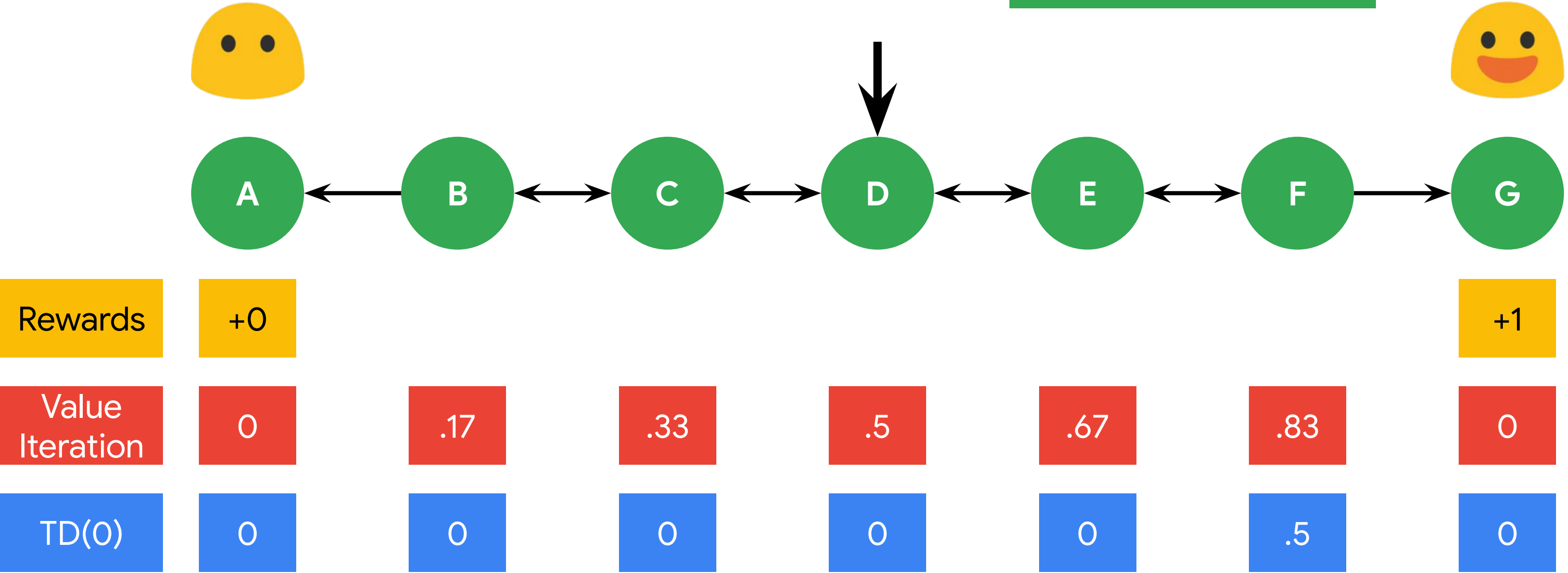
Tails
Left
50%



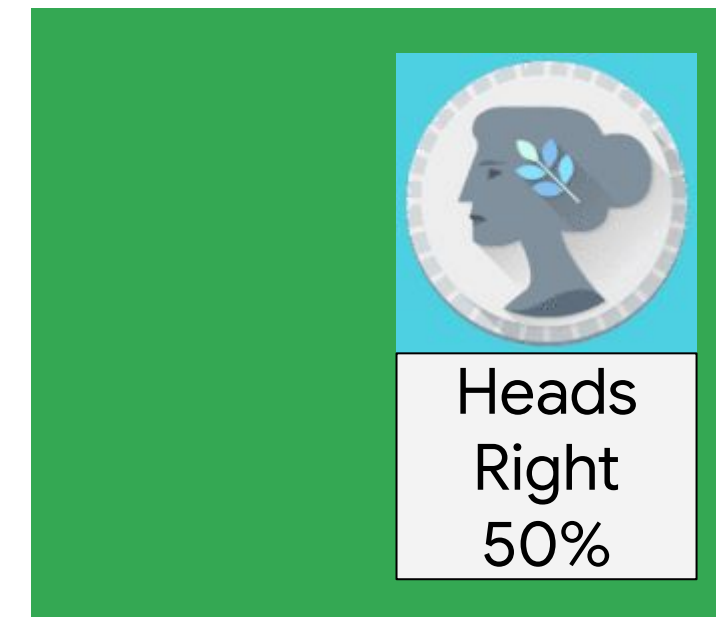
Heads
Right
50%

$\gamma = 1$

$\alpha = .5$

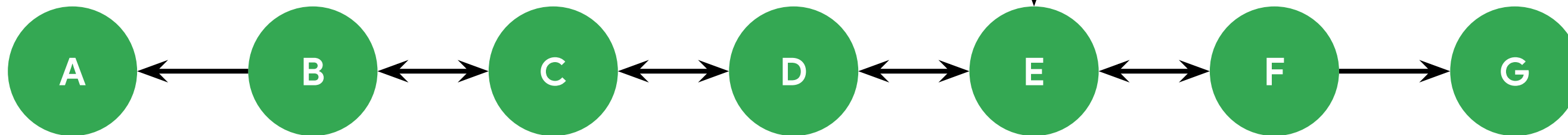


TD(0) Random Walk



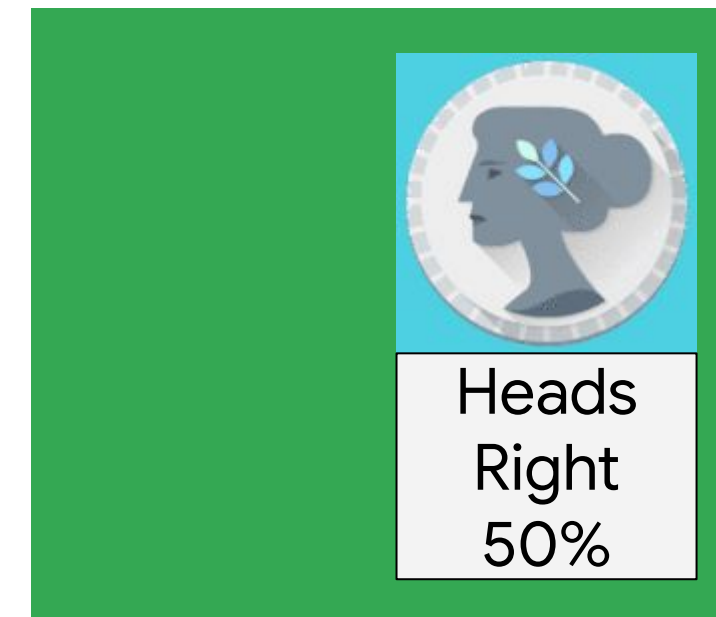
$$\gamma = 1$$

$$\alpha = .5$$



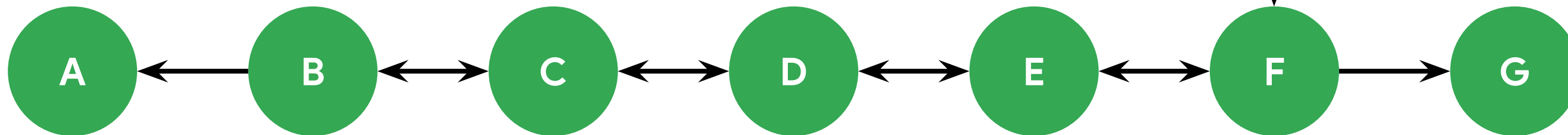
Rewards	+0								+1
Value Iteration	0	.17	.33	.5	.67	.83	0		
TD(0)	0	0	0	0	0	.5	0		

TD(0) Random Walk



$$\gamma = 1$$

$$\alpha = .5$$



Rewards

+0

+1

Value
Iteration

0

.17

.33

.5

.67

.83

0

TD(0)

0

0

0

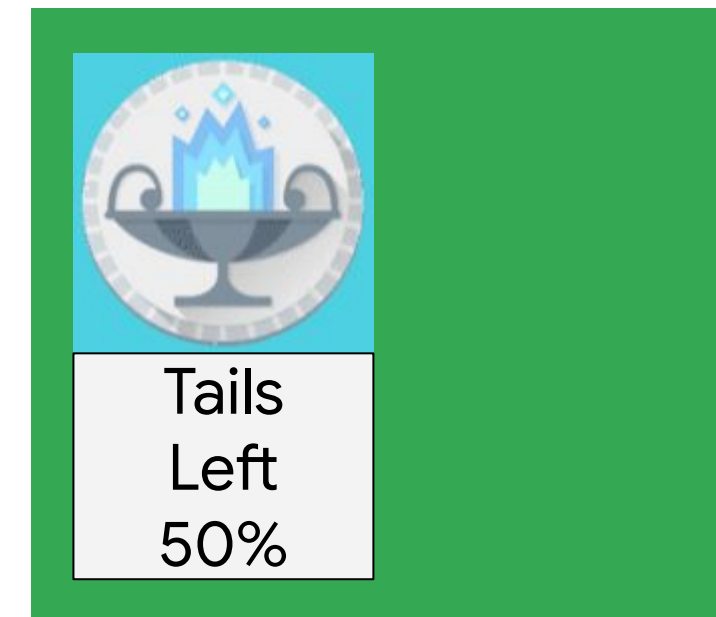
0

.25

.5

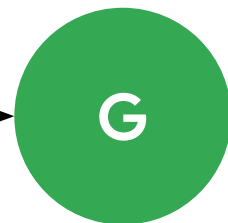
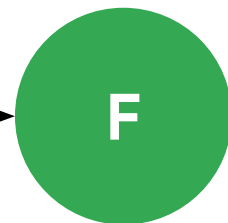
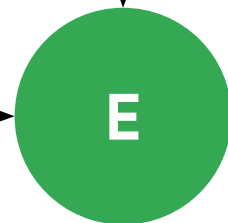
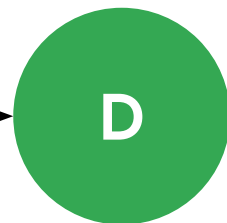
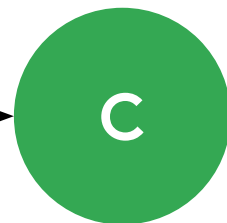
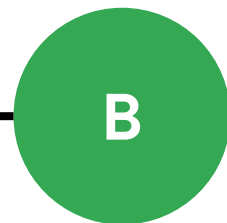
0

TD(0) Random Walk



$$\gamma = 1$$

$$\alpha = .5$$



$$V(s_{t-1}) = V(s_{t-1}) + \alpha_t (R(s_{t-1}, a) + \gamma V(s_t) - V(s_{t-1}))$$

$$= .5 + .5 (0 + 1 \cdot .25 - .5)$$

TD(0)

0

0

0

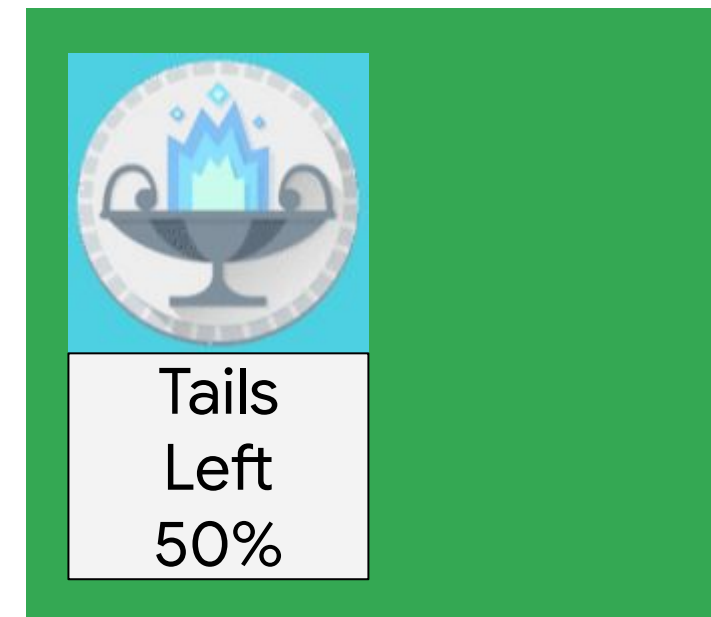
0

.25

.375

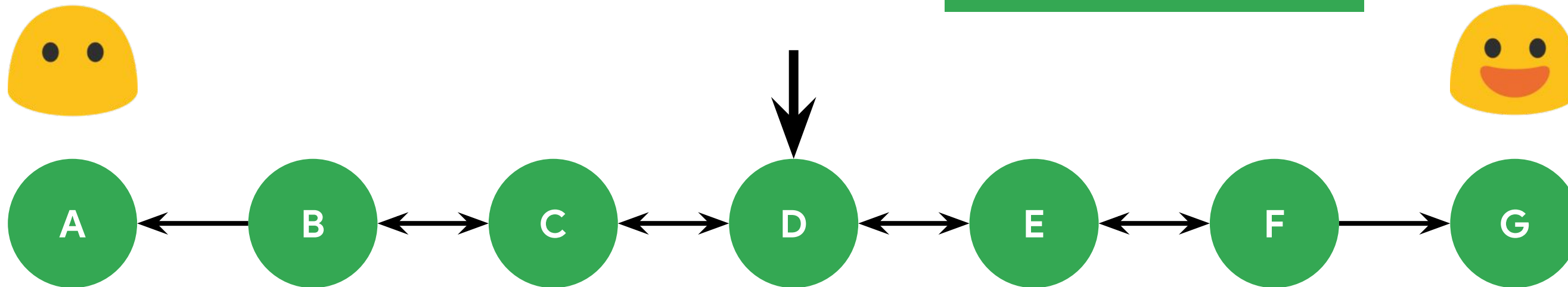
0

TD(0) Random Walk



$$\gamma = 1$$

$$\alpha = .5$$



Rewards	+0						+1
Value Iteration	0	.17	.33	.5	.67	.83	0
TD(0)	0	0	0	0	.125	.375	0

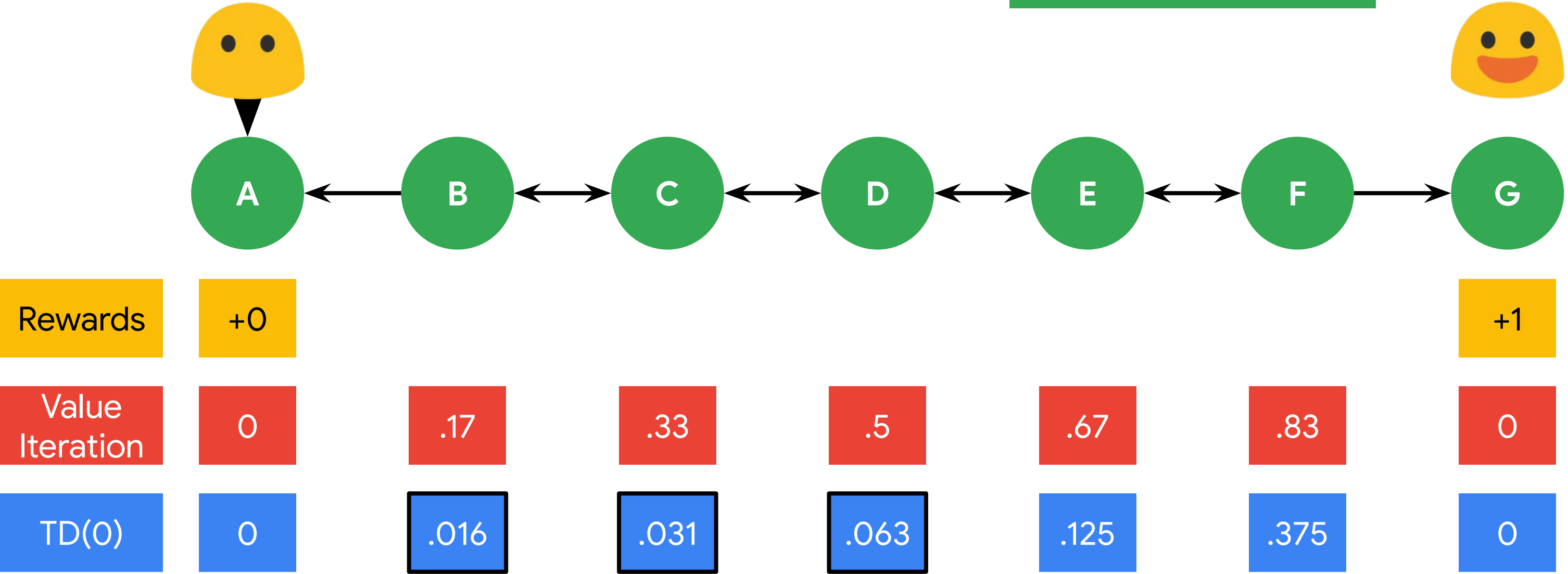
TD(0) Random Walk



Tails
Left
50%

$\gamma = 1$

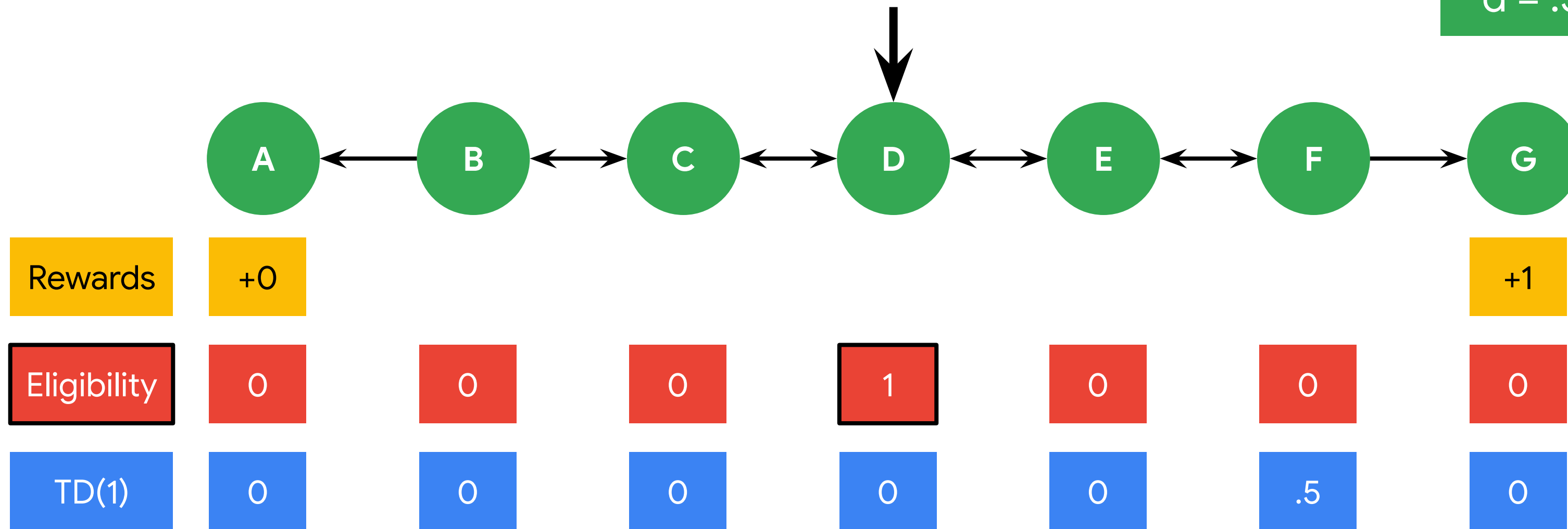
$\alpha = .5$



TD(1) Random Walk

$$\gamma = .9$$

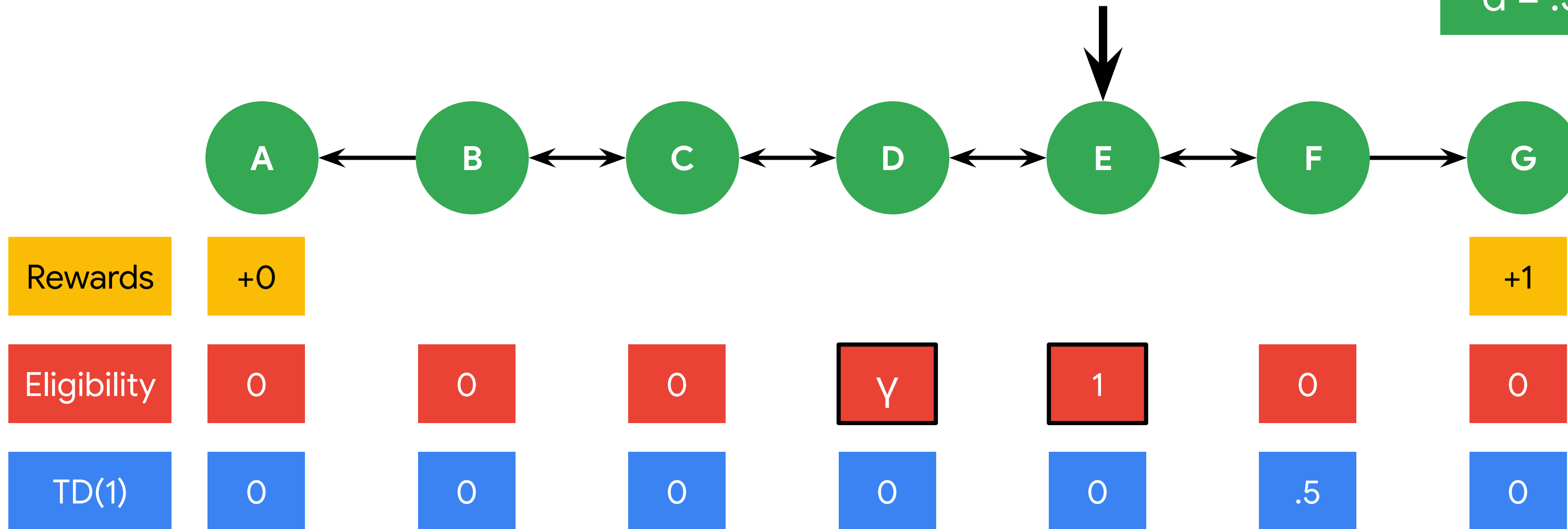
$$\alpha = .5$$



TD(1) Random Walk

$$\gamma = .9$$

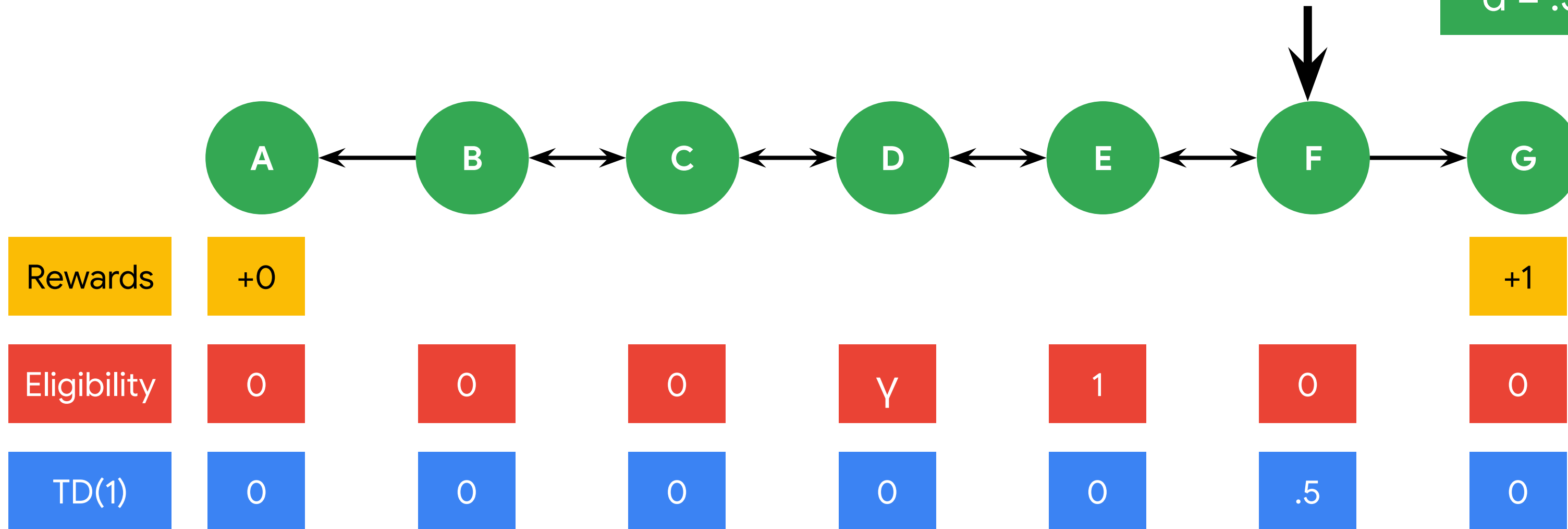
$$\alpha = .5$$



TD(1) Random Walk

$$\gamma = .9$$

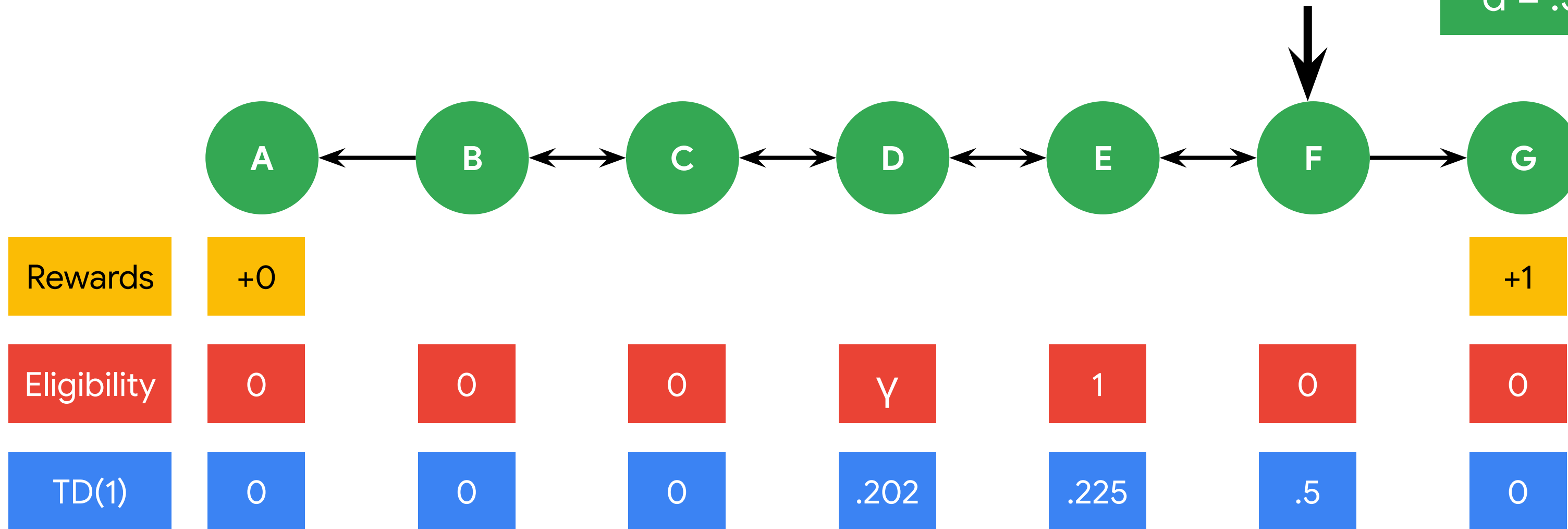
$$\alpha = .5$$



TD(1) Random Walk

$$\gamma = .9$$

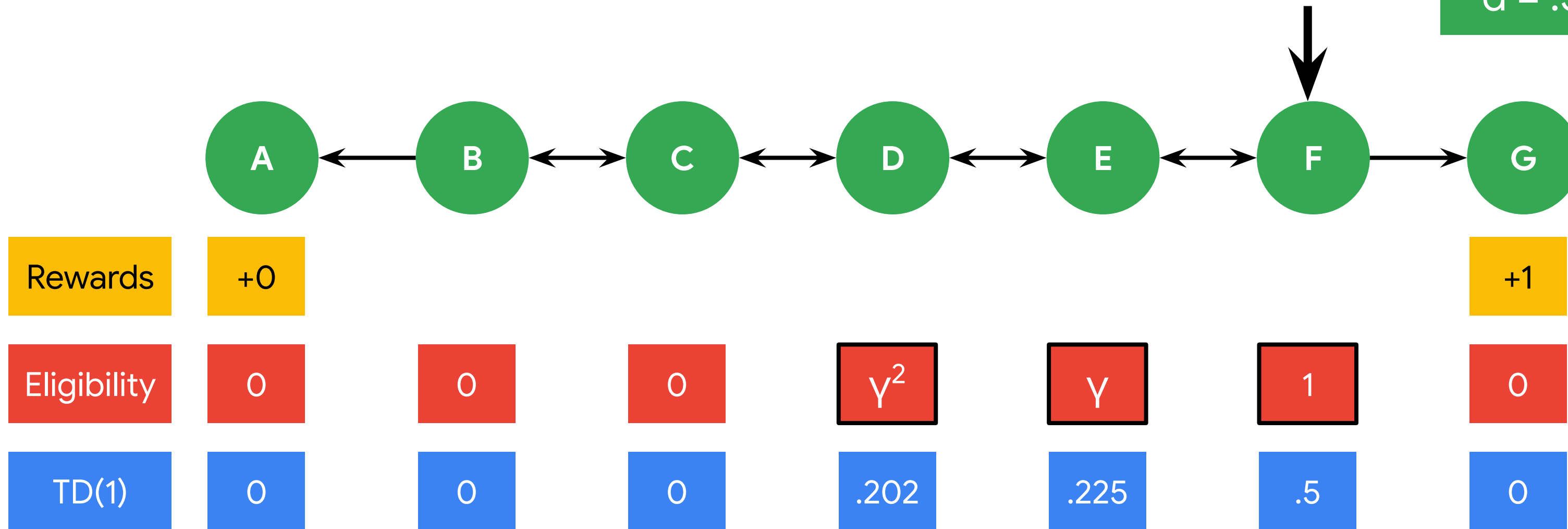
$$\alpha = .5$$



TD(1) Random Walk

$$\gamma = .9$$

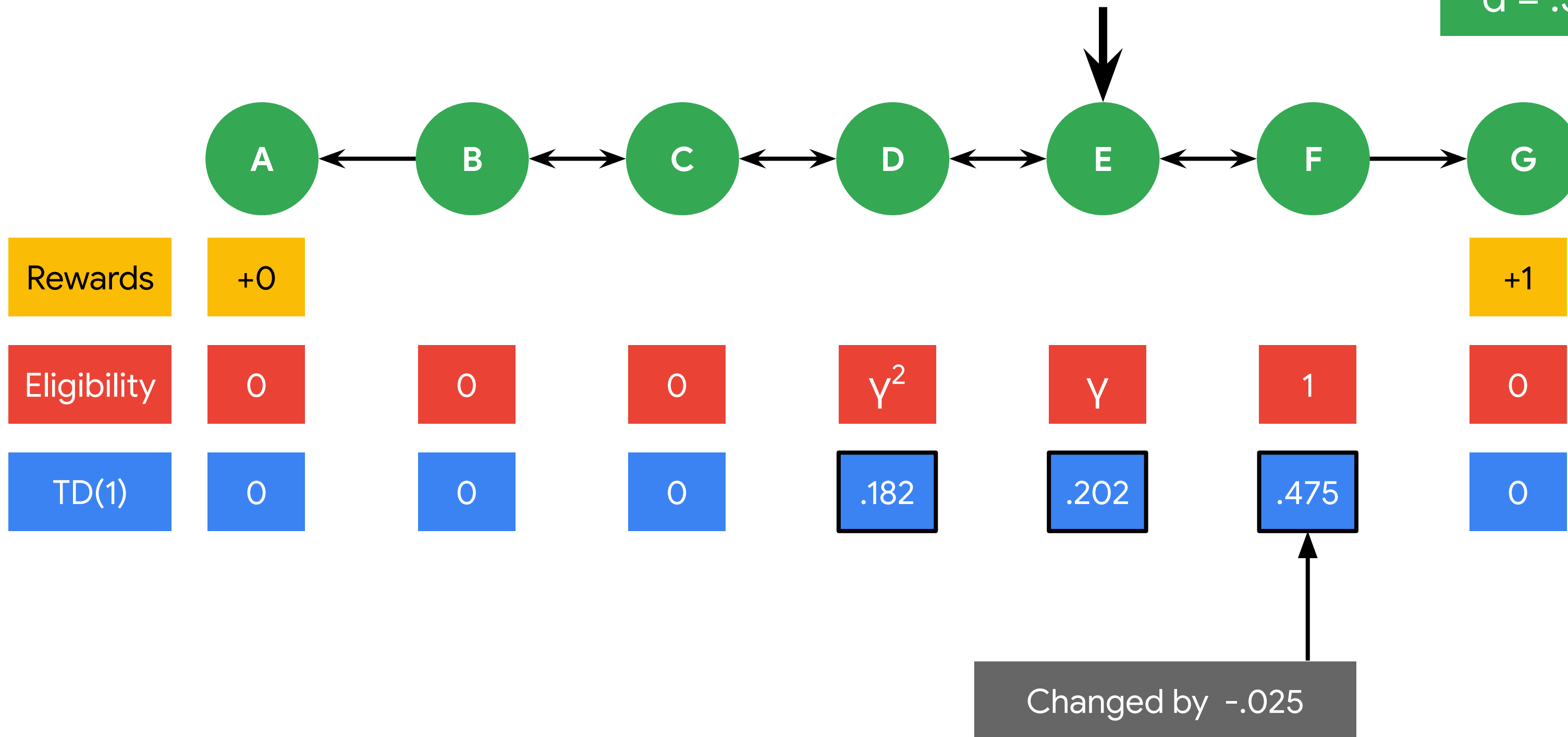
$$\alpha = .5$$



TD(1) Random Walk

$$\gamma = .9$$

$$\alpha = .5$$

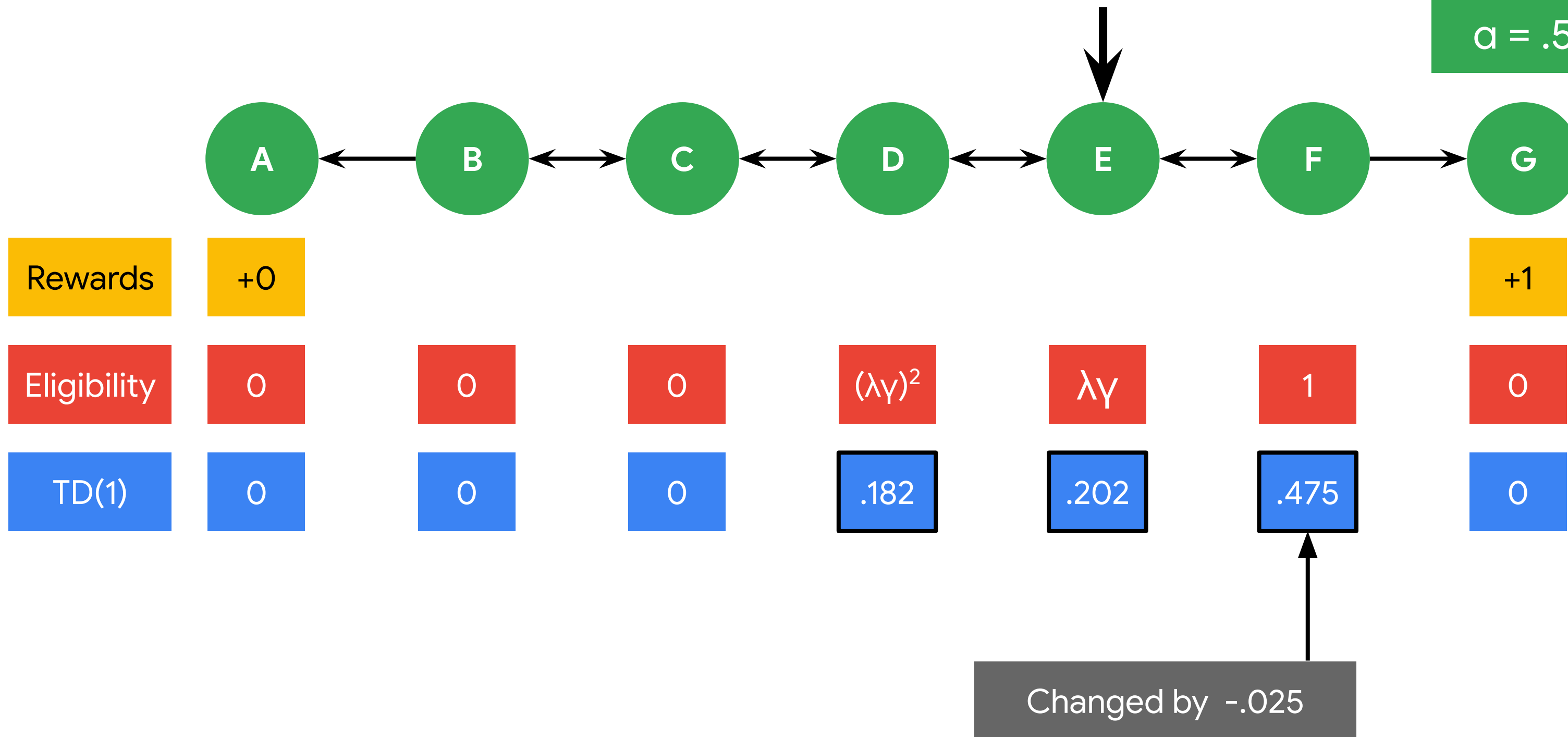


TD(λ) Random Walk

$$\lambda = 1$$

$$\gamma = .9$$

$$\alpha = .5$$



Agenda

History Overview

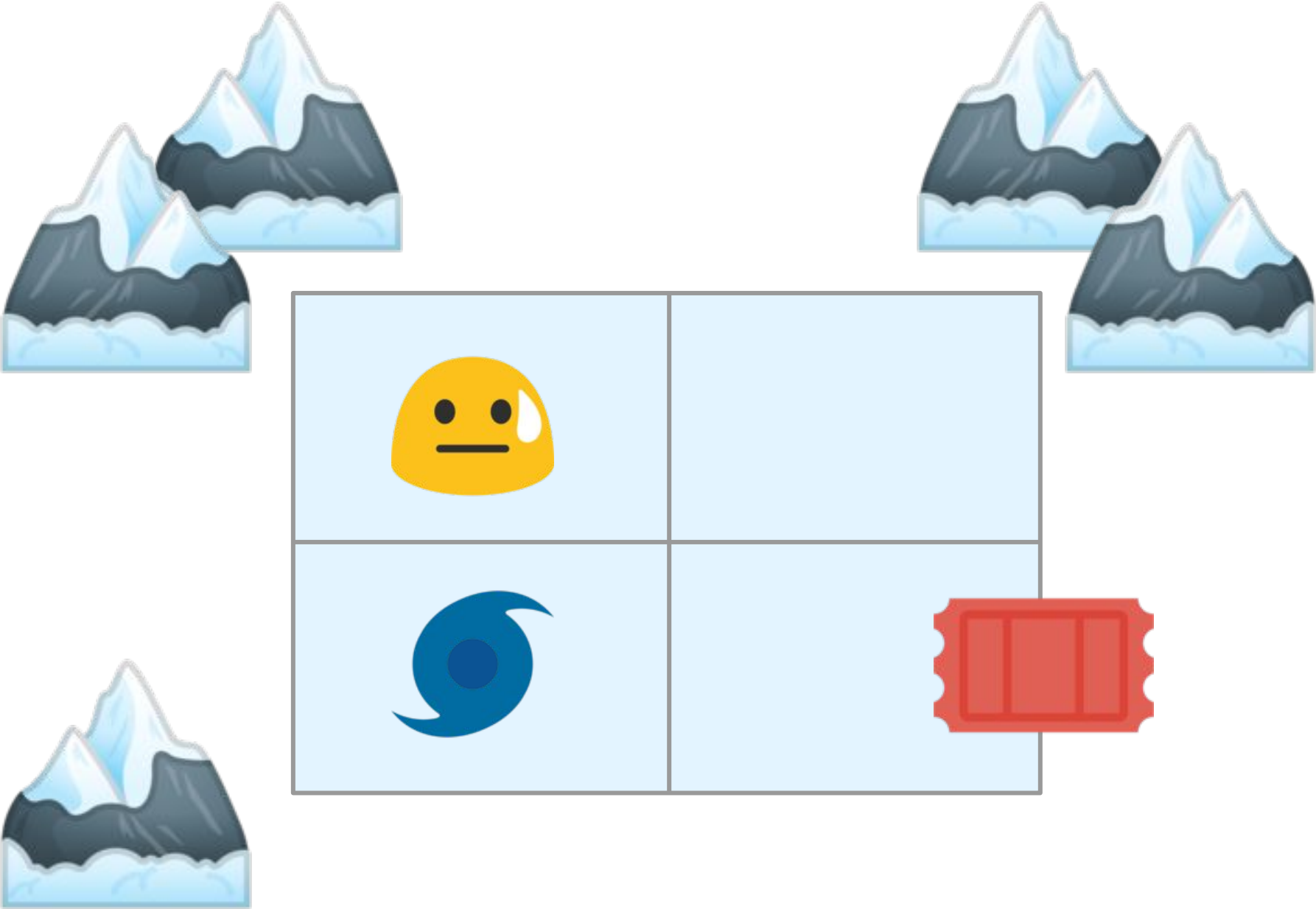
Value Iteration

Policy Iteration

TD(Lambda)

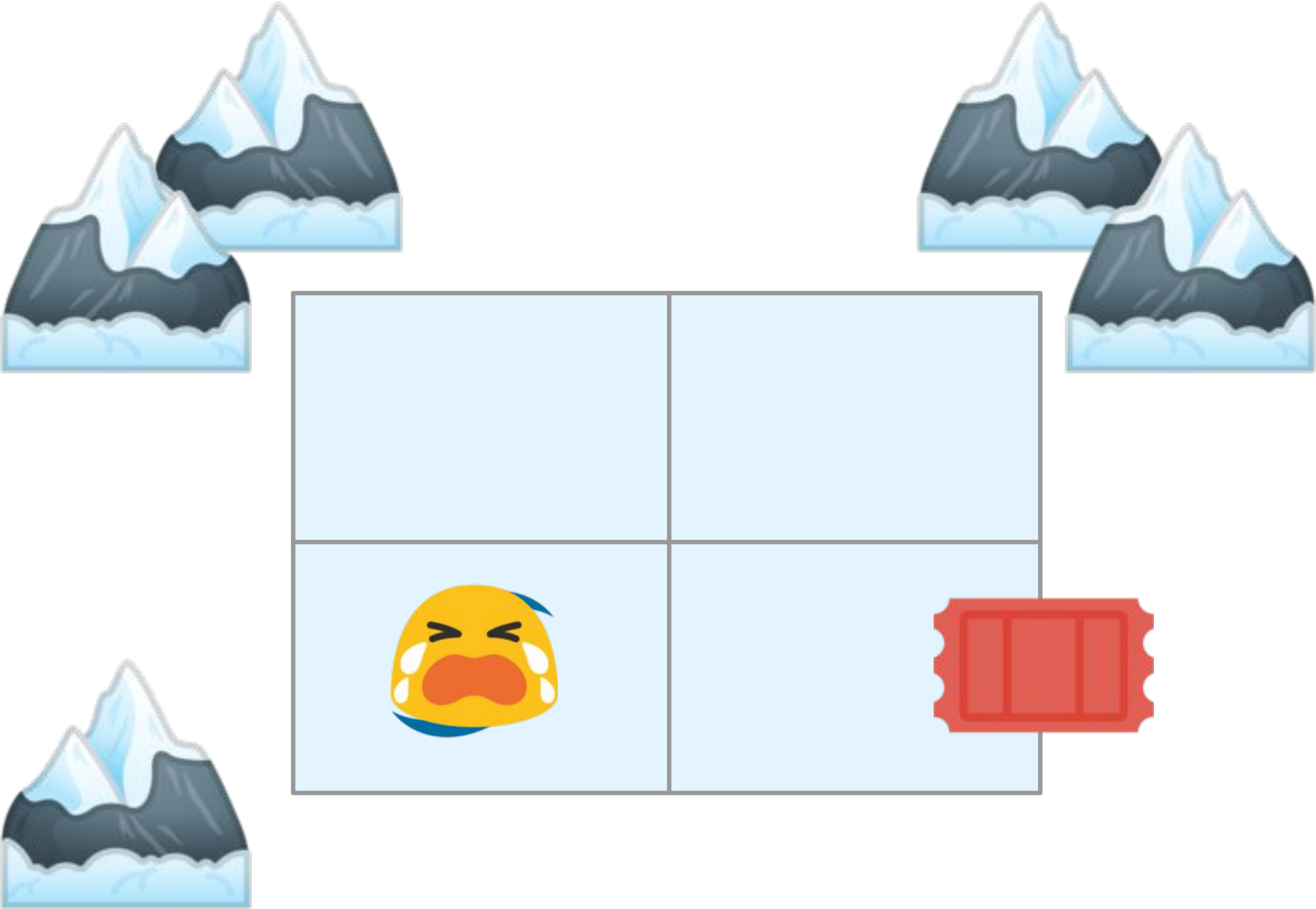
Q-Learning

The Q Table



Q - table				
	Left	Down	Right	Up
0	0	0	0	0
1	0	0	0	0
2	0	0	0	0
3	0	0	0	0

The Q Table

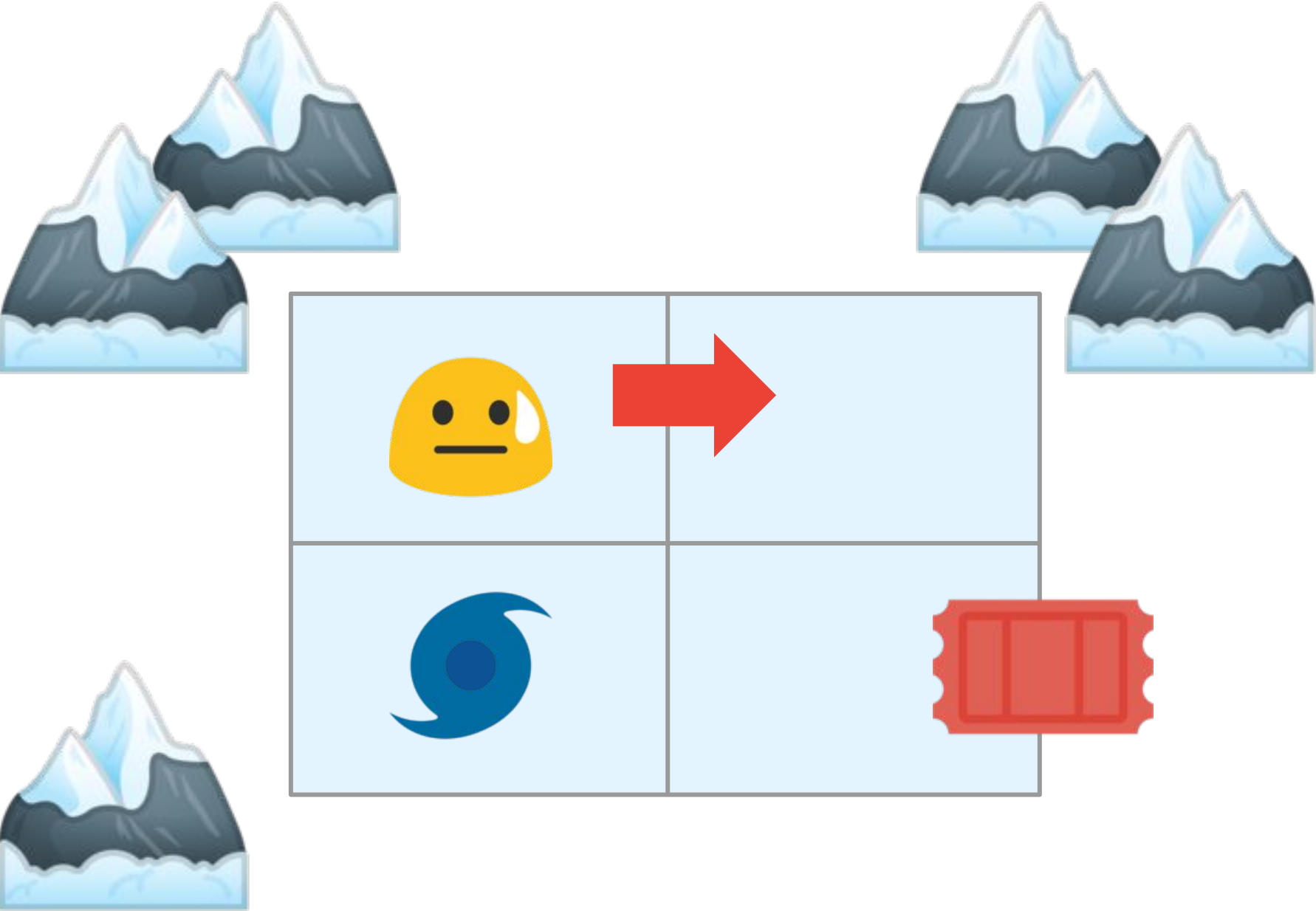


Q - table				
	Left	Down	Right	Up
0	0	-.5	0	0
1	0	0	0	0
2	0	0	0	0
3	0	0	0	0

$\gamma = .9$

$\alpha = .5$

The Q Table



Q - table				
	Left	Down	Right	Up
0	0	-.5	0	0
1	0	0	0	0
2	0	0	0	0
3	0	0	0	0

$$\gamma = .9$$

$$\alpha = .5$$

Deep Q Learning

$$V(s_{t-1}) = V(s_{t-1}) + \alpha_t (R(s_{t-1}, a_{t-1}) + \gamma \cdot V(s_t) - V(s_{t-1}))$$

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha_t (r_t + \gamma \cdot \max_a \{Q(s_{t+1}, a)\} - Q(s_t, a_t))$$

Deep Q Learning

$$V(s_{t-1}) = V(s_{t-1}) + \alpha_t (R(s_{t-1}, a_{t-1}) + \gamma \cdot V(s_t) - V(s_{t-1}))$$

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha_t (r_t + \gamma \cdot \max_a \{Q(s_{t+1}, a)\} - Q(s_t, a_t))$$

Anatomy of an Agent

```
class Agent():  
    def __init__(num_states, num_actions, discount, learning_rate):  
        ...  
  
    def update_q(self, state, action, reward, state_prime)  
        ...  
  
    def act(self, state):  
        ...
```


Anatomy of an Agent

```
class Agent():
    def __init__(num_states, num_actions, discount, learning_rate):
        self.discount = discount
        self.learning_rate = learning_rate
        self.q_table = np.zeros((num_states, num_actions))

    def update_q(self, state, action, reward, state_prime)
        ...

    def act(self, state):
        ...
```

Anatomy of an Agent

```
class Agent():
    def __init__(num_states, num_actions, discount, learning_rate):
        self.discount = discount
        self.learning_rate = learning_rate
        self.q_table = np.zeros((num_states, num_actions))

    def update_q(self, state, action, reward, state_prime):
        alpha = self.learning_rate
        future_value = reward + self.discount * np.max(q_table[state_prime])
        old_value = q_table[state, action]
        q_table[state, action] = old_value + alpha * (future_value - old_value)

    def act(self, state):
        ...
```

Anatomy of an Agent

```
class Agent():
    def __init__(num_states, num_actions, discount, learning_rate):
        self.discount = discount
        self.learning_rate = learning_rate
        self.q_table = np.zeros((num_states, num_actions))

    def update_q(self, state, action, reward, state_prime):
        alpha = self.learning_rate
        future_value = reward + self.discount * np.max(q_table[state_prime])
        old_value = q_table[state, action]
        q_table[state, action] = old_value + alpha * (future_value - old_value)

    def act(self, state):
        action_values = q_table[state_row]
        max_indexes = np.argwhere(action_values == action_values.max())
        max_indexes = np.squeeze(max_indexes, axis=-1)
        action = np.random.choice(max_indexes)
        return action
```

On Purpose Mistakes?



Anatomy of an Agent

```
class Agent():
    def __init__(..., learning_rate, random_rate):
        ...
        self.num_actions = num_actions
        self.random_rate = random_rate # I'm between 0 and 1.

    def update_q(self, state, action, reward, state_prime)
        ...

    def act(self, state, training=True):
        if random.random() < self.random_rate and training:
            return random.randint(0, self.num_actions-1)

        action_values = q_table[state_row]
        max_indexes = np.argwhere(action_values == action_values.max())
        ...
        return action
```

Anatomy of an Agent

```
EPISODES = 1000
agent = AGENT(NUM_STATES, NUM_ACTIONS, DISCOUNT, LEARNING_RATE, RANDOM_RATE)
environment = gym.make('FrozenLake-v0')

def play_game(environment, agent):
    state = environment.reset()
    done = False

    while not done:
        action = agent.act(state)
        state_prime, reward, done = environment.step(action)
        agent.update_q(state, action, reward, state_prime)
        state = new_state

for episode in range(EPISODES):
    play_game(environment, agent)
```

Anatomy of an Agent

```
EPISODES = 1000
agent = AGENT(NUM_STATES, NUM_ACTIONS, DISCOUNT, LEARNING_RATE, RANDOM_RATE)
environment = gym.make('FrozenLake-v0')

def play_game(environment, agent):
    state = environment.reset()
    done = False

    while not done:
        action = agent.act(state)
        state_prime, reward, done = environment.step(action)
        agent.update_q(state, action, reward, state_prime)
        state = state_prime

for episode in range(EPISODES):
    play_game(environment, agent)
```


File Name:

T-AIFORF-I-p3_M1_I10_benefits_of_using_reinforcement_learning_in_your_trading_strategy_part1

Content Type: Video - Lecture Presenter

Presenter: Jack Farmer



Benefits of Reinforcement Learning in Your Trading Strategy

Learning Objectives

- Understand the difference between deep learning (DL) and deep reinforcement learning (DRL)
- Identify the components of a deep reinforcement learning trading strategy
- Identify the advantages of DRL that can help it improve the efficiency and performance of quantitative strategies

Agenda

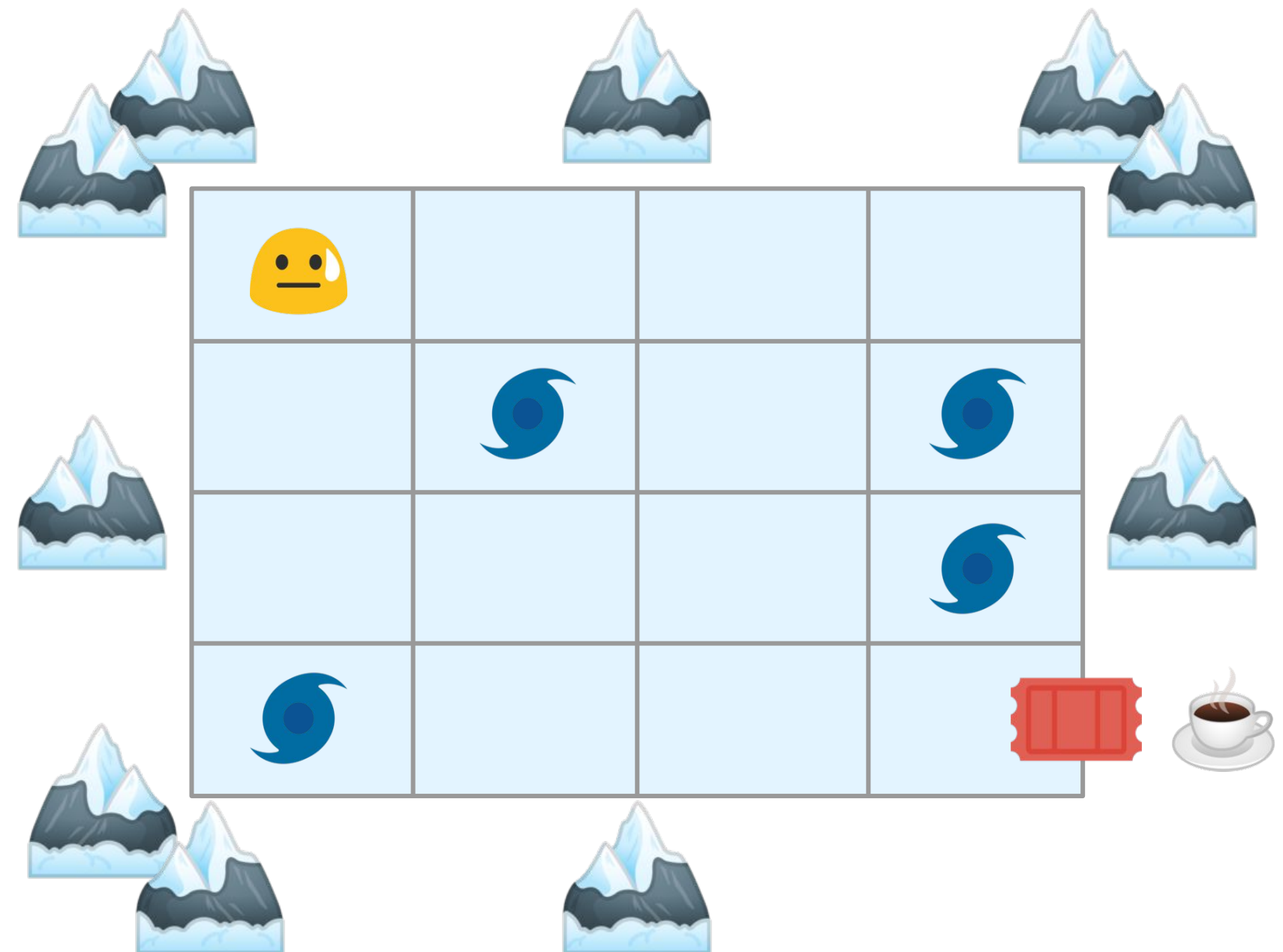
What is Deep Reinforcement Learning?

How to Use DRL in Trading Strategies

DRL Advantages for Strategy Efficiency and Performance

What is DRL?

- Naive Agent
- Unknown Environment
- No knowledge or experience
- Goal is to collect information by taking actions

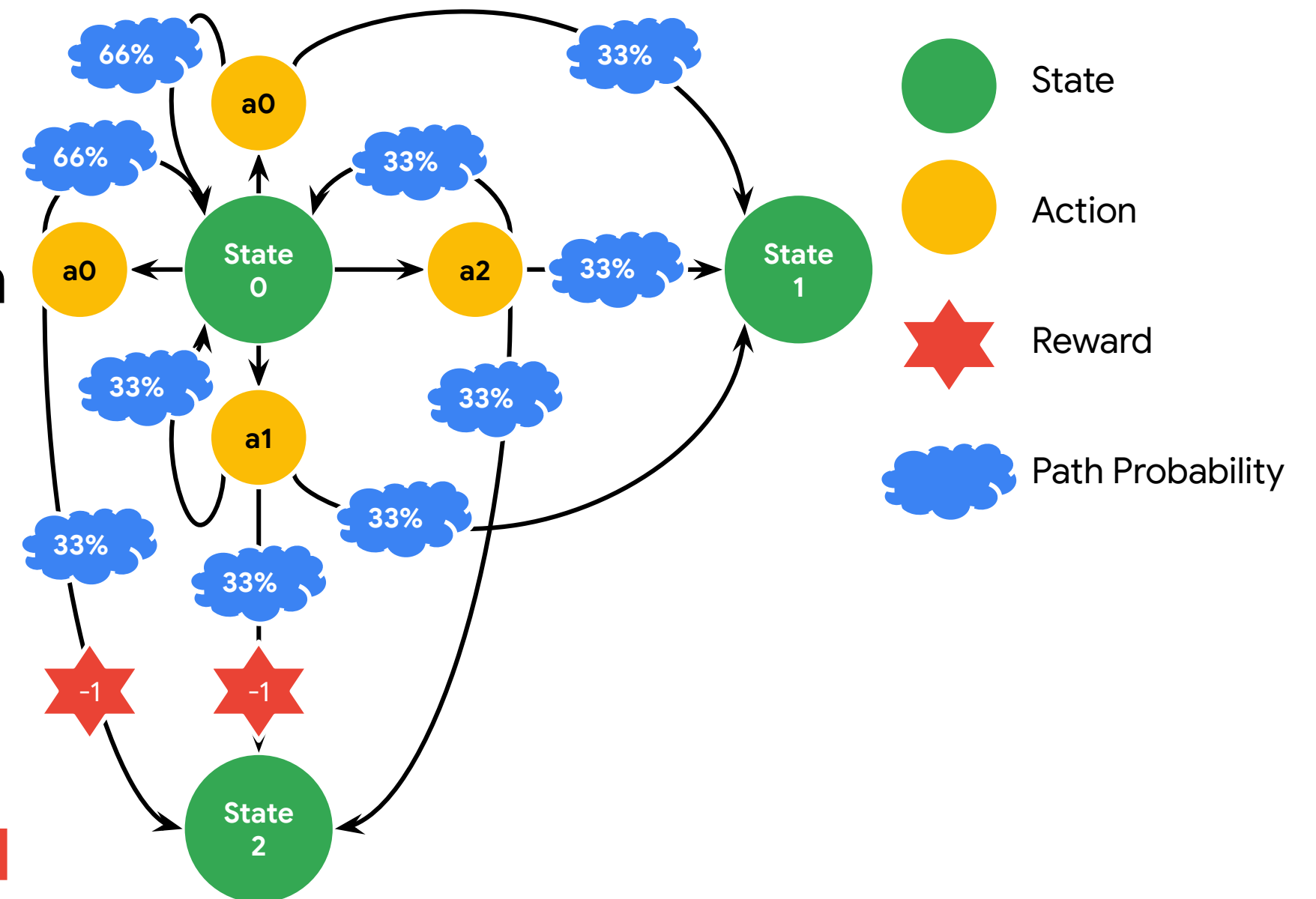


DRL Agent

- Tests State Spaces
- Action => Reaction? = New State?
- Needs input to distinguish between “bad” and “good” decision
- Developer sets rewards and penalties

Interaction => Knowledge

=> Better Decisions => Max Reward



DRL Agent vs DL Agent

- DRL Agents given a high degree of freedom
- Build on and develop initial logic based on experience
- Become independent operators with their own experience-based logic
- Can **extend beyond developer's knowledge** and **solve more complex problems**

Agenda

What is Deep Reinforcement Learning?

How to Use DRL in Trading Strategies

DRL Advantages for Strategy Efficiency and Performance

Trading Challenges

- Strategies require error-free handling of large volumes of data
- Agents' actions may result in longer-term consequences that other ML techniques are unable to measure
- And also have short-term impacts on the current market conditions which makes the trading environment highly unpredictable

Trading Challenges

- Strategies require error-free handling of large volumes of data
- Agents' actions may result in longer-term consequences that other ML techniques are unable to measure
- And also have short-term impacts on the current market conditions which makes the trading environment highly unpredictable

Trading Challenges

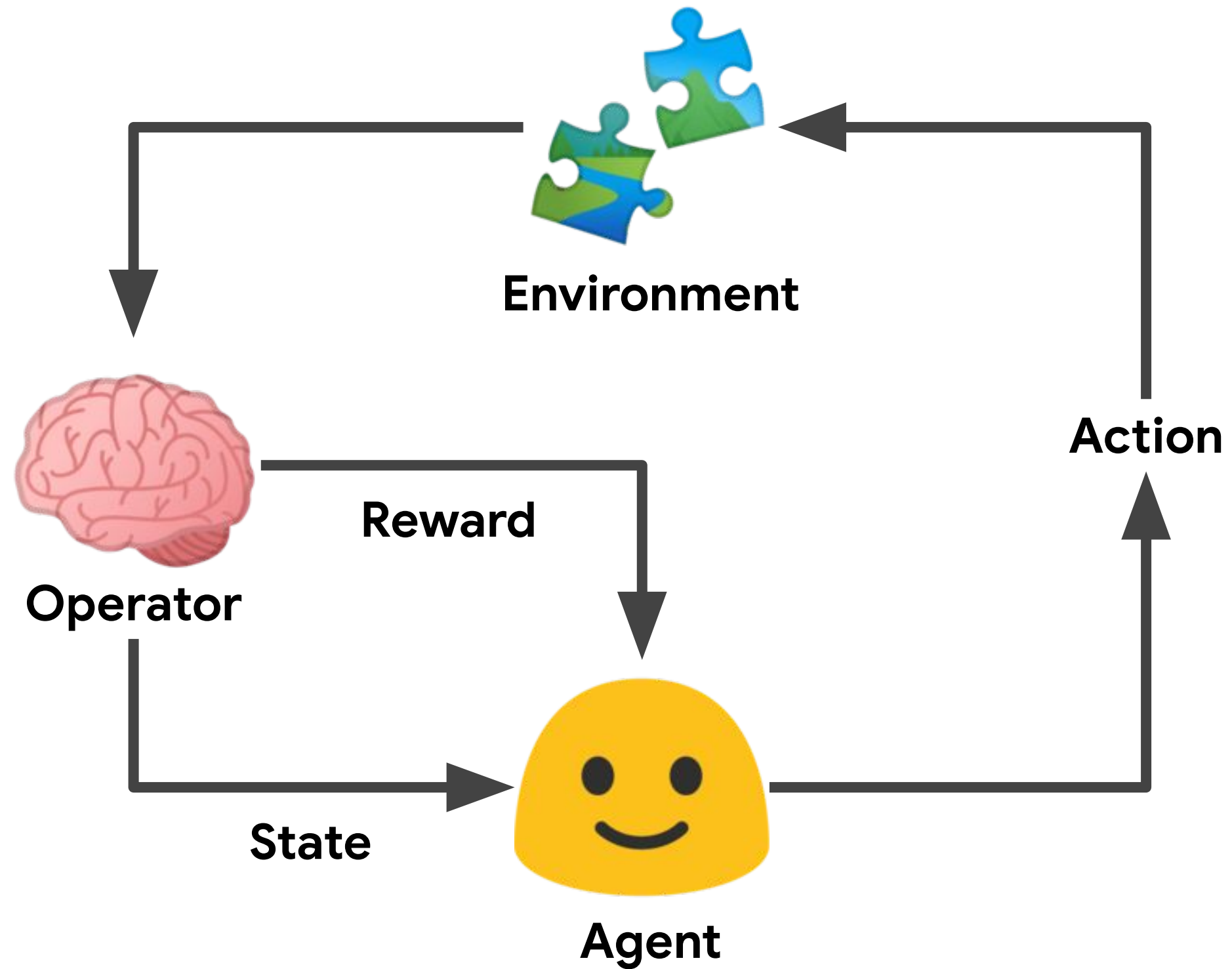
- Strategies require error-free handling of large volumes of data
- Agents' actions may result in longer-term consequences that other ML techniques are unable to measure
- And also have short-term impacts on the current market conditions which makes the trading environment highly unpredictable

Trading Challenges

- Strategies require error-free handling of large volumes of data
- Agents' actions may result in longer-term consequences that other ML techniques are unable to measure
- And also have short-term impacts on the current market conditions which makes the trading environment highly unpredictable

DRL Trading Algorithm Components

1. Agent
2. Environment
3. State
4. Reward



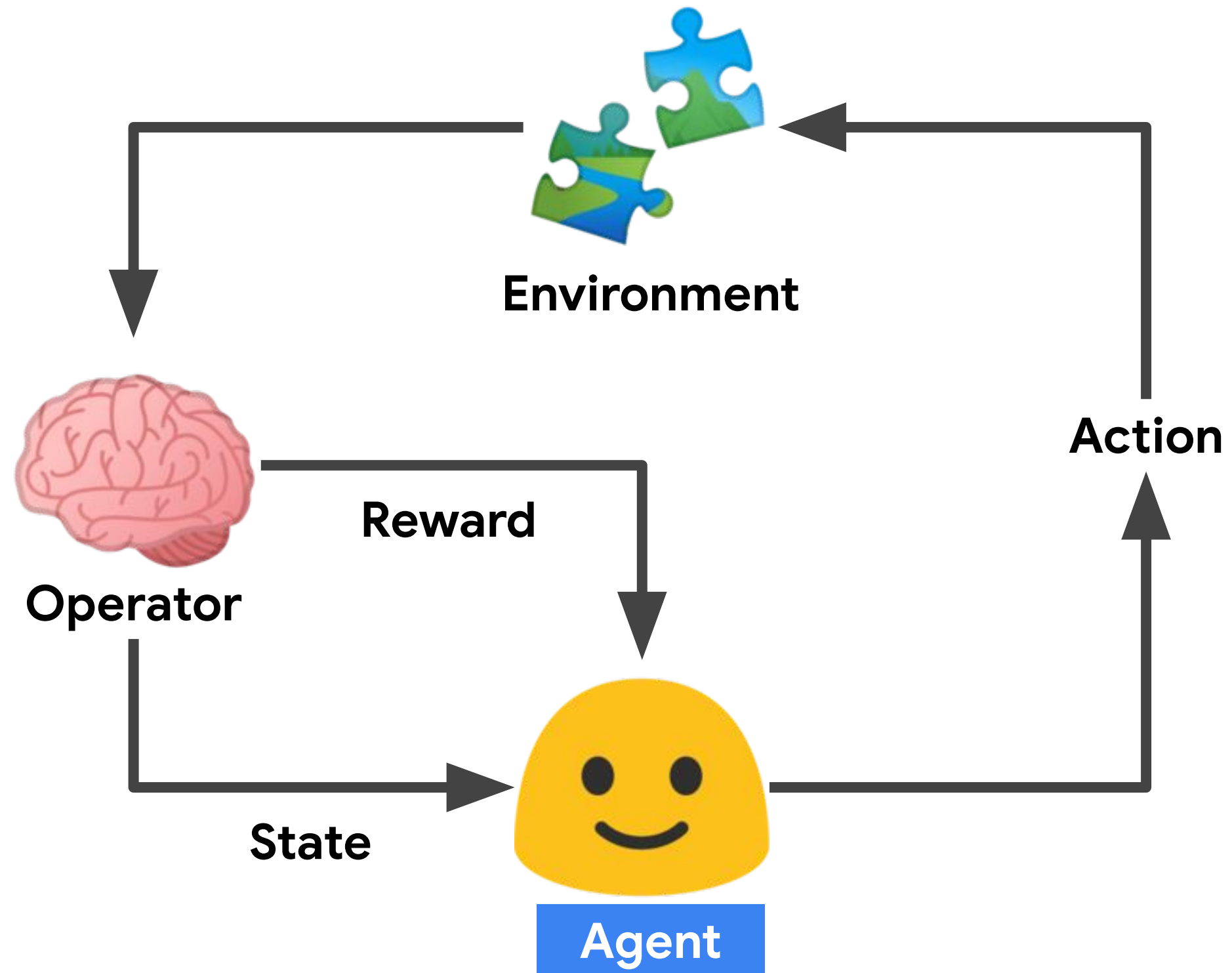
DRL Trading Algorithm Components

1. Agent

2. Environment

3. State

4. Reward



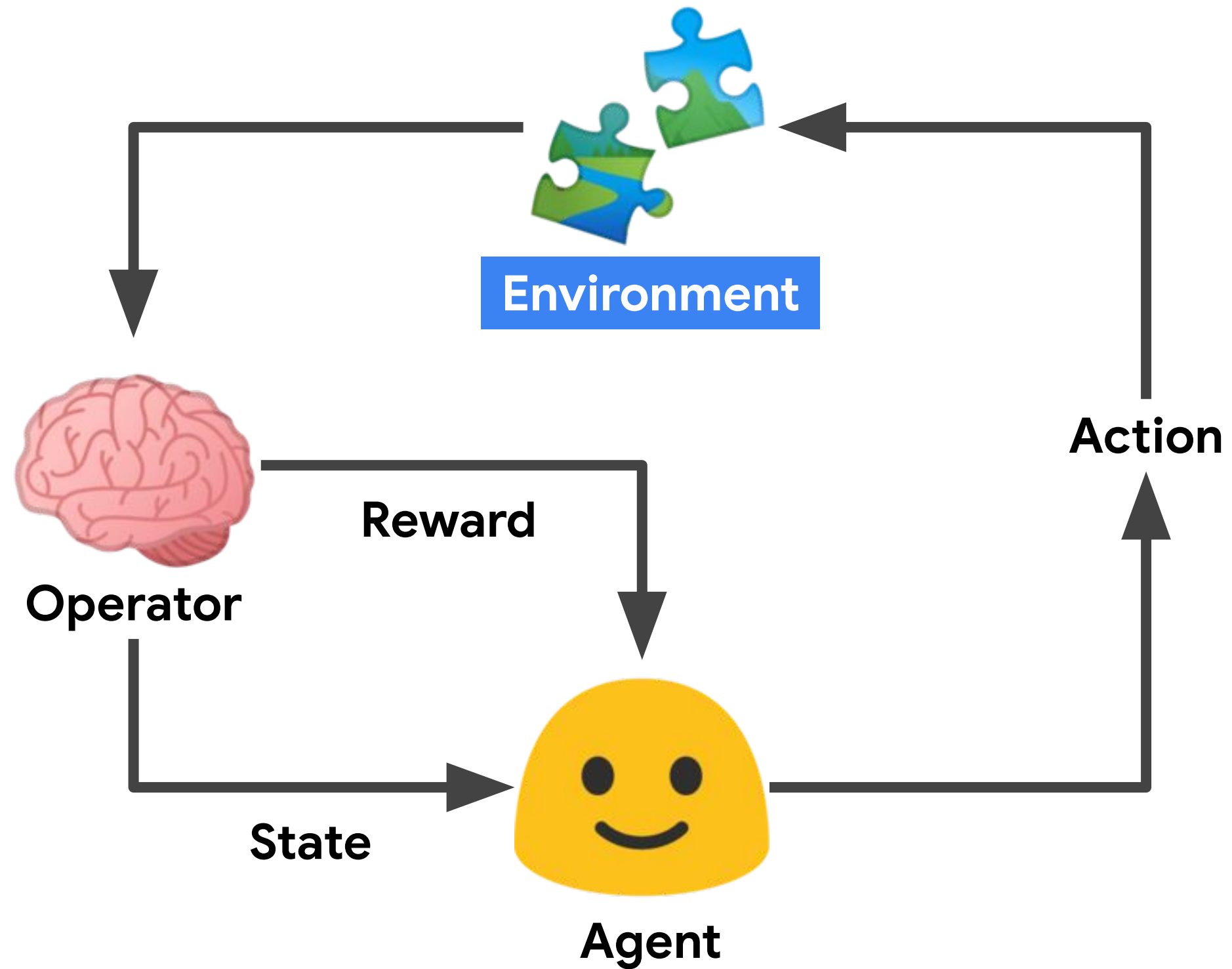
DRL Trading Algorithm Components

1. Agent

2. Environment

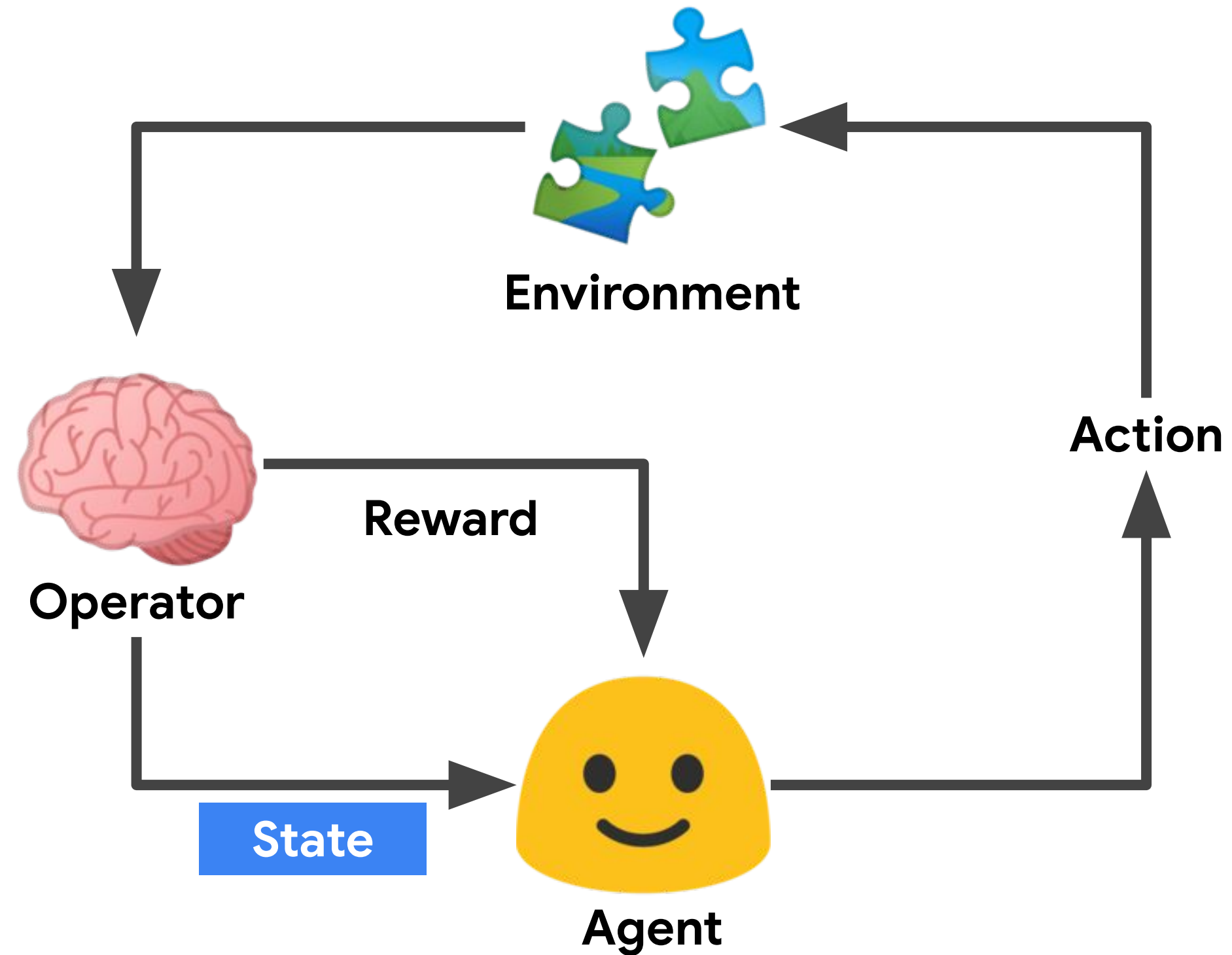
3. State

4. Reward



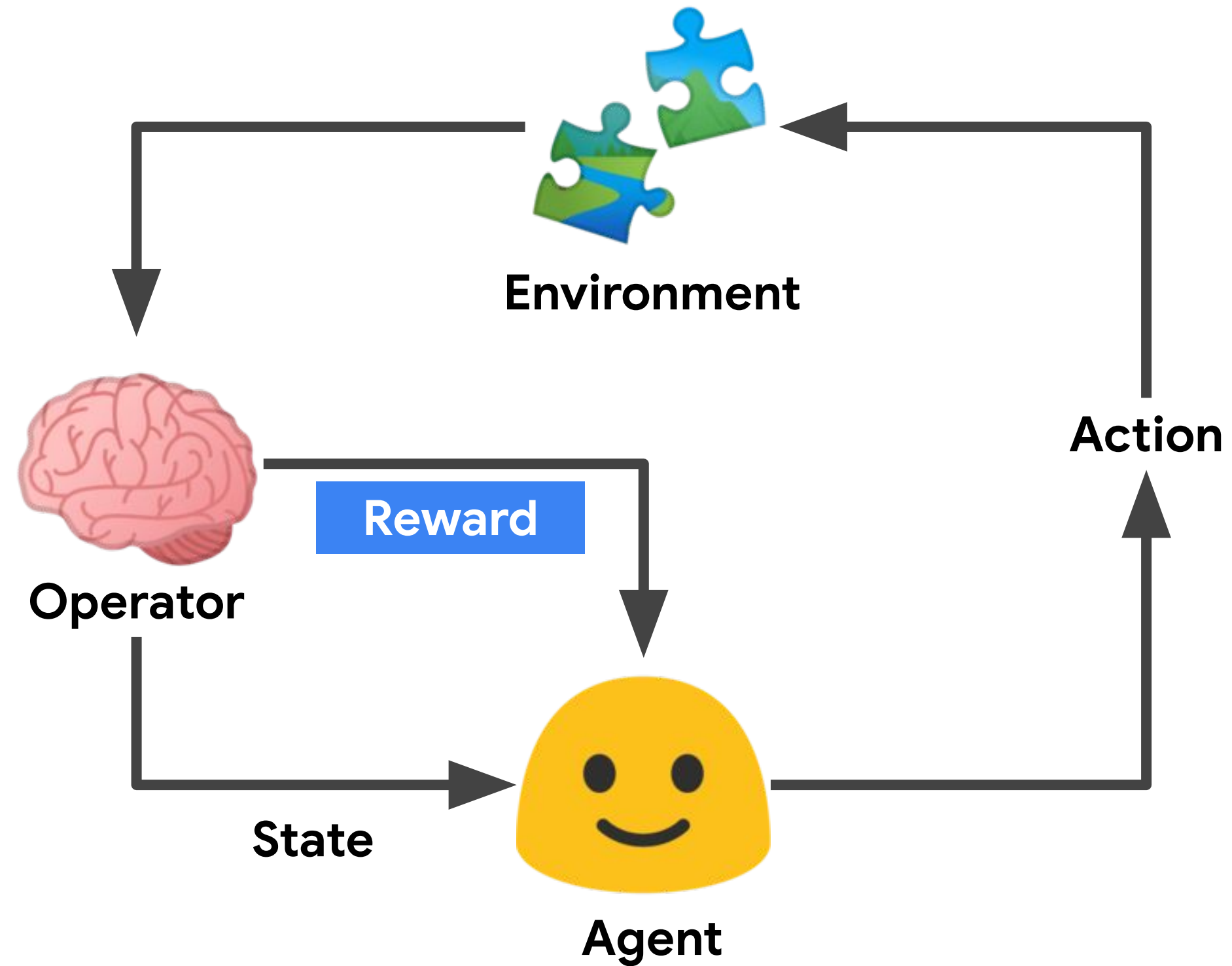
DRL Trading Algorithm Components

1. Agent
2. Environment
3. State
4. Reward



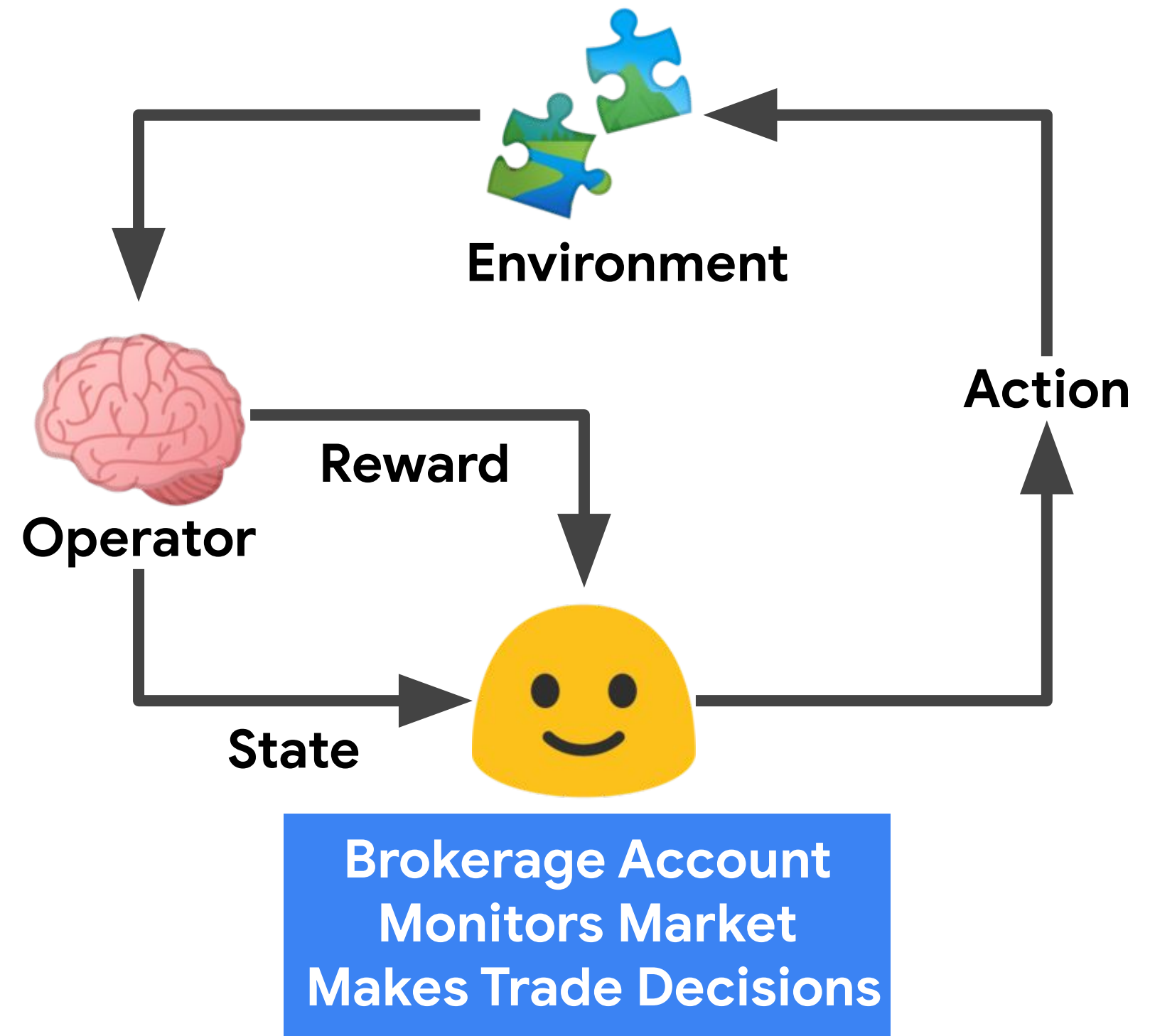
DRL Trading Algorithm Components

1. Agent
2. Environment
3. State
4. Reward



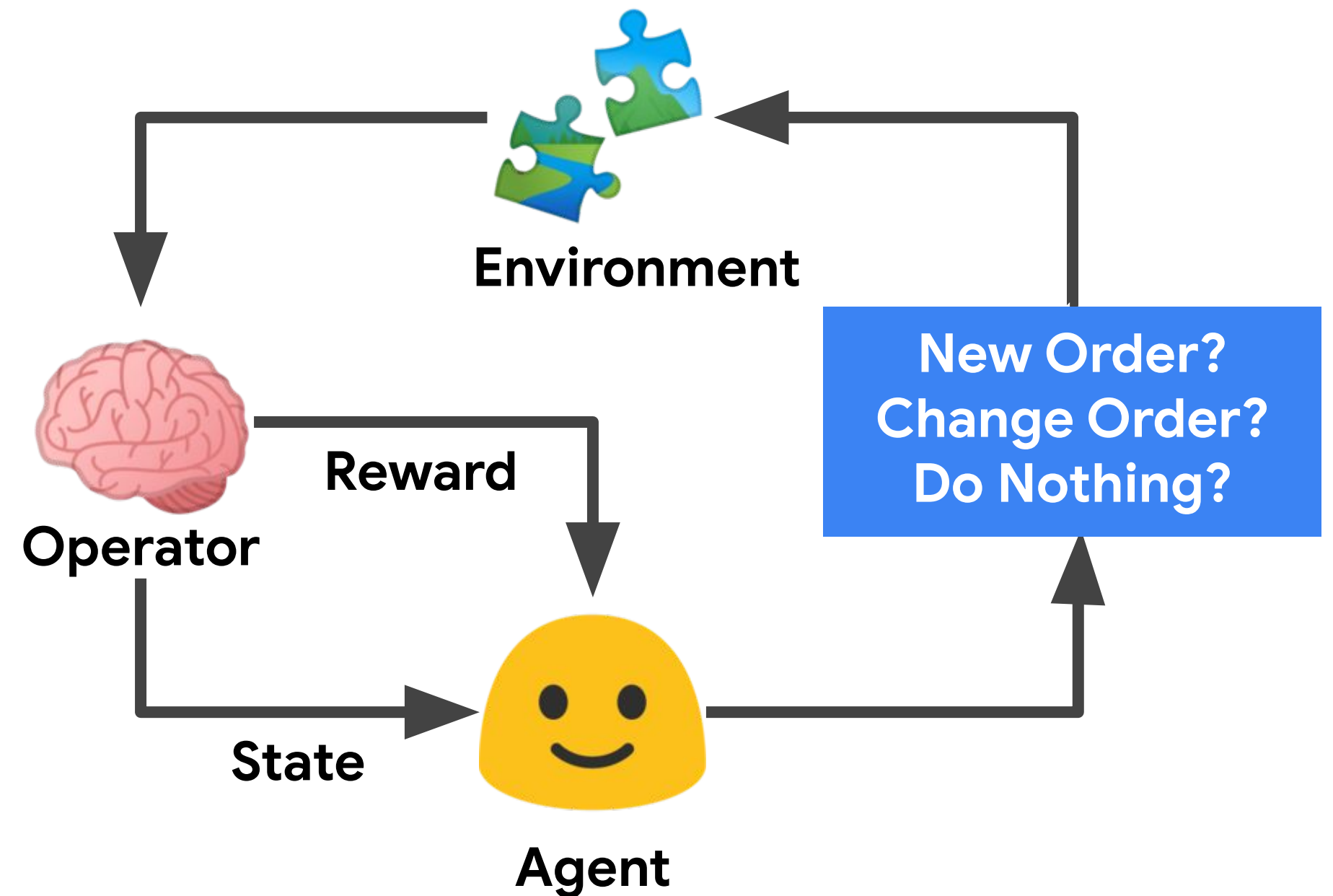
DRL Agent

- Agent = Trader
- Access to brokerage account
- Monitors market conditions
- Makes trading decisions



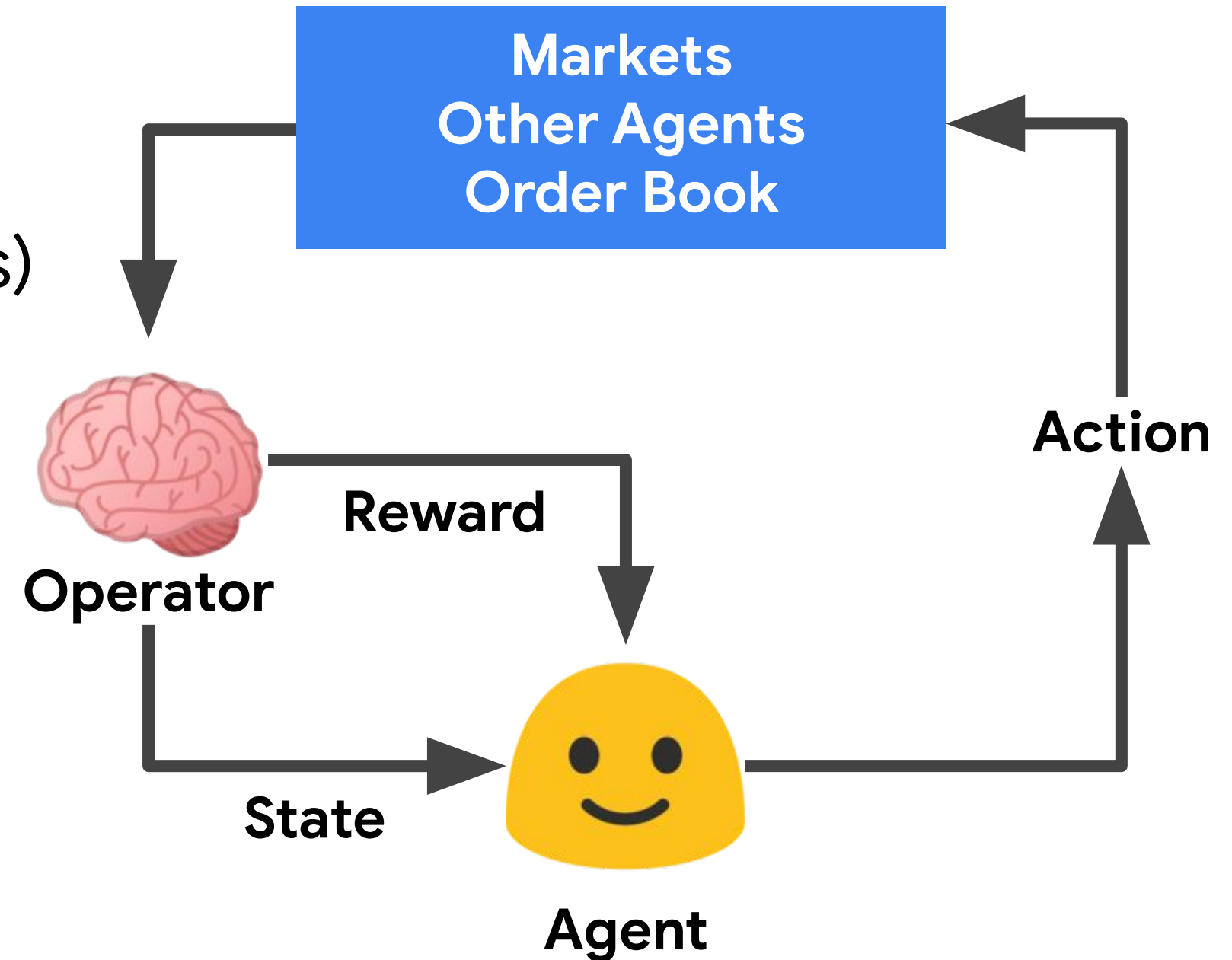
Agent/Algo Methodology

1. Make Trading Decision \Rightarrow Order **Filled** or **Not Filled**?
2. Assess New Market Conditions
3. Make Decision
 - \Rightarrow New Order?
 - \Rightarrow Change Order?
 - \Rightarrow Do nothing?



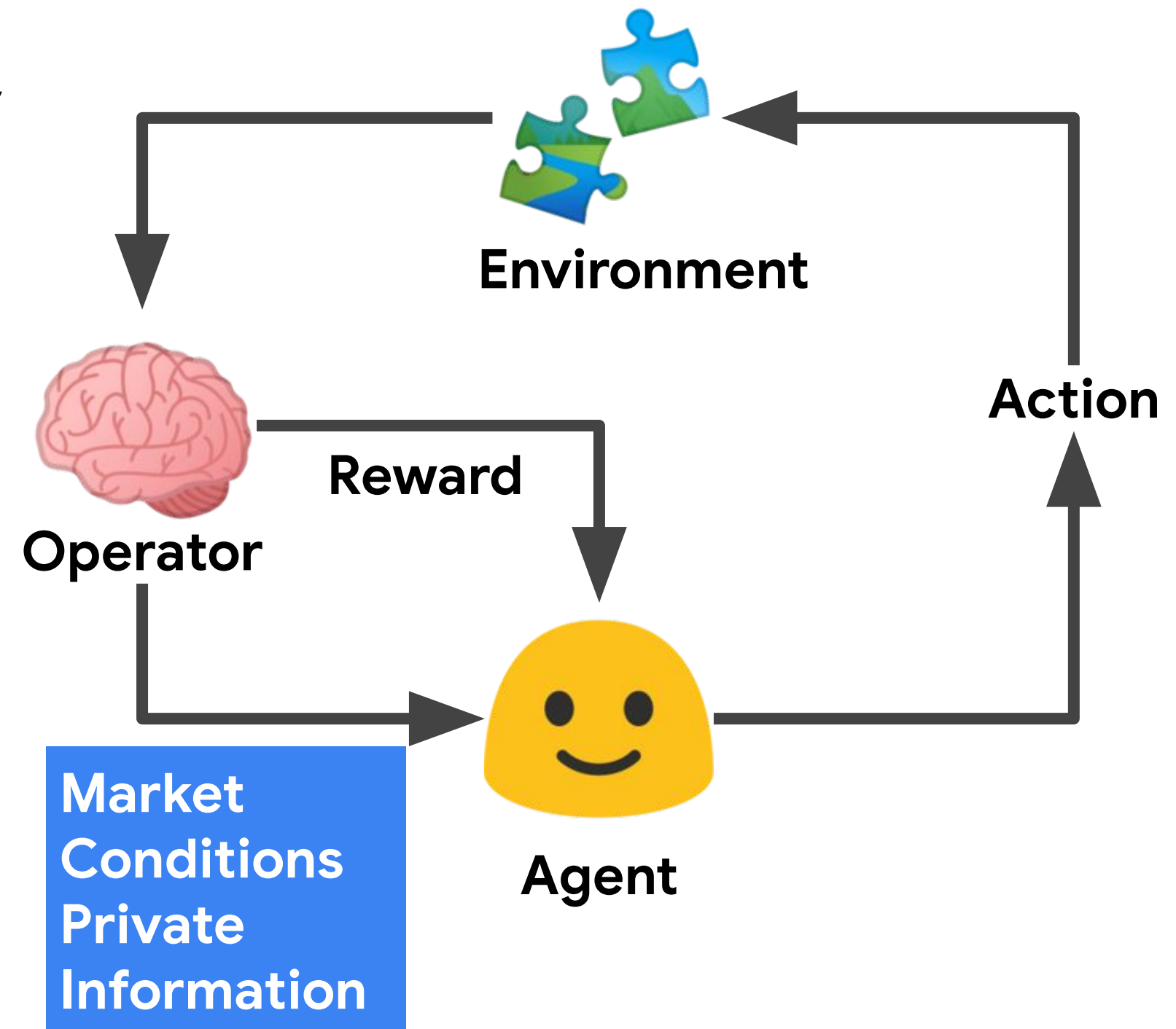
DRL Environment

- Market(s)
- Other agents (algos and humans)
- Order Book (public liquidity)
- Order Execution Strategies (hidden liquidity)



State

- Market Conditions (only partially knowable by Agent)
- Unknowable:
 - Number of other agents
 - Their actions and positions
 - Their order specifications
- Advantage gained from private information or tech superiority



File Name:

T-AIFORF-I-p3_M1_I11_benefits_of_using_reinforcement_learning_in_your_trading_strategy_part2

Content Type: Video - Lecture Presenter

Presenter: Jack Farmer

Agenda

What is Deep Reinforcement Learning?

How to Use DRL in Trading Strategies

DRL Advantages for Strategy Efficiency and Performance

DRL's Key Advantages

1. The self-learning process is a good match for a rapidly evolving market environment
2. Brings more power and efficiency to a dense and complex state space
3. It builds on machine learning techniques that have already proven successful in a variety of markets

Good Match for Markets

- Financial markets are dynamic and turbulent structures
- Increased volatility and unstable liquidity lead to periodic flash crashes
- Complex quantitative strategies and technologically enhanced participants create short-lived, hard to identify patterns
- Historical data quickly becomes irrelevant for predicting current market movements

Good Match for Markets

- Even the most successful trading firms are being forced to adapt
- RenTech's RIDA fund has reduced the use of pattern-based strategies by over 60%¹
- Other hedge funds have also given up trend following as they struggle to replicate past returns

¹ Hedgefundresearch.com 2019

Good Match for Markets

- Automated strategies must be flexible and not completely dependent on past data
- DRL can learn on the go by doing, just like humans, but faster
- DRL algos are getting better at taking real-time decisions based on current market conditions and the immediate results of their actions

Power and Efficiency

- Traders must factor in many market variables to make the set of interconnected decisions that comprise an order
- Price, size, order time, duration, and type require decisions on:
 - What price to buy/sell?
 - What quantity?
 - How many orders?
 - Sequentially or simultaneously?

Power and Efficiency

- A medium frequency trading algo will reconsider its options every second*
- Each action results in orders with unique characteristics
- Financial Markets are too complex for straightforward algorithms
- Their action space is continuously expanding with possible order combinations dependent on a dynamically changing market state

* "Idiosyncrasies and challenges of data driven learning in electronic trading"
(JPM November 30, 2018 <https://arxiv.org/pdf/1811.09549.pdf>)

Builds on Successful ML Techniques

- Algo strategies consist of:
 - Strategy
 - Implementation
- Designed by trader and implemented by a machine
- Human-machine symbiosis often breaks down and performs poorly

Builds on Successful ML Techniques

- One of the main challenges is selecting un-biased, representative financial data
- Although widely recognized this task is often poorly implemented (usually by the trader)
- With advancement in DRL we are getting closer to an Autonomous machine in charge of both strategy and implementation

Remaining Challenges to Creating a DRL Trader

- DRL still requires millions of test scenarios to trade profitably and is dependent on an operator to structure rewards
- Reward design is tricky and has potential to make or break a trading system
- Still we are closer to full automation than ever before

Remaining Challenges to Creating a DRL Trader

- DRL still requires millions of test scenarios to trade profitably and is dependent on an operator to structure rewards
- Reward design is tricky and has potential to make or break a trading system
- Still we are closer to full automation than ever before

Remaining Challenges to Creating a DRL Trader

- DRL still requires millions of test scenarios to trade profitably and is dependent on an operator to structure rewards
- Reward design is tricky and has potential to make or break a trading system
- Still we are closer to full automation than ever before

Lab

Use Deep Q Framework
for a Buy/Sell Strategy

Lab Objectives

-
-

Screencast