Google Cloud

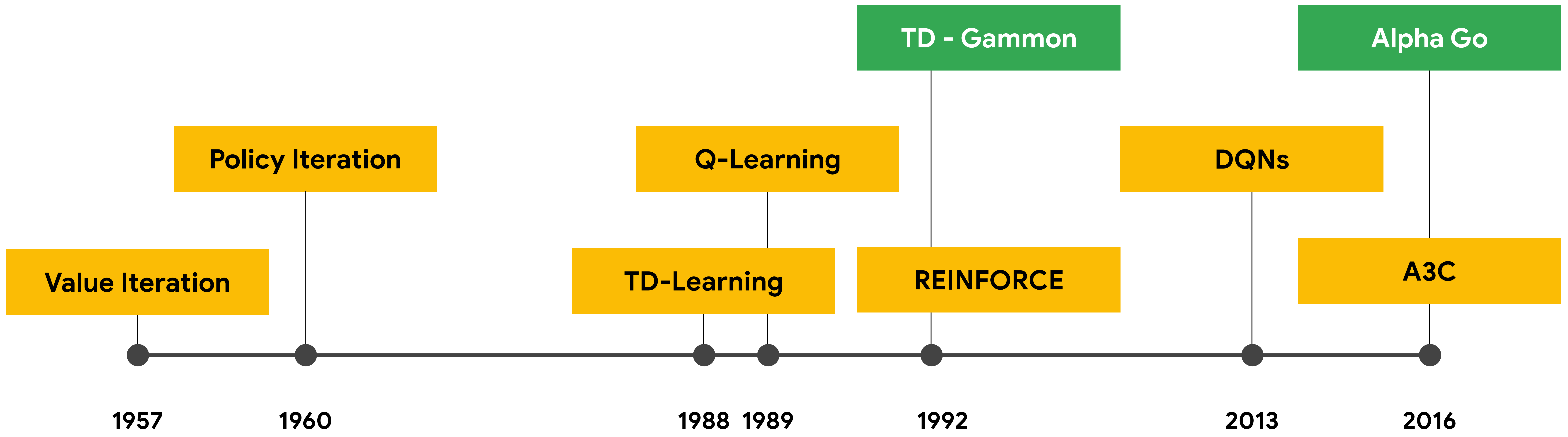
Early Reinforcement Learning

Learning Objectives

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

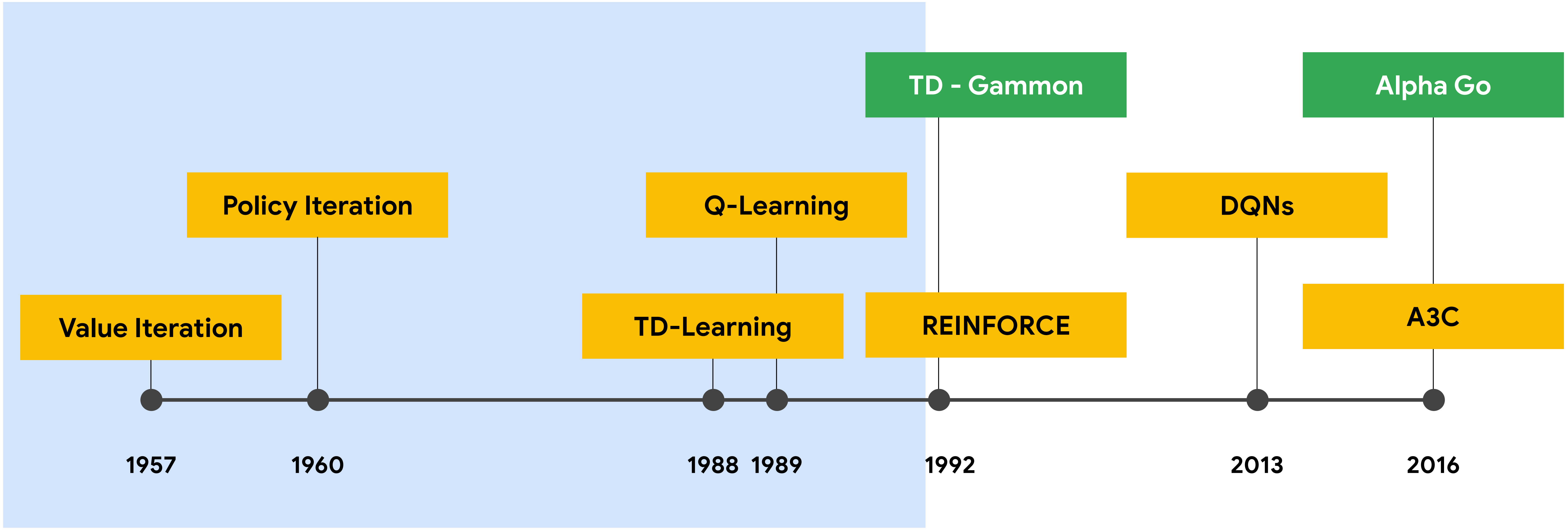
Agenda

An RL Timeline



Google Cloud

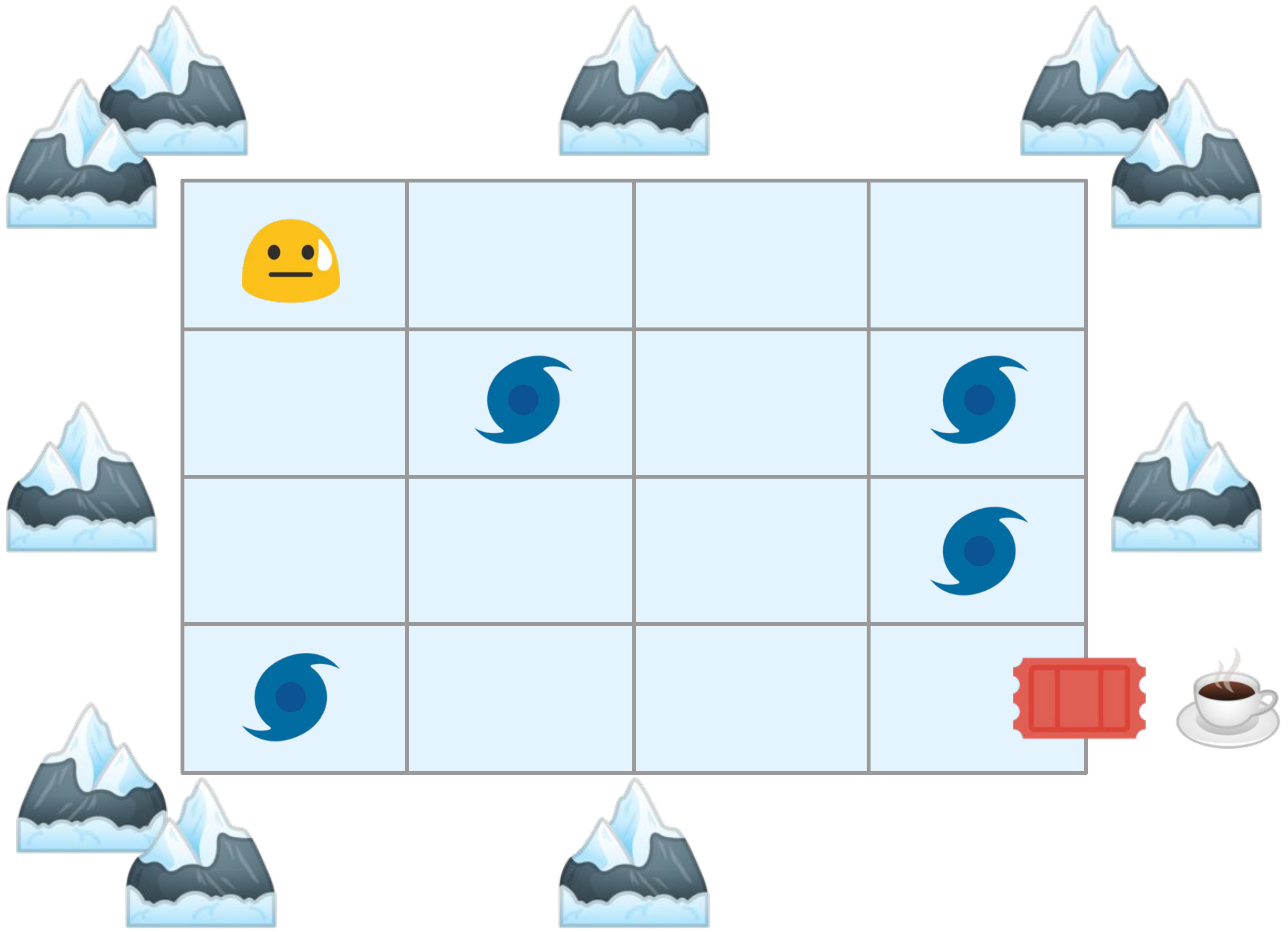
An RL Timeline



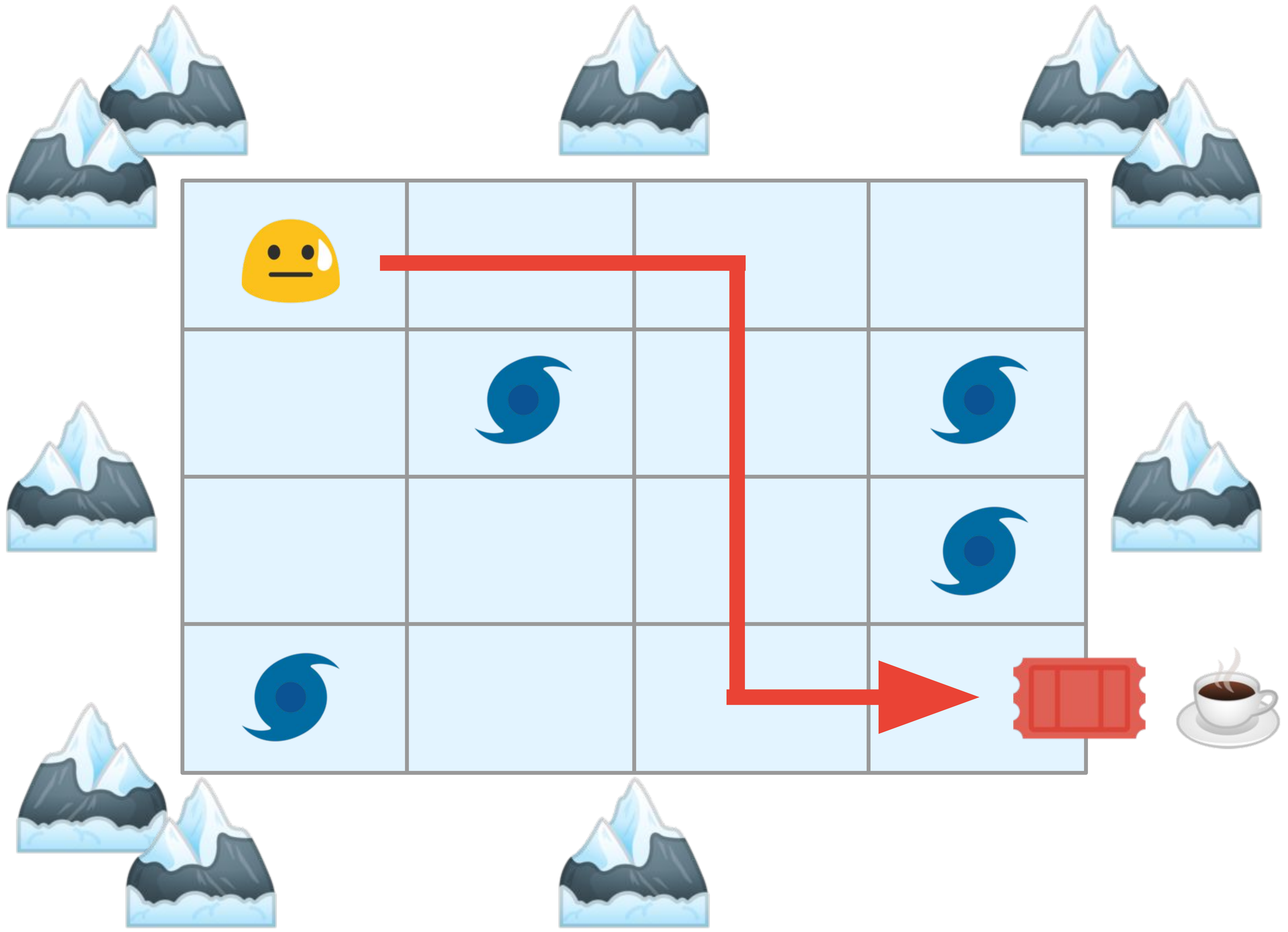
A Simple Story

Google Cloud

Frozen Lake



Frozen Lake



Google Cloud

Agenda

History Overview

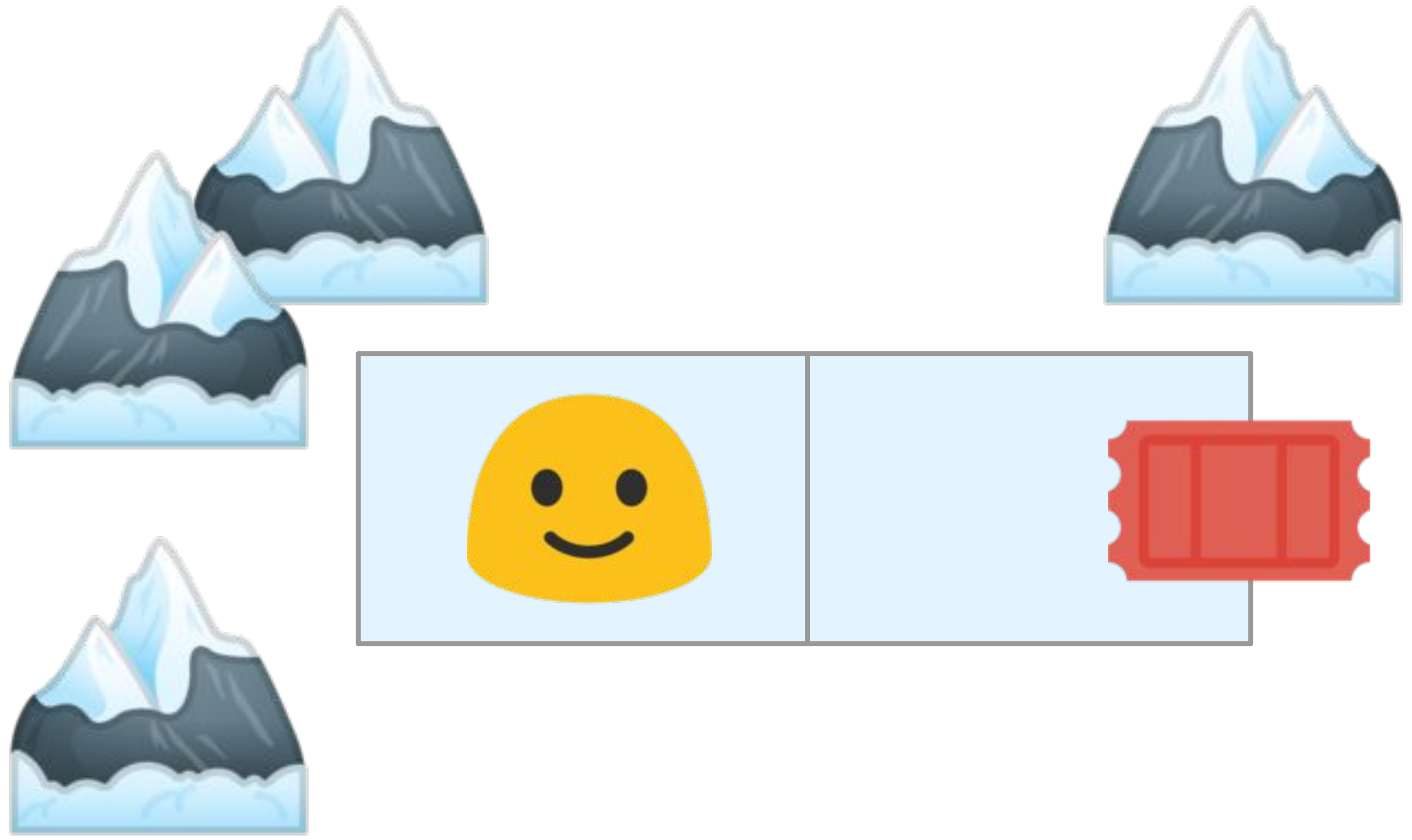
Value Iteration

Policy Iteration

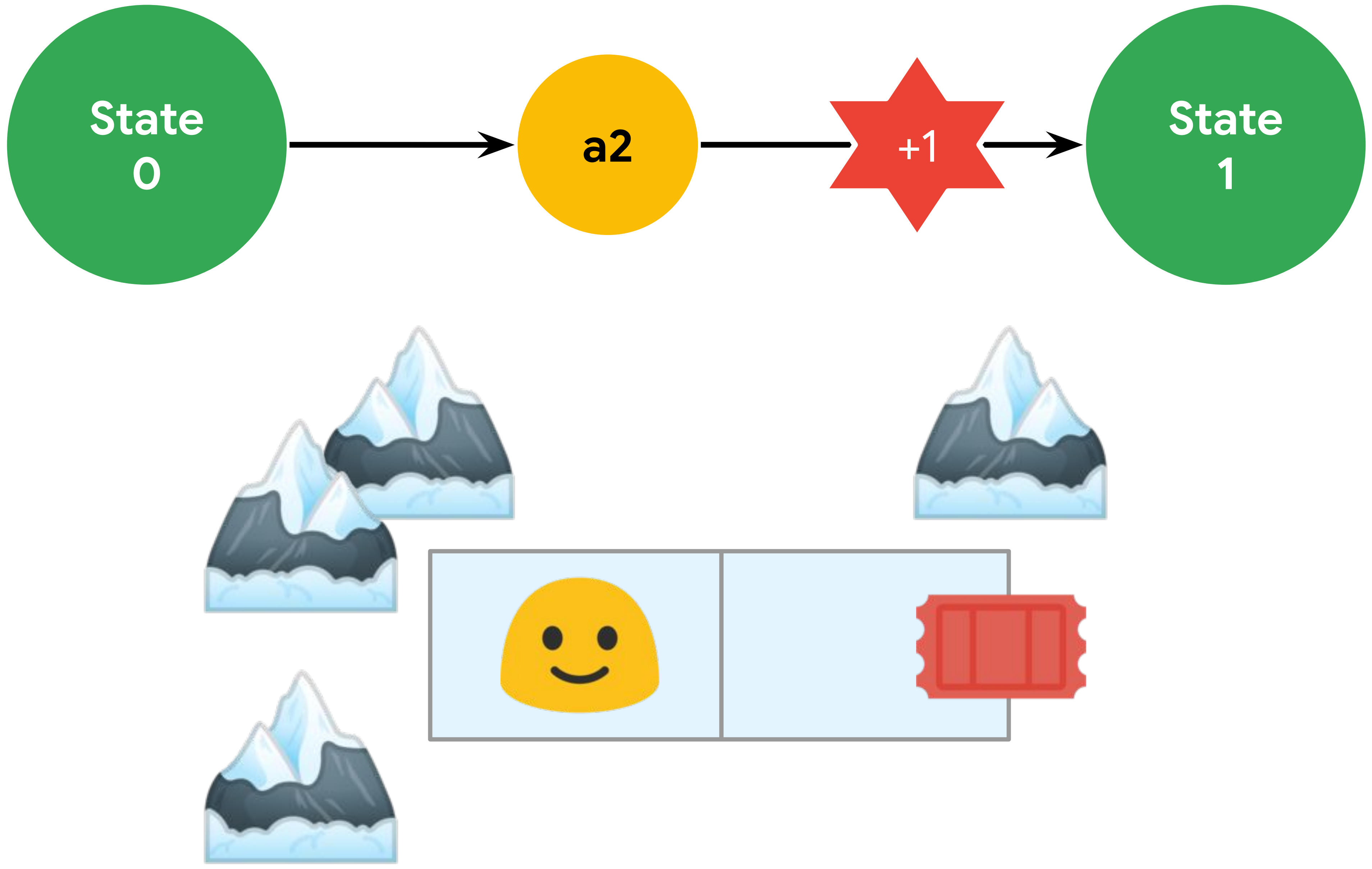
TD(Lambda)

Q-Learning

A Simpler Lake



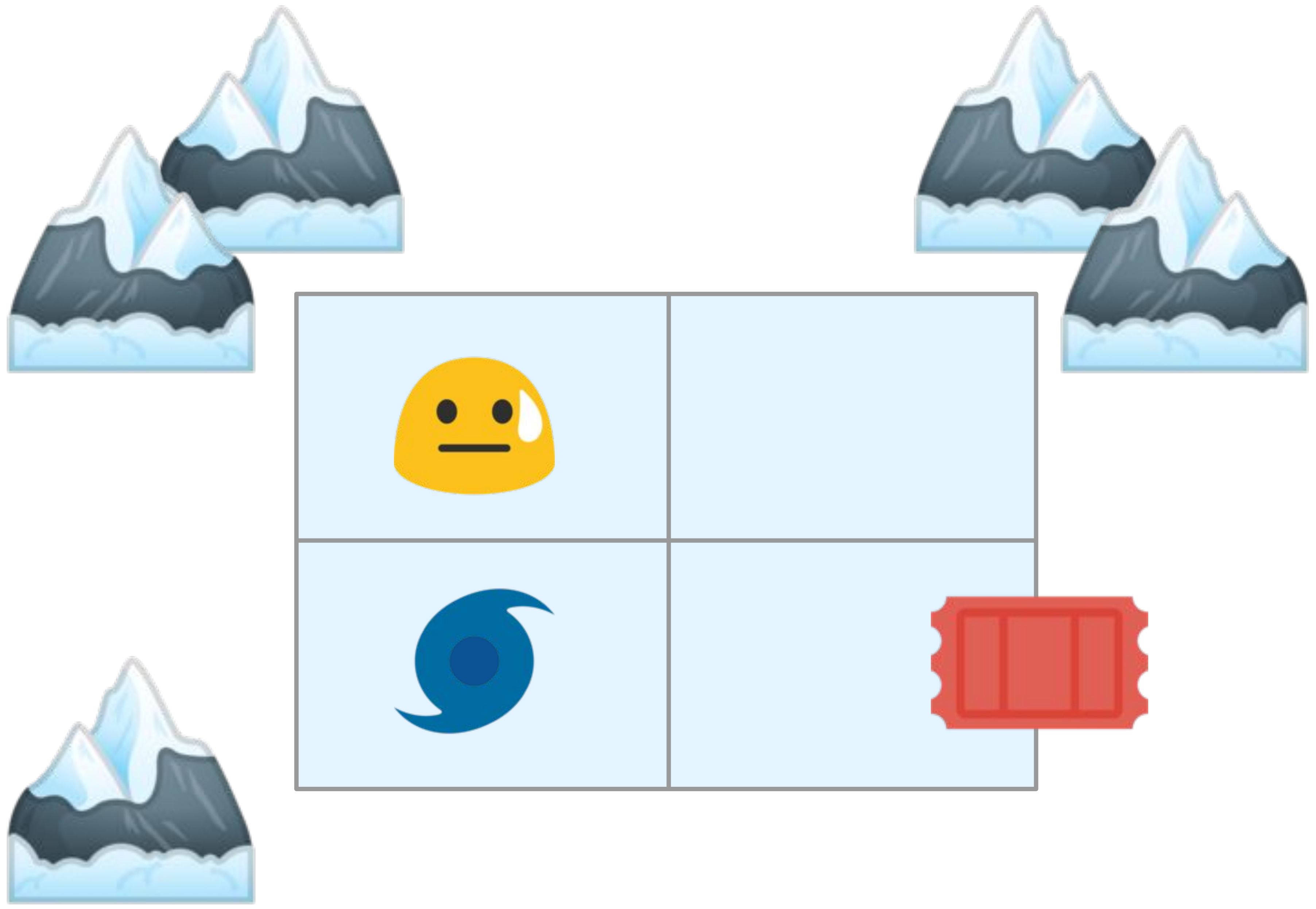
Markov Decision Process (MDP)

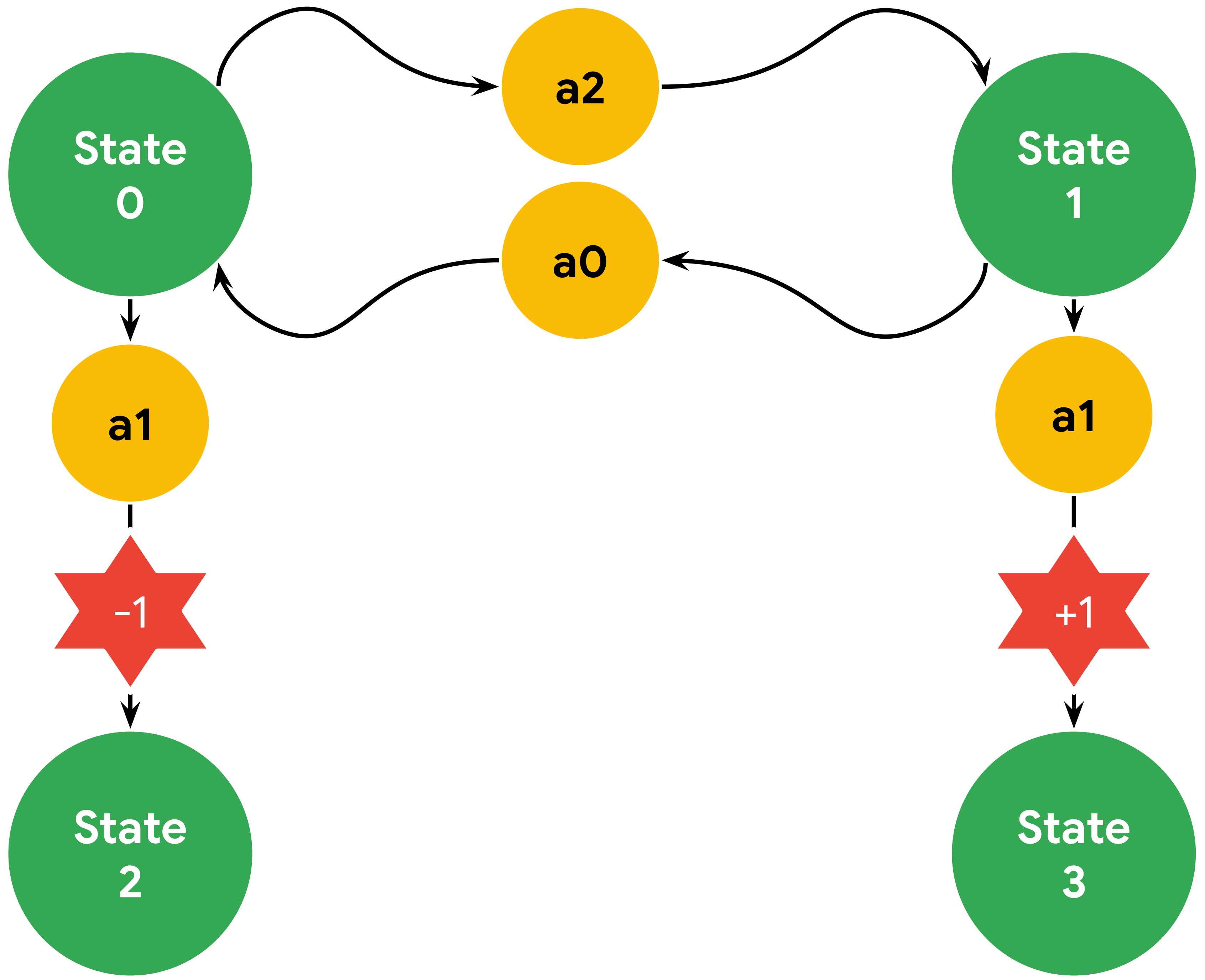


●	State
●	Action
✶	Reward

Google Cloud

A Simpler Lake





Google Cloud

Bellman Equation

$$\underbrace{V(s)}_{\text{Value of the current state}} = \overbrace{R(s,a)}^{\text{Rewards received}} + \underbrace{\gamma V(s')}_{\text{Discounted future state}}$$

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

Google Cloud

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

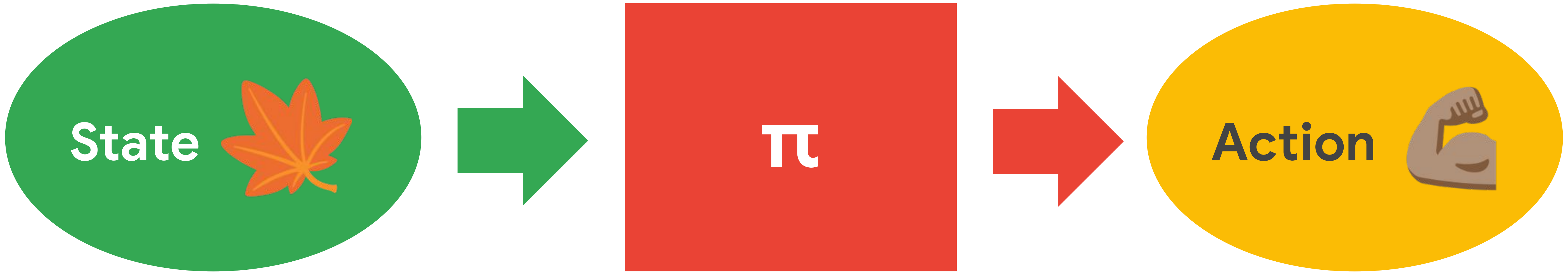
The Discount Factor (γ)

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

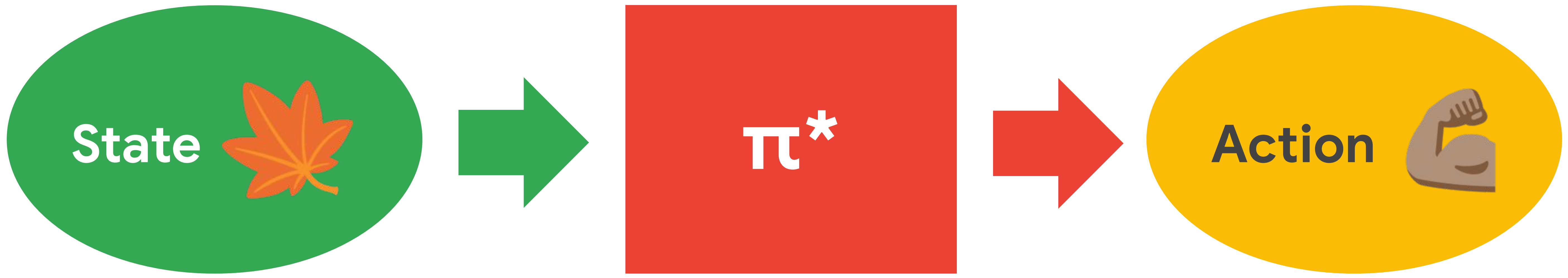
Bellman Equation

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

The Policy



The Policy

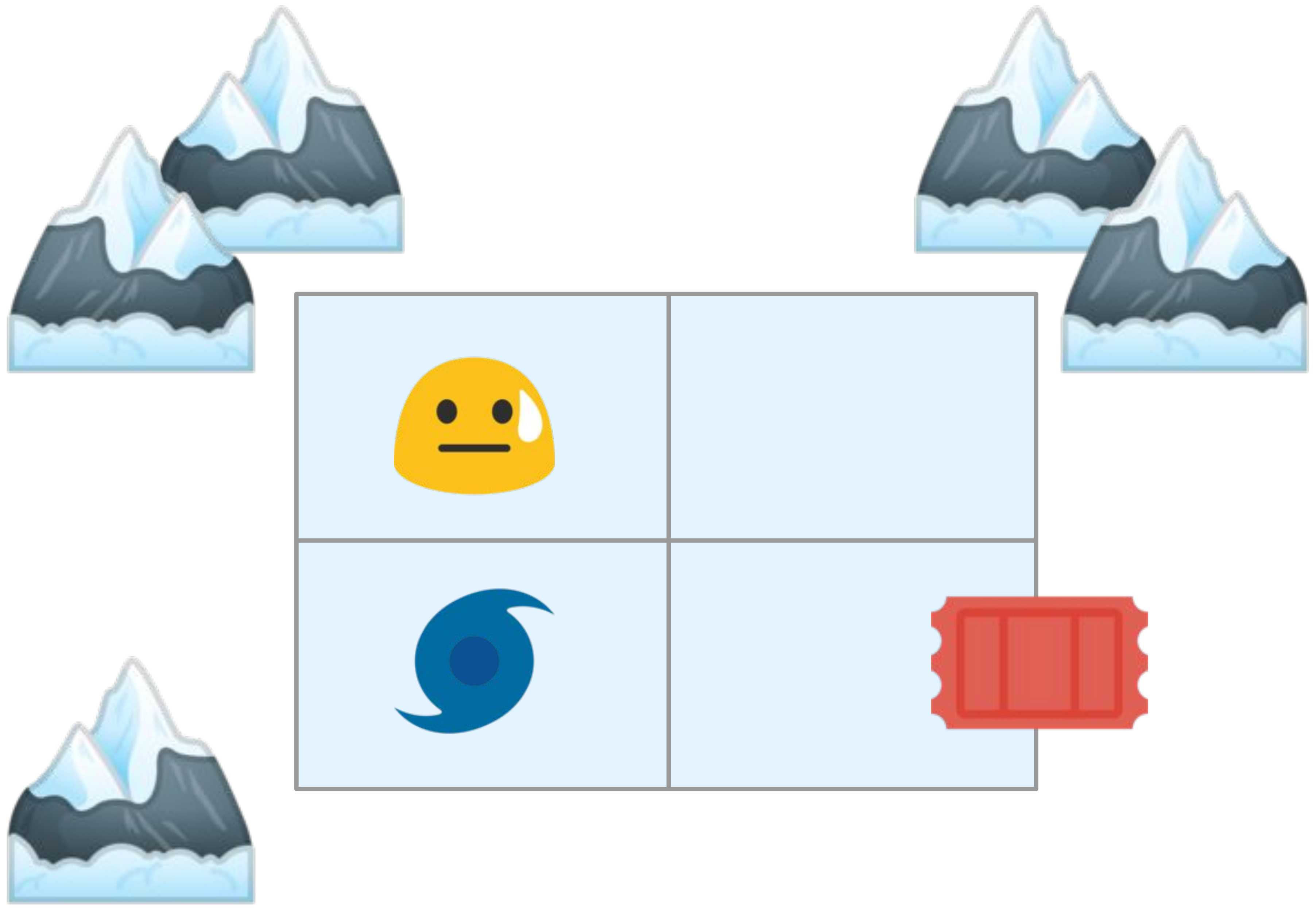


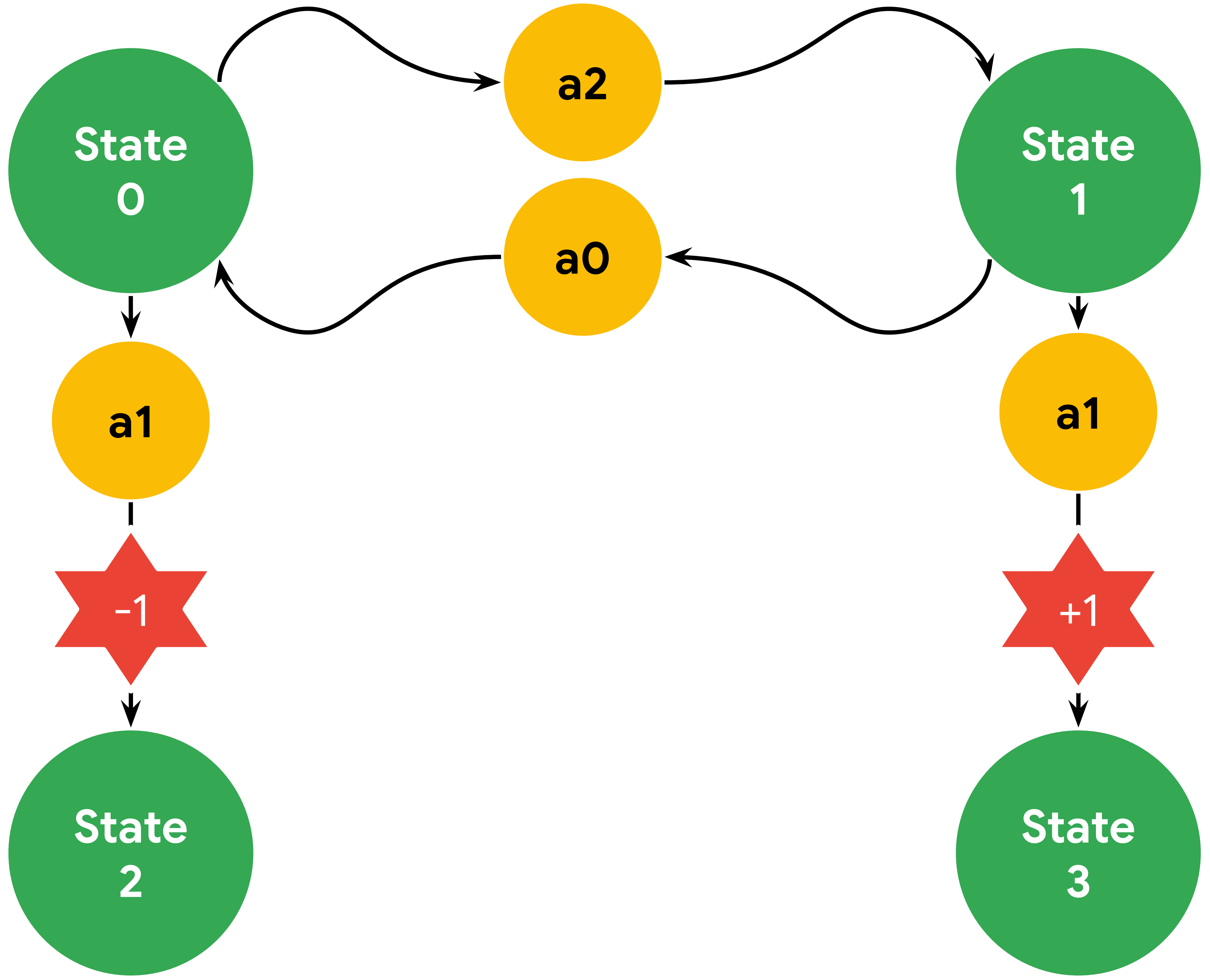
Bellman Equation

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

— = new addition

Simple Lake Value





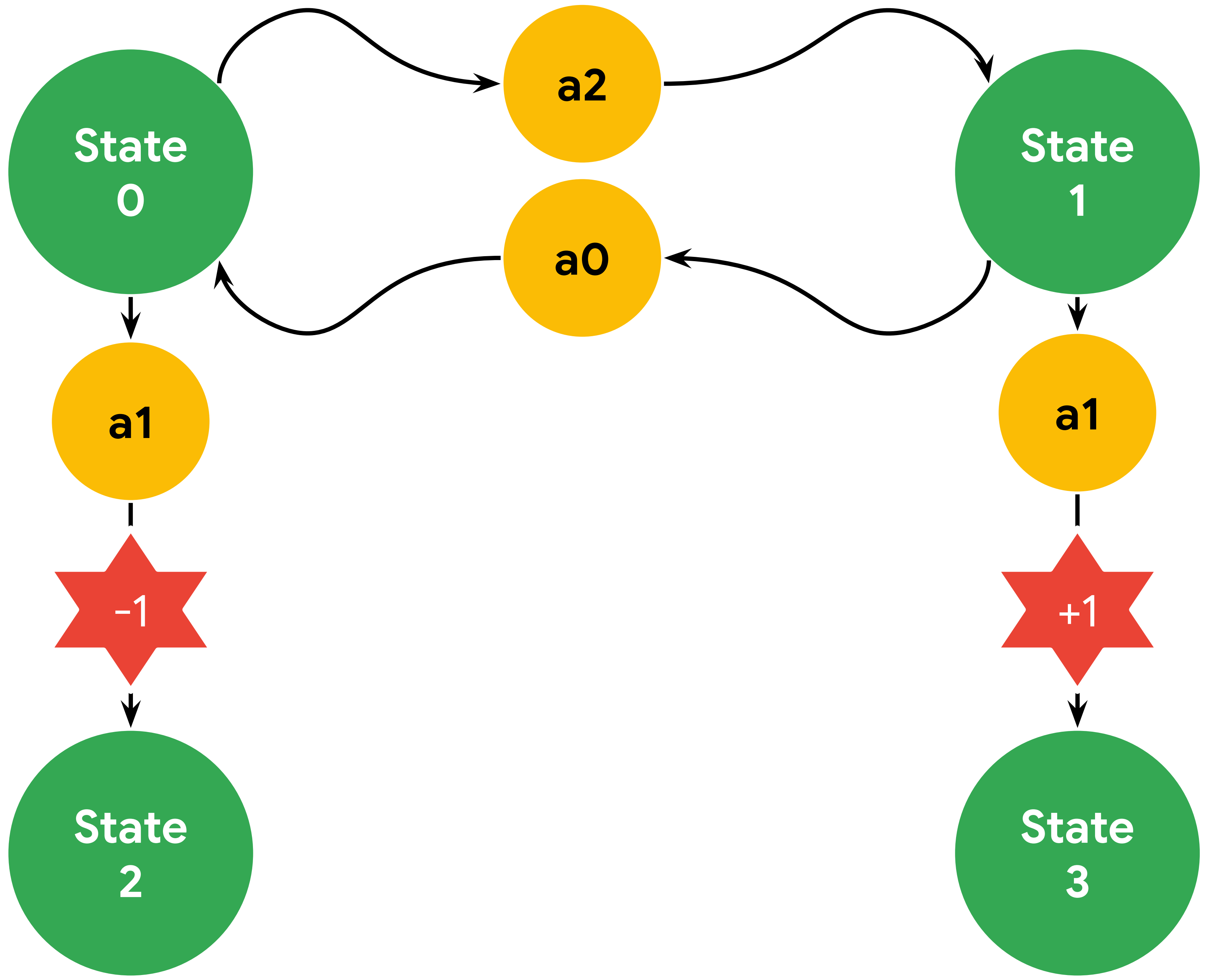
Simple Lake Value

State Map	
0	1
2	3

Current Value	
0	0
0	0

Policy Map	
?	?
-	-

Prime Value	
0	0
0	0



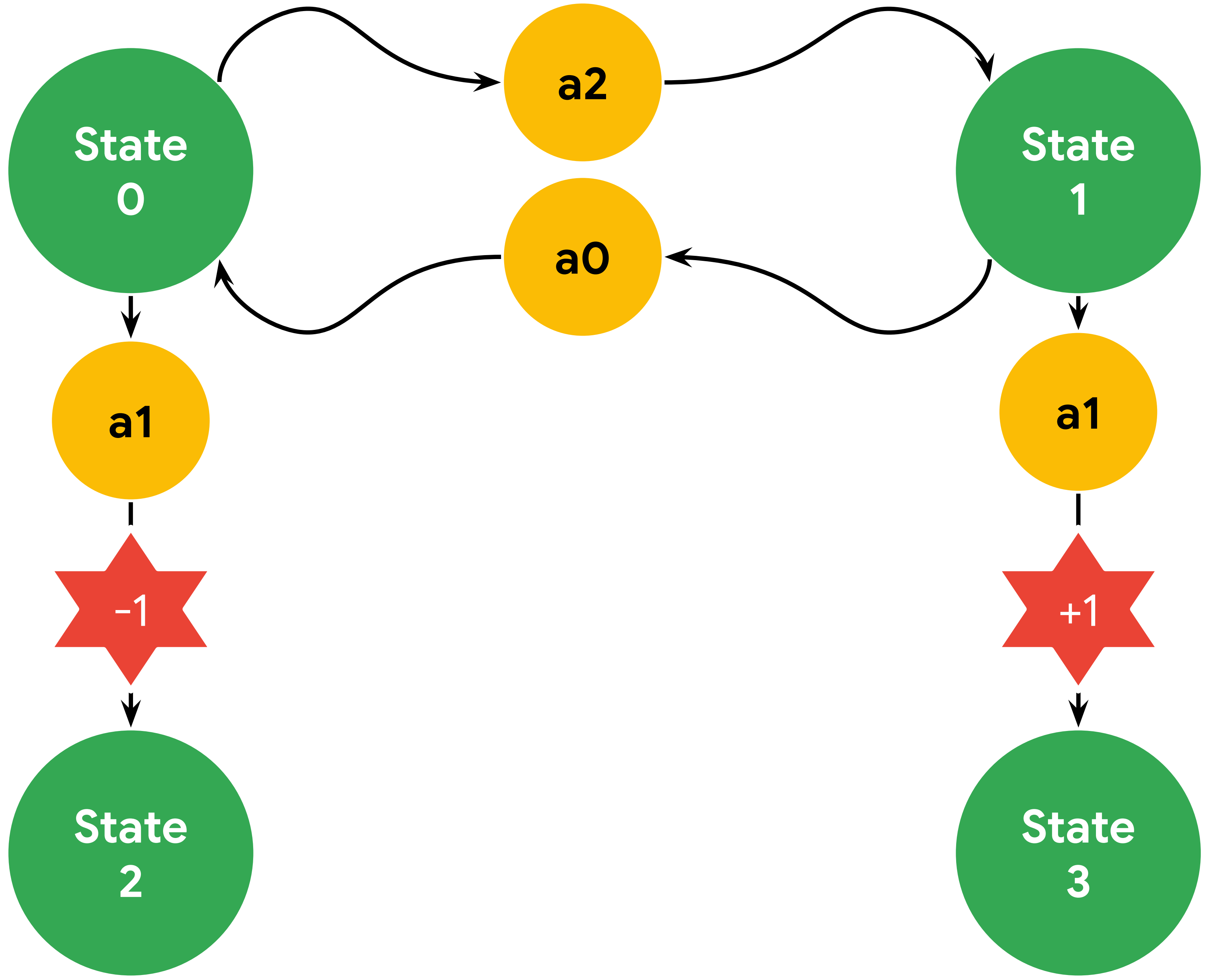
Simple Lake Value

State Map	
0	1
2	3

Current Value	
0	0
0	0

Policy Map	
?	?
-	-

Prime Value	
0	0
0	0



Google Cloud

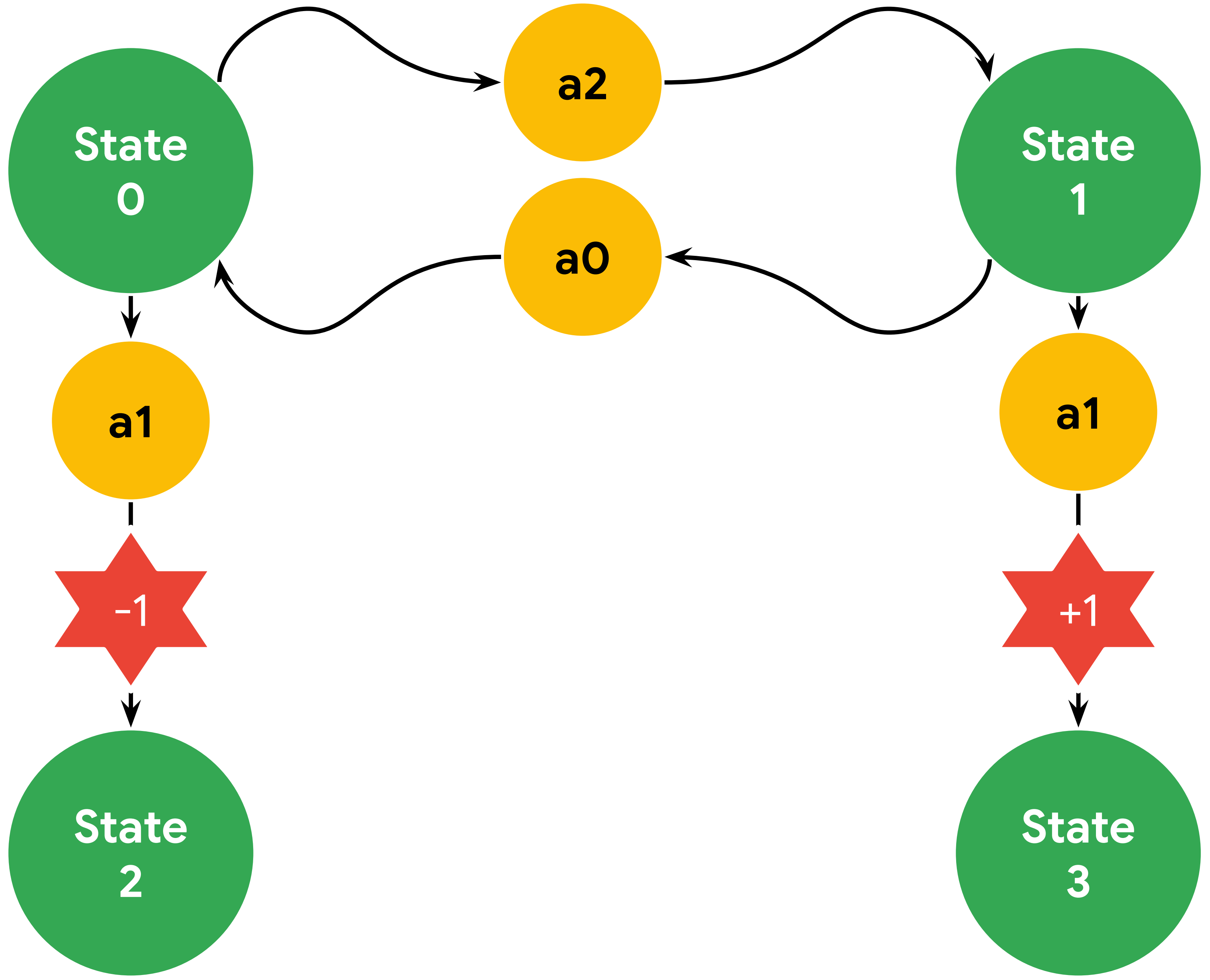
Simple Lake Value

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



Google Cloud

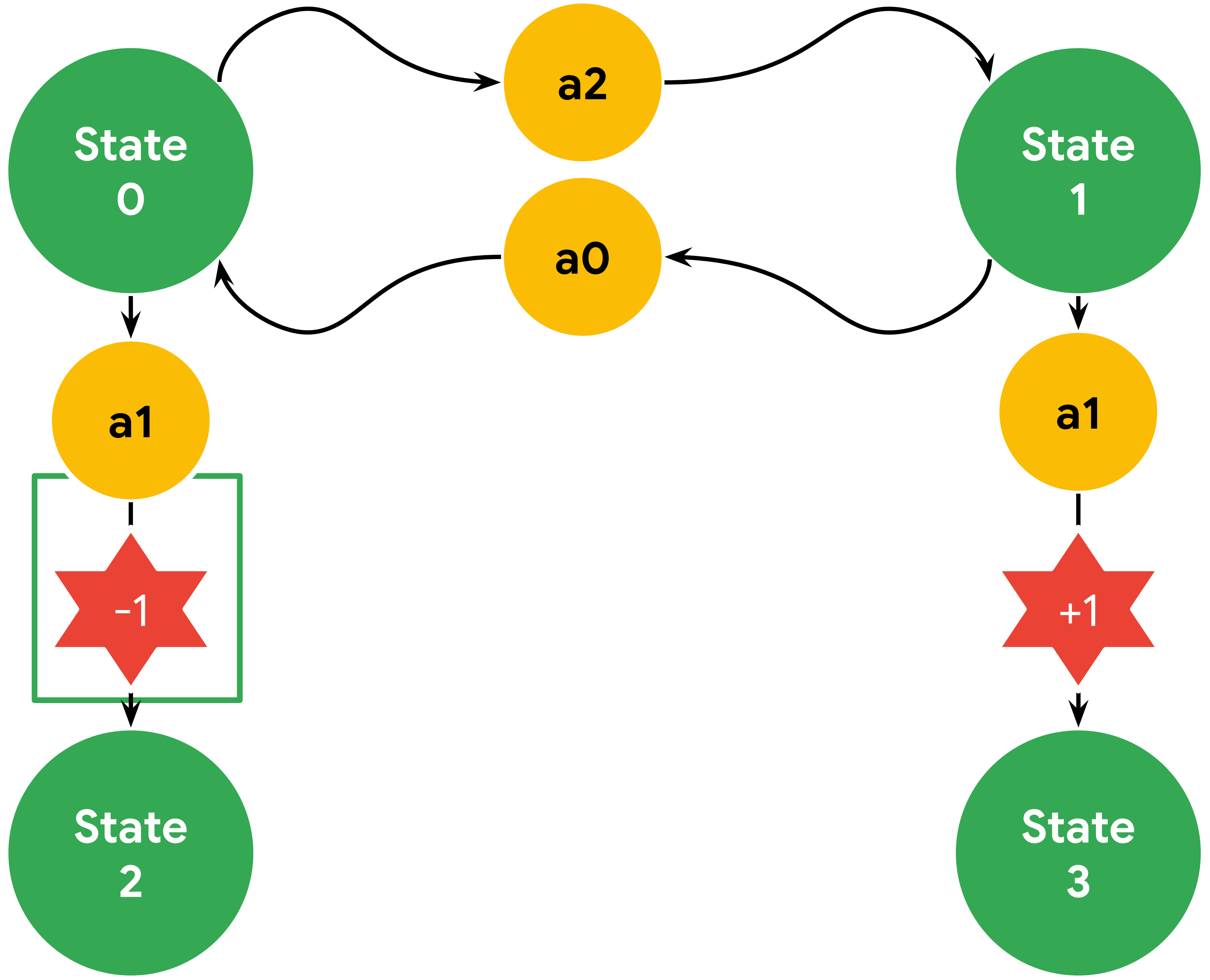
Simple Lake Value

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



Google Cloud

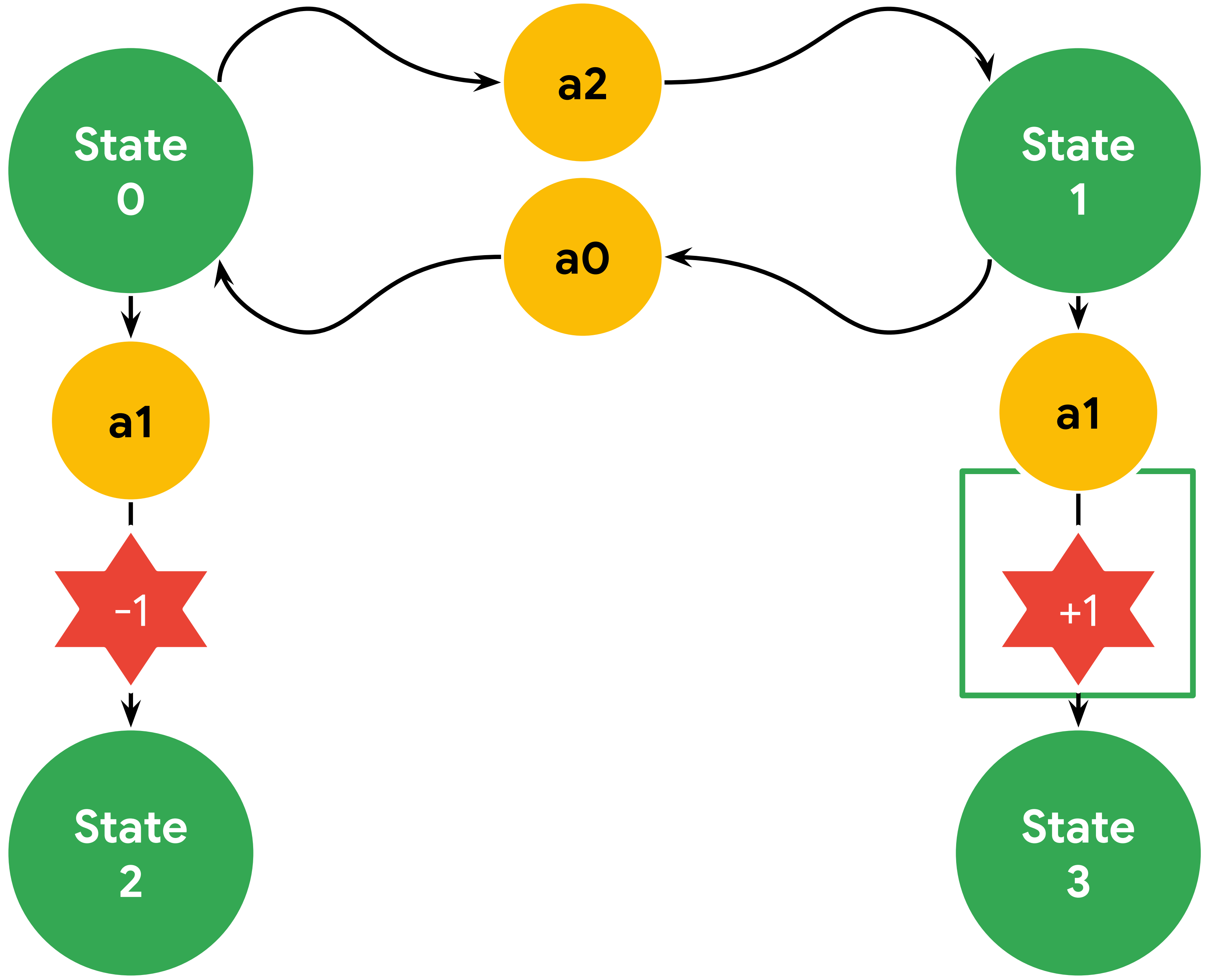
Simple Lake Value

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

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

Policy Map	
a2	**a1**
-	-

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



Google Cloud

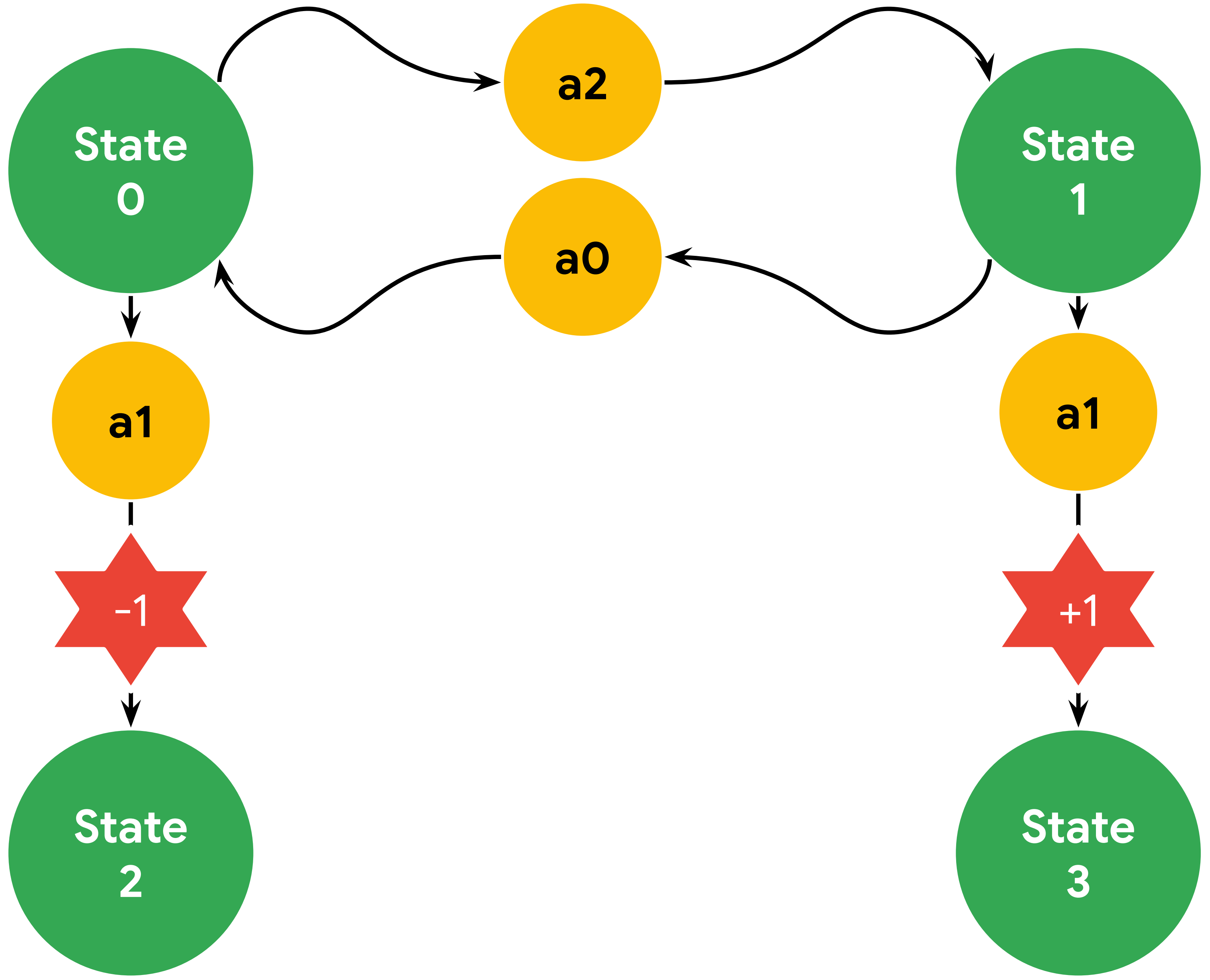
Simple Lake Value

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>



Google Cloud

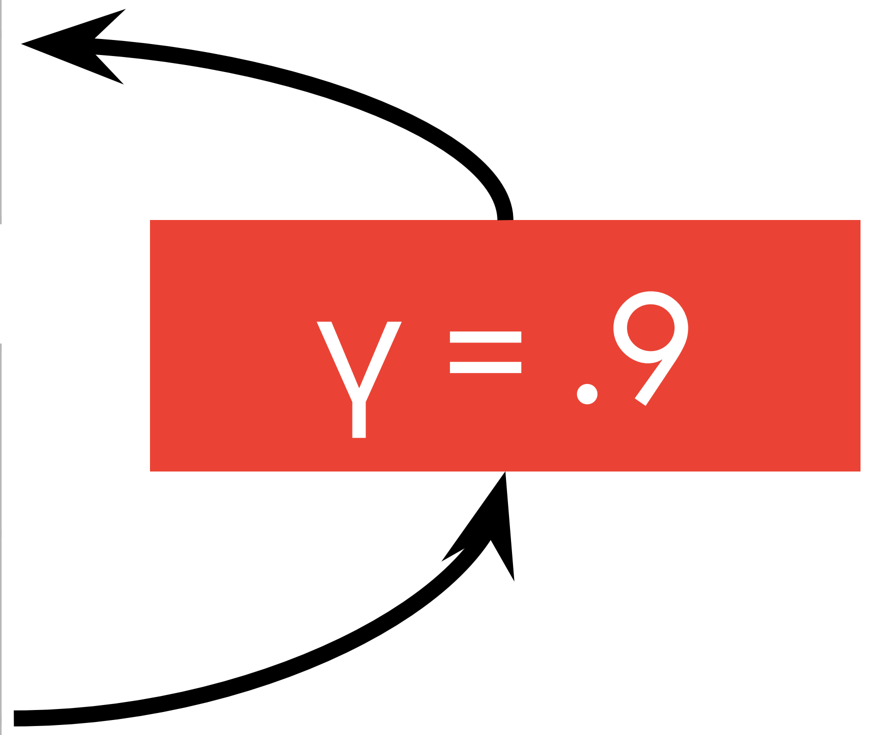
Simple Lake Value

State Map	
0	1
2	3

Current Value	
0	0
0	0

Policy Map	
a2	a1
-	-

Prime Value	
0	1
0	0



Google Cloud

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

$\times$ $\gamma = .9$ $+$

Rewards	
0	1
0	0

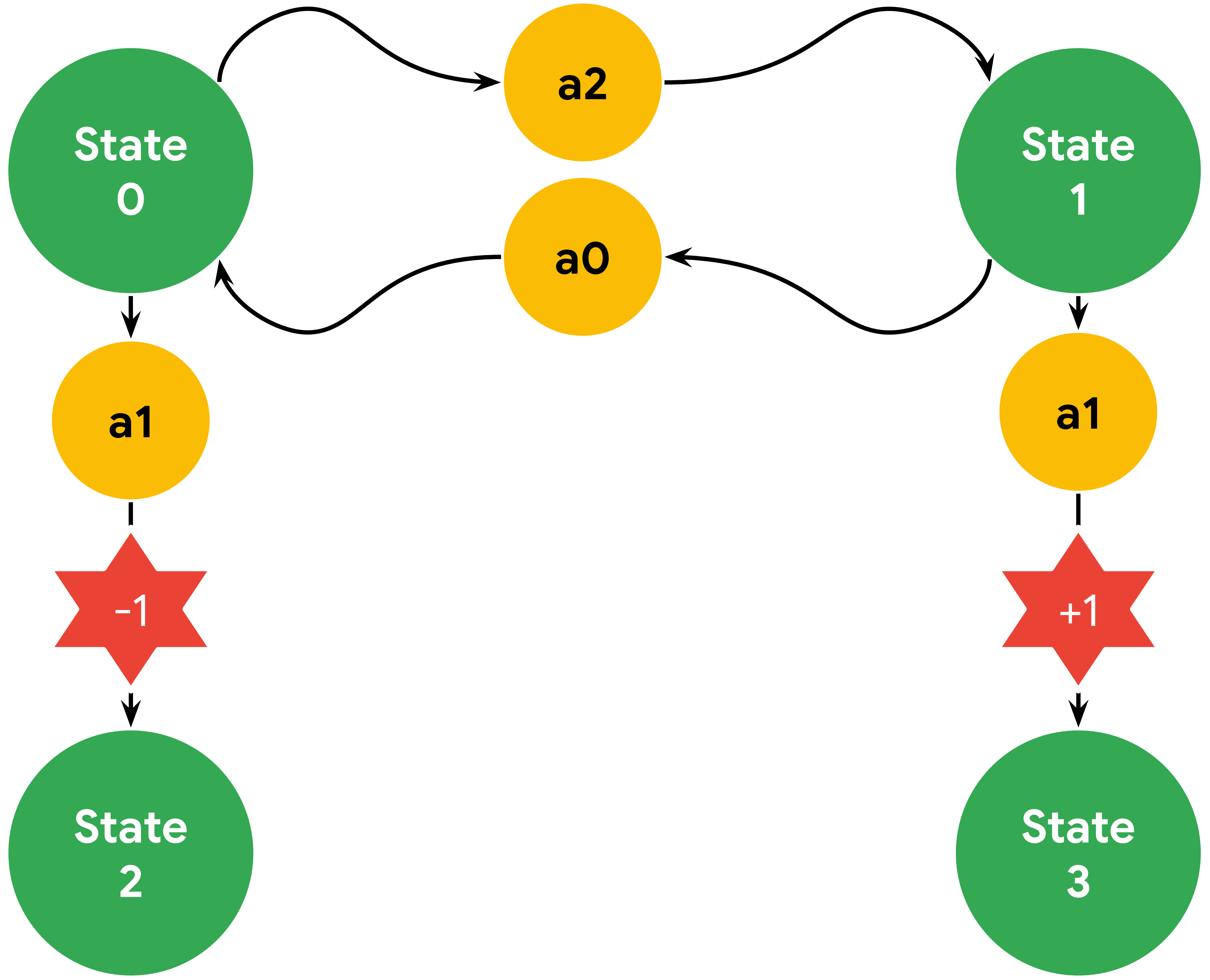
Simple Lake Value (1 iteration)

State Map	
0	1
2	3

Current Value	
0	.9
0	0

Policy Map	
a2	a1
-	-

Prime Value	
0	0
0	0



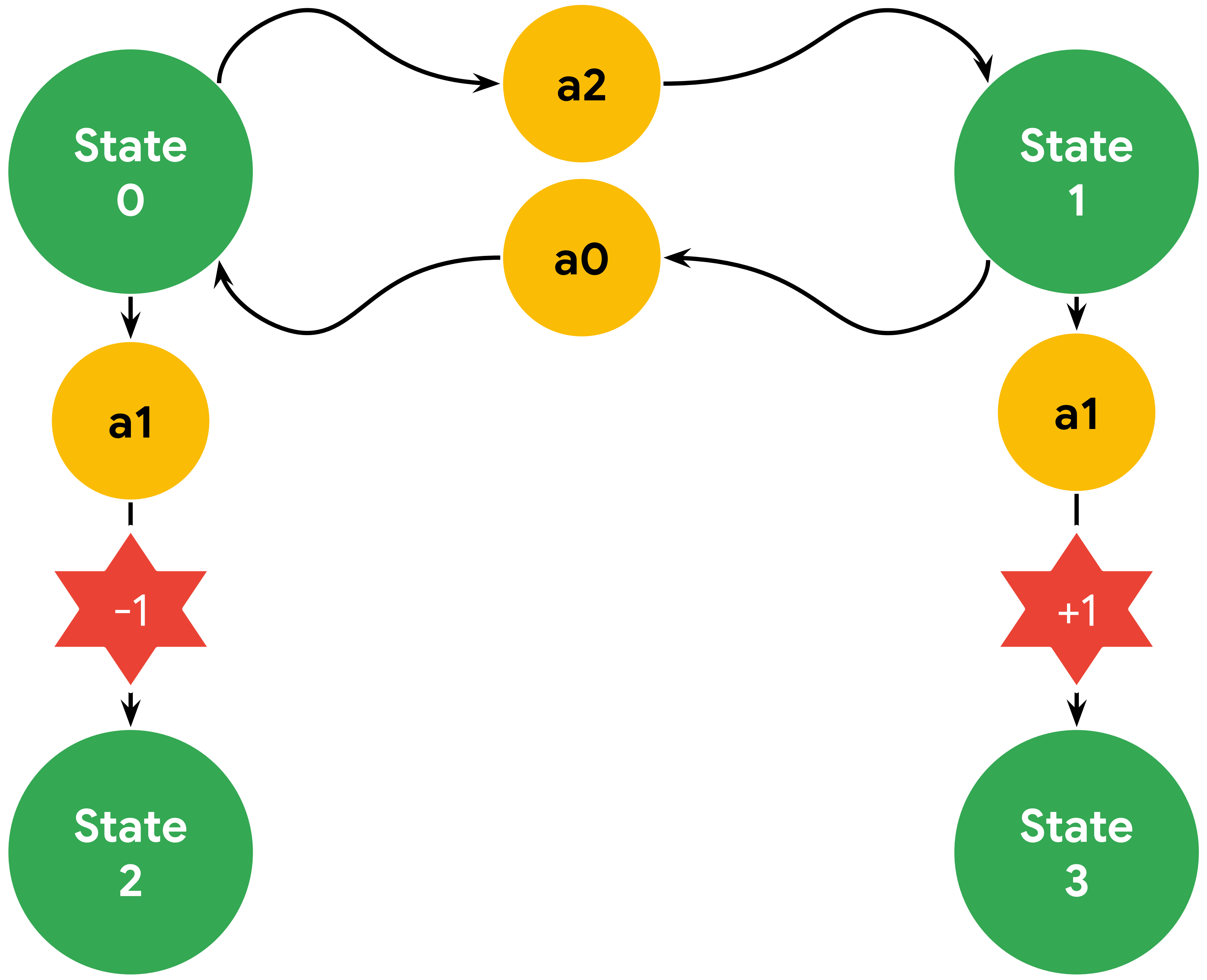
Simple Lake Value (2 and 3 iterations)

State Map	
0	1
2	3

Current Value	
.81	.9
0	0

Policy Map	
a2	a1
-	-

Prime Value	
0	0
0	0



Google Cloud

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))
```

Google Cloud

Value Iteration Code

```python
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
```

Google Cloud

Value Iteration Code

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

Iteration 6			
.53	.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

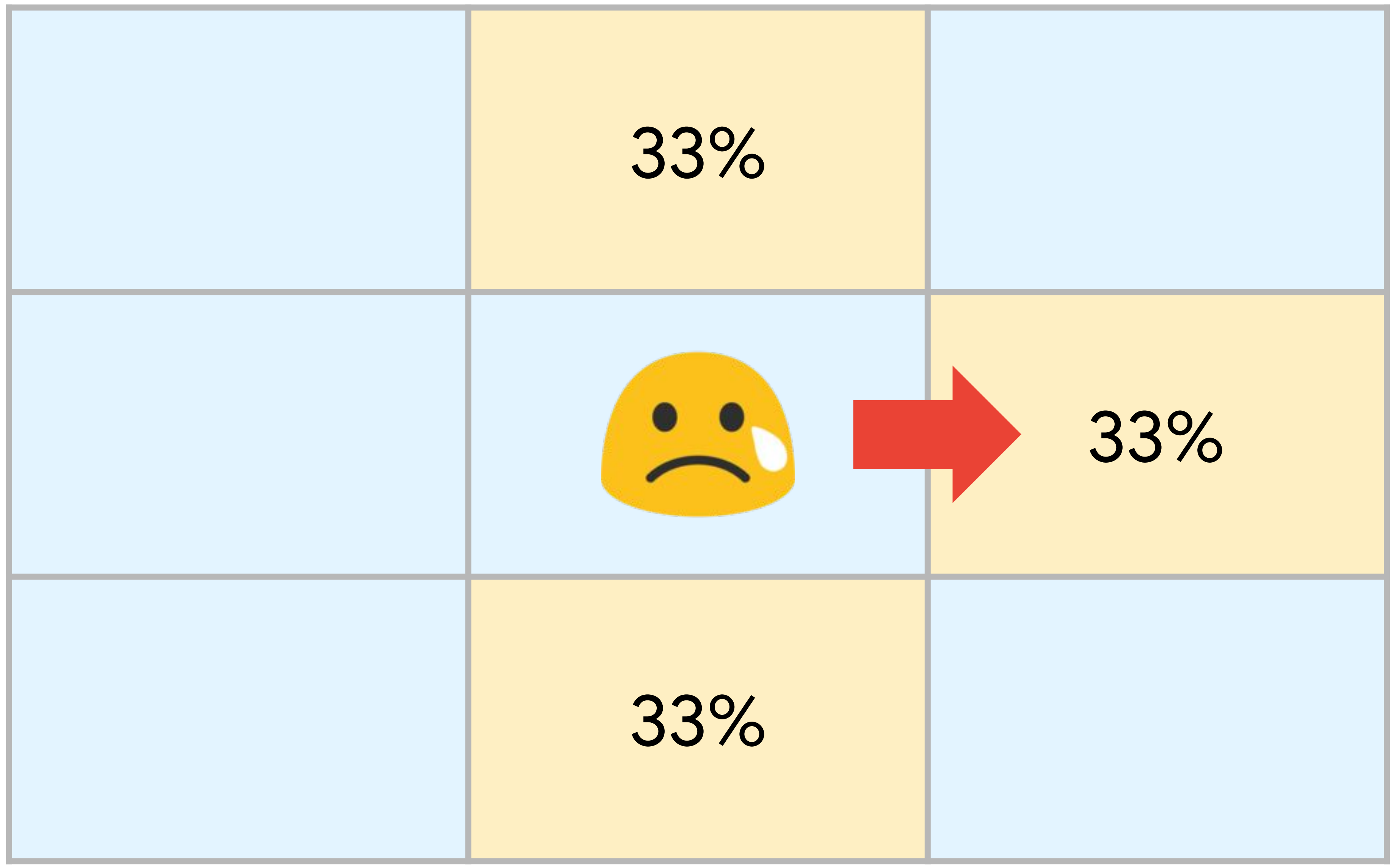
Value Iteration

Policy Iteration

TD(Lambda)

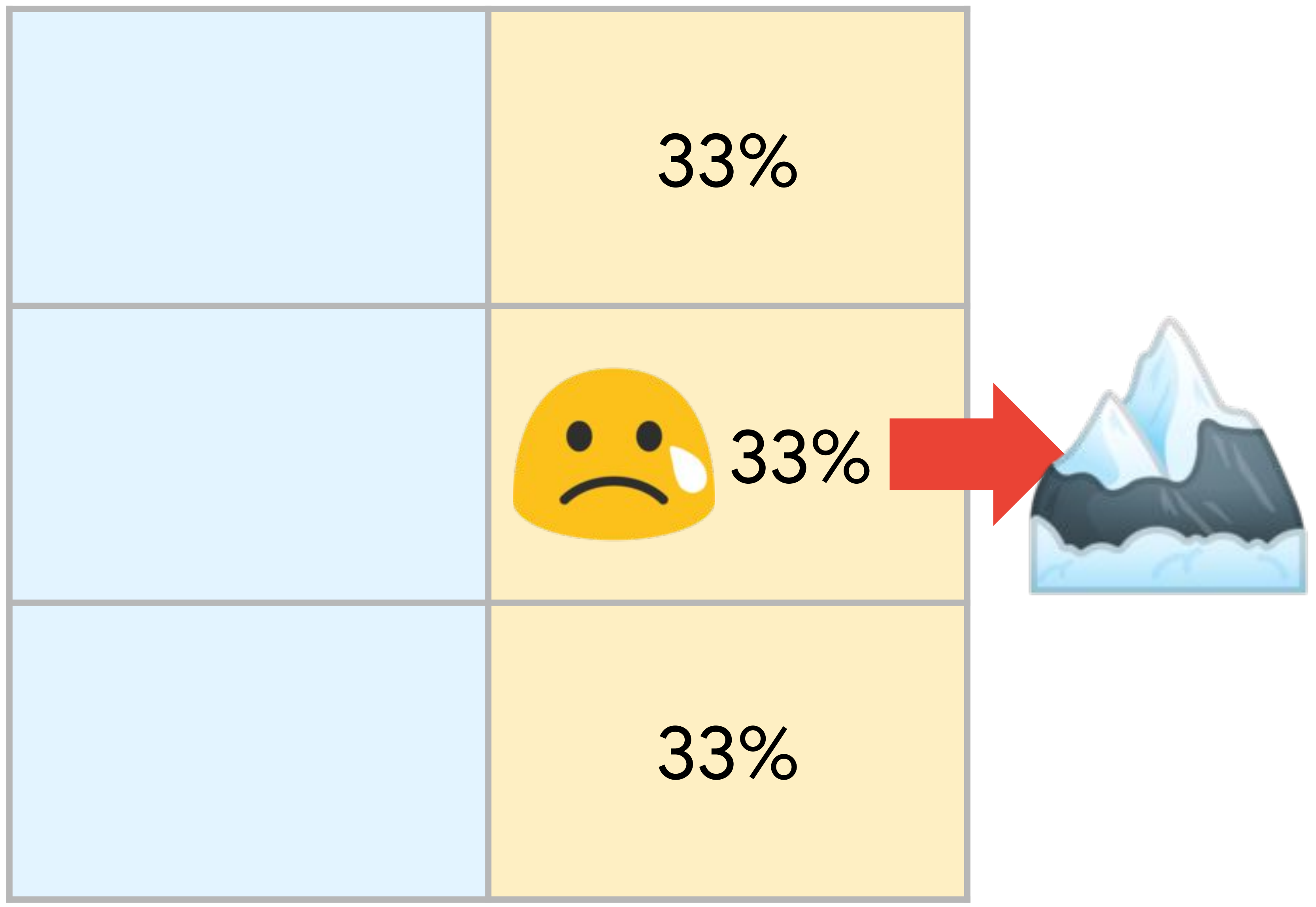
Q-Learning

Probabilities and Slipping

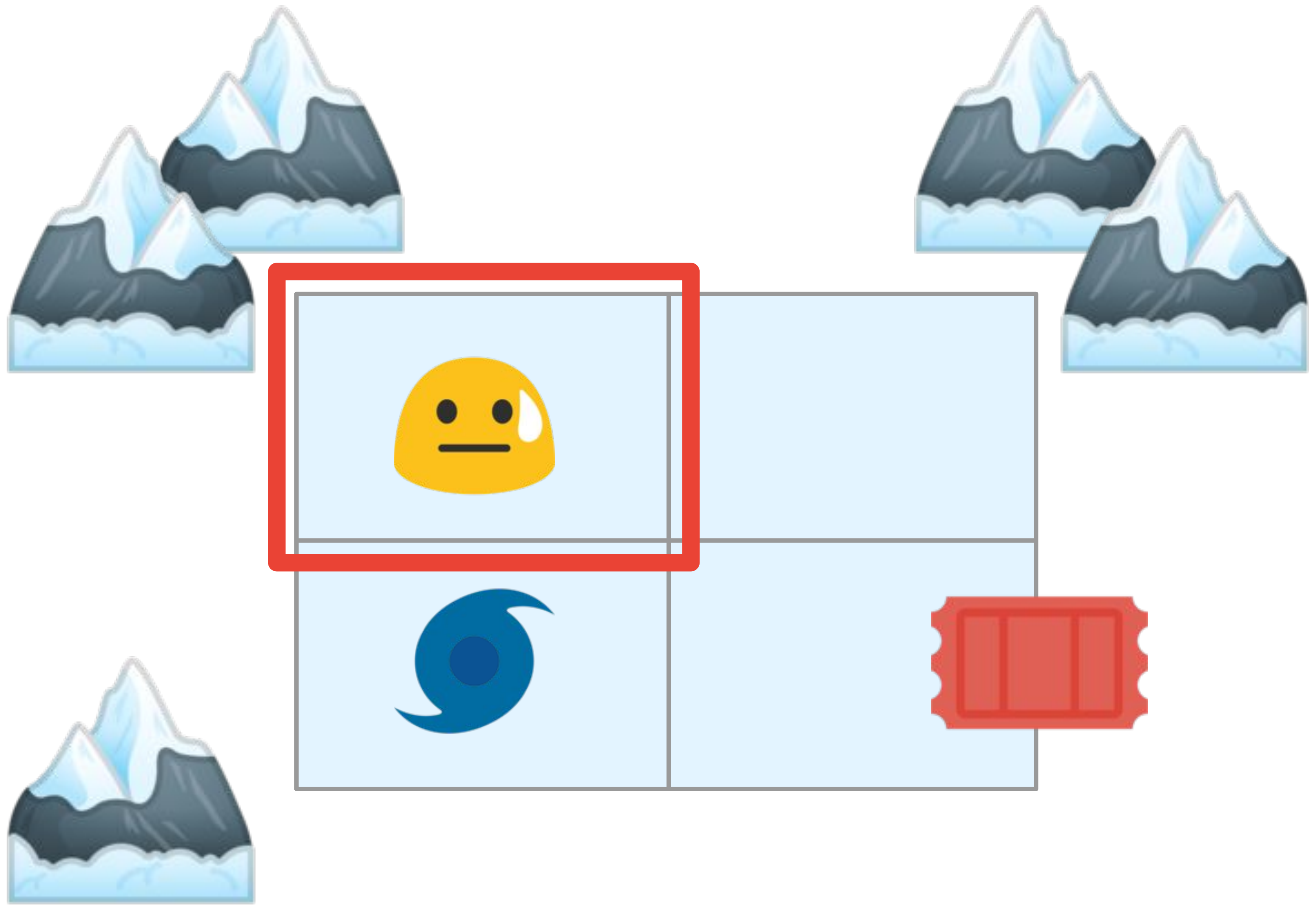


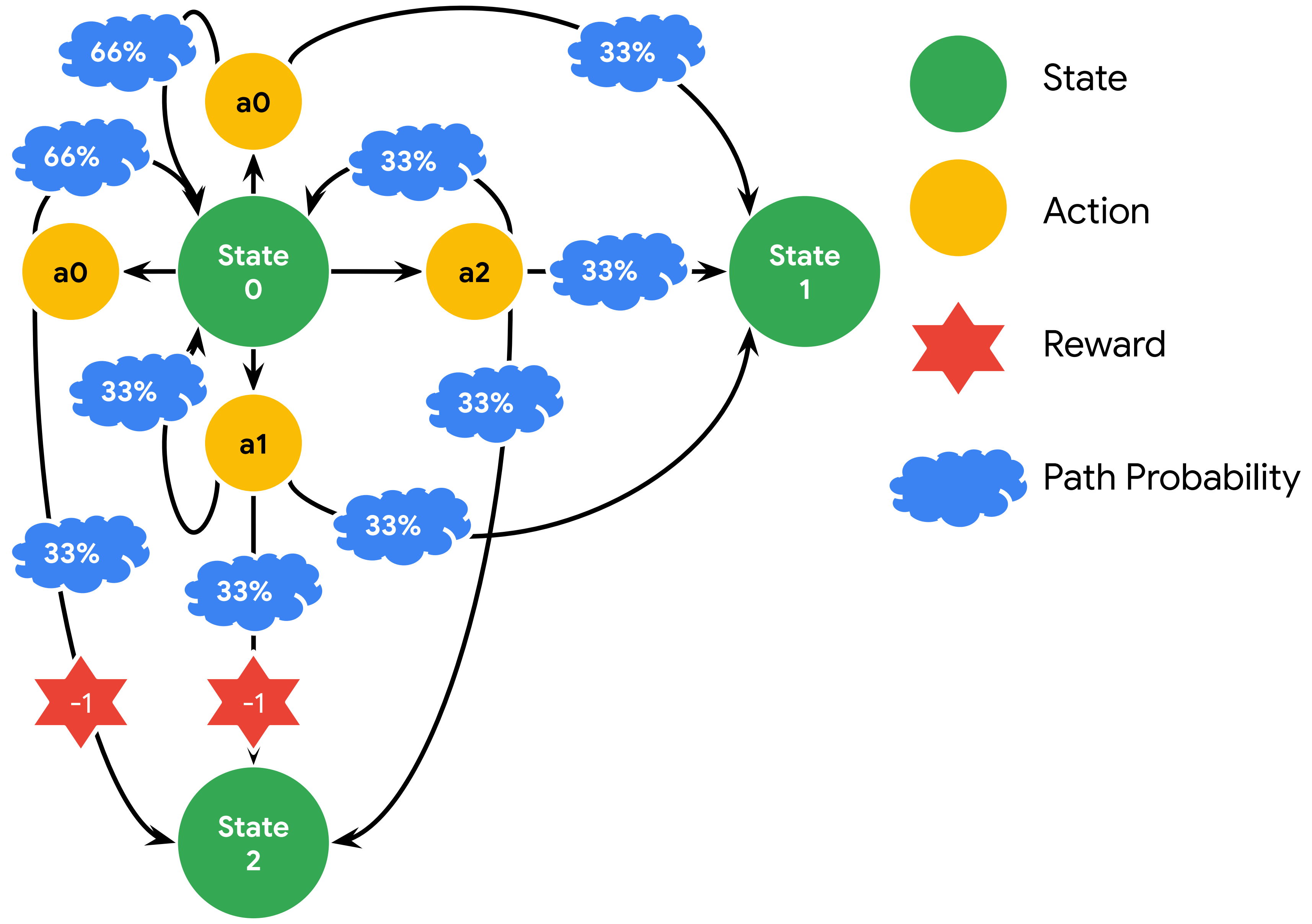
Google Cloud

Probabilities and Slipping



Slippery Simple Lake





Google Cloud

Bellman Equation

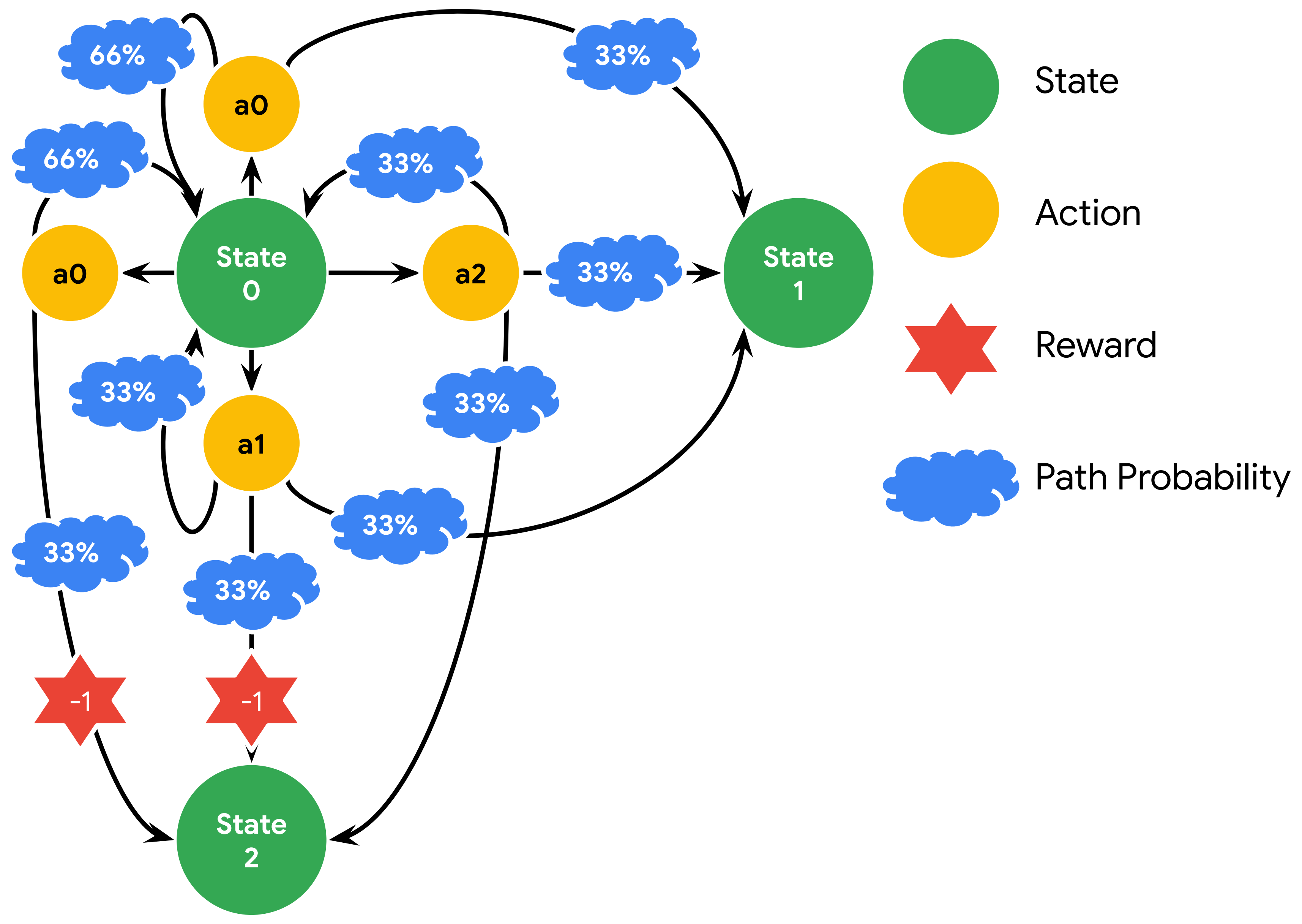
$$V^{\pi^*}(s) = max_a \left\{ R(s,a) + \gamma \underline{\sum_{s'} P(s'|s,a)} V^{\pi^*}(s') \right\}$$

$\underline{\qquad\qquad}$ = 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



Google Cloud

Weighting State Prime

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

.33 · V(Counter Clockwise) +
.33 · V(Forward) +
.33 · V(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

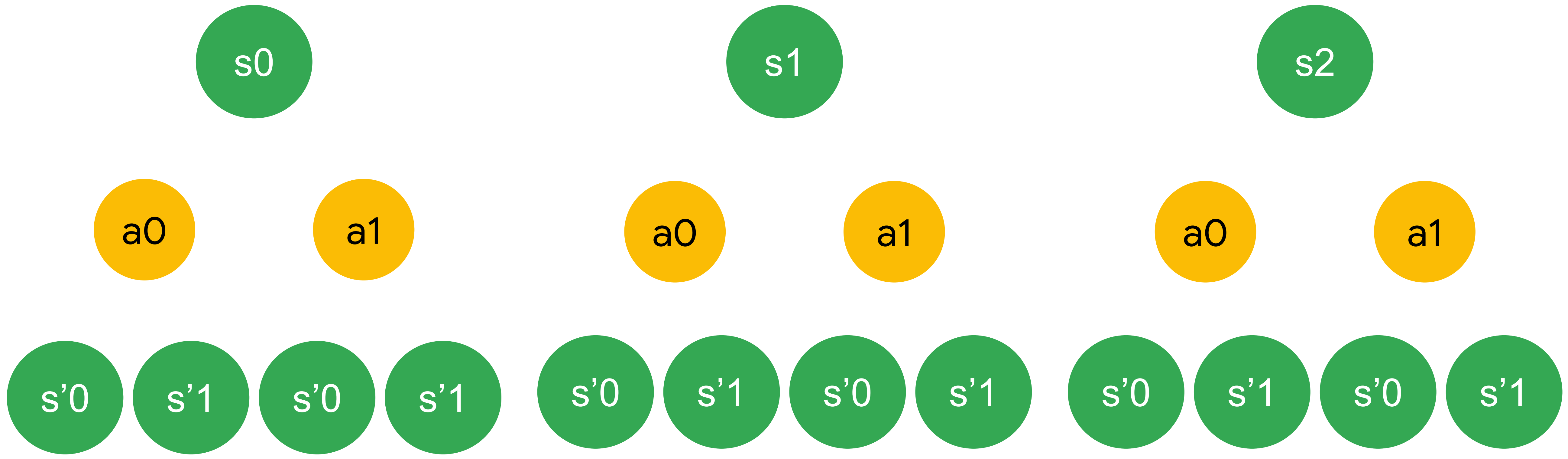
Value Iteration Complexity

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

For each state ...

Compare each action ...

By weighting each new state ...



Repeat up to the total number of states.

Google Cloud

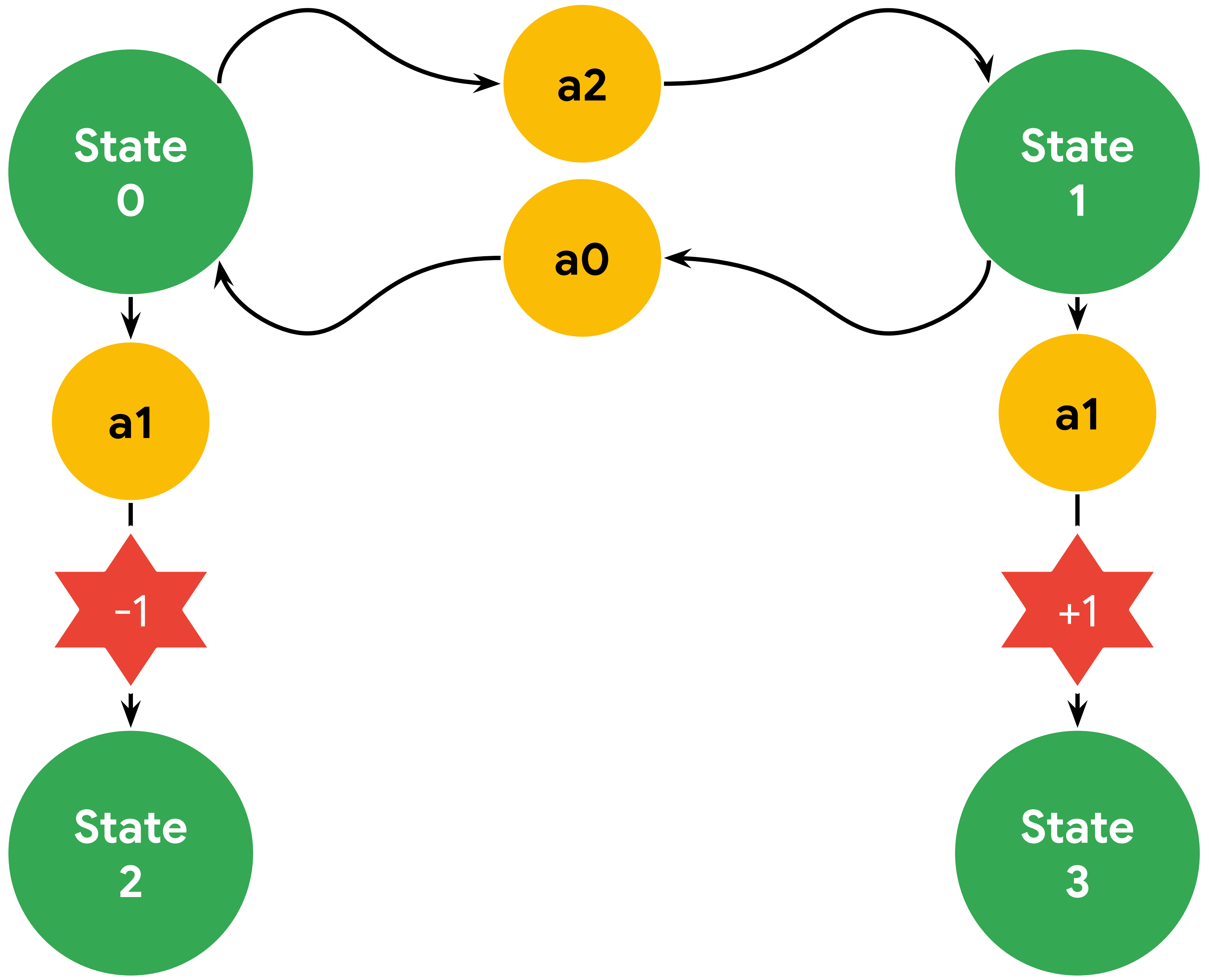
Policy Iteration

State Map	
0	1
2	3

Current Value	
0	0
0	0

Policy Map	
1	1
-	-

Prime Value	
0	0
0	0



Google Cloud

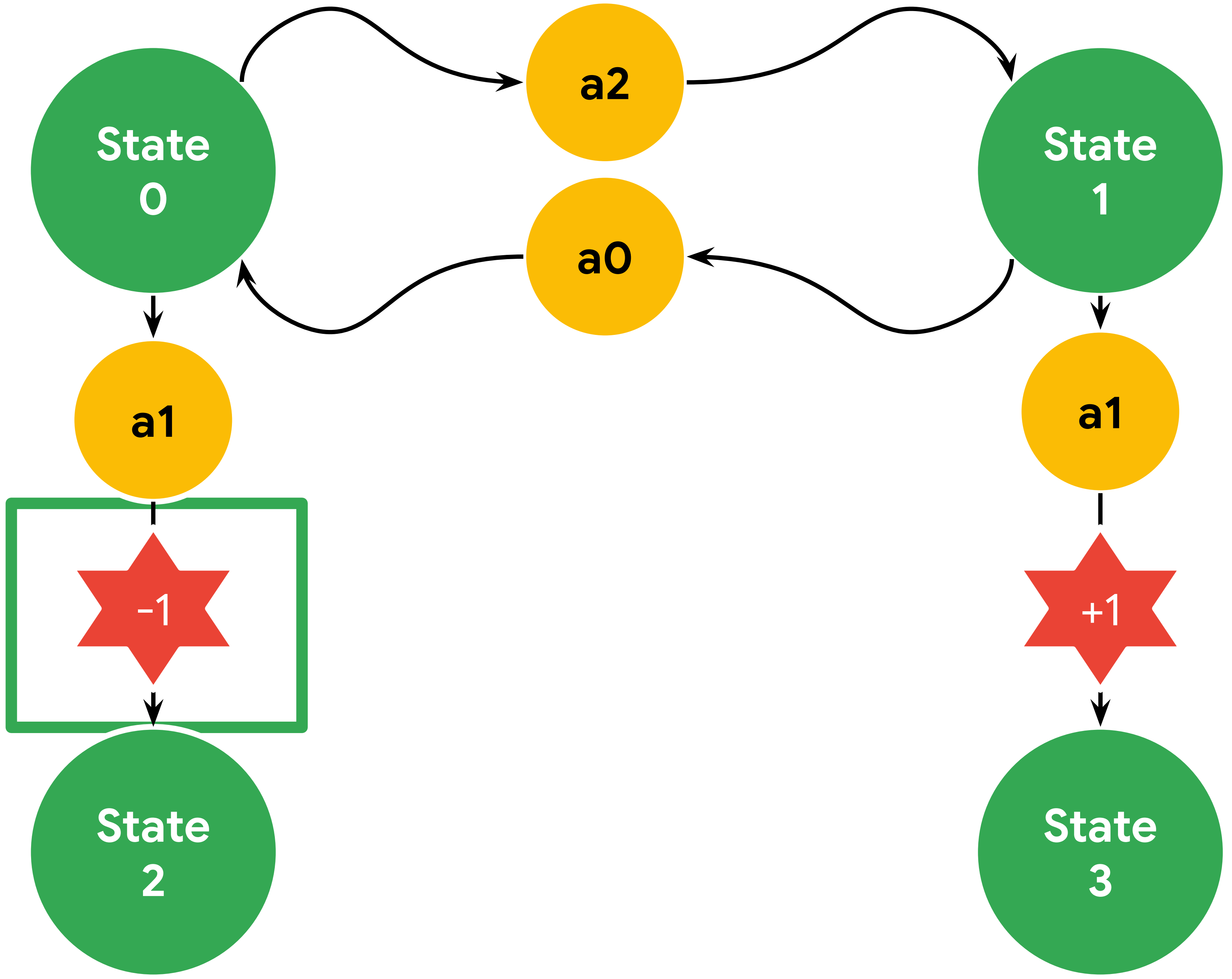
Policy Iteration

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



Google Cloud

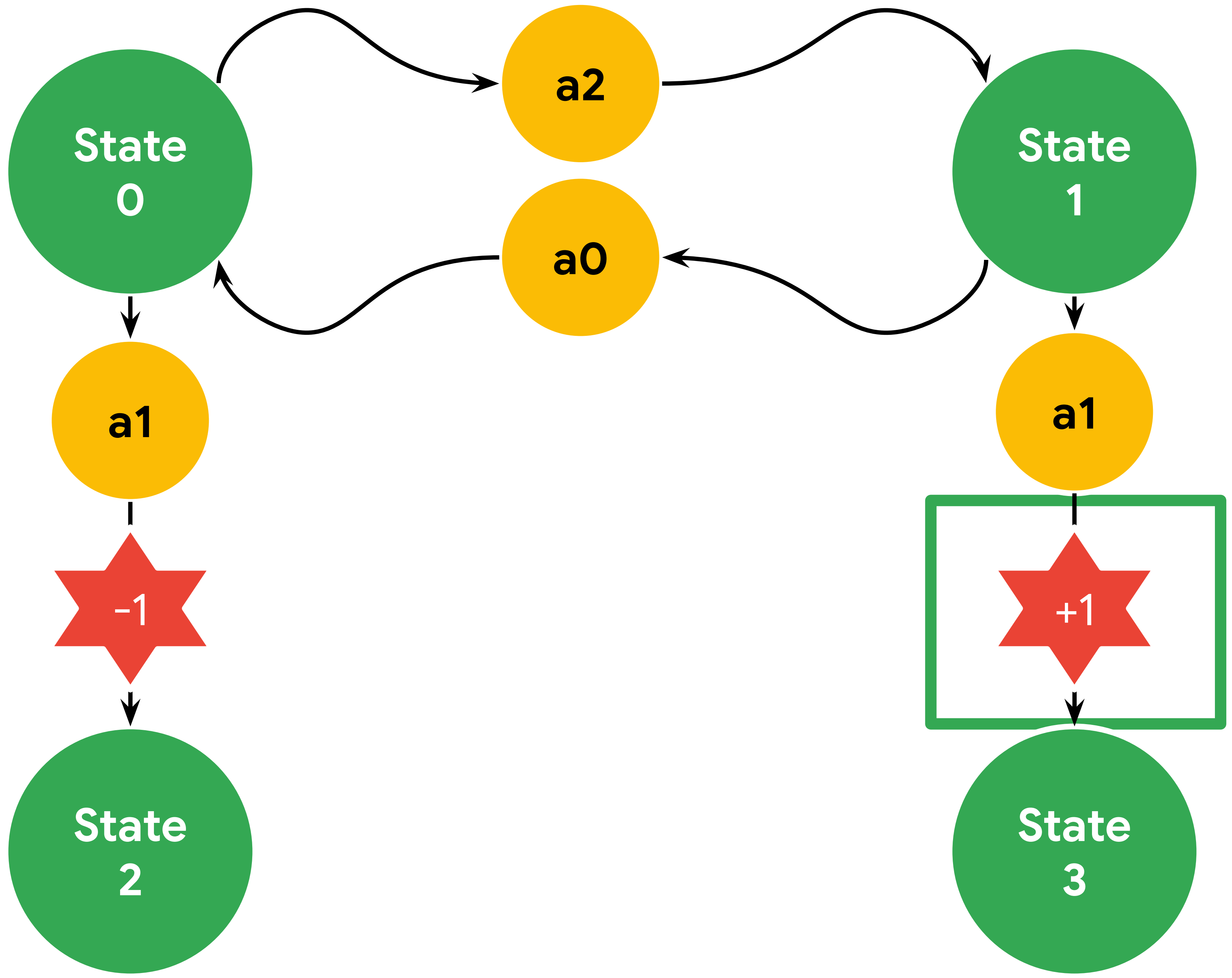
Policy Iteration

State Map	
0	1
2	3

Current Value	
0	0
0	0

Policy Map	
1	1
-	-

Prime Value	
-1	1
0	0



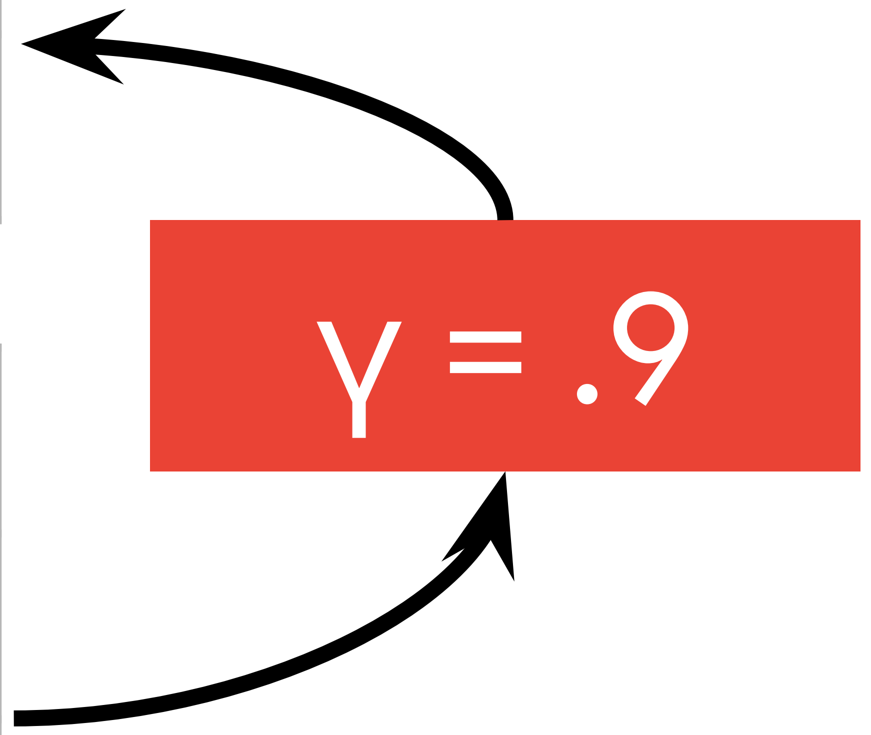
Policy Iteration

State Map	
0	1
2	3

Current Value	
0	0
0	0

Policy Map	
1	1
-	-

Prime Value	
-1	1
0	0



Google Cloud

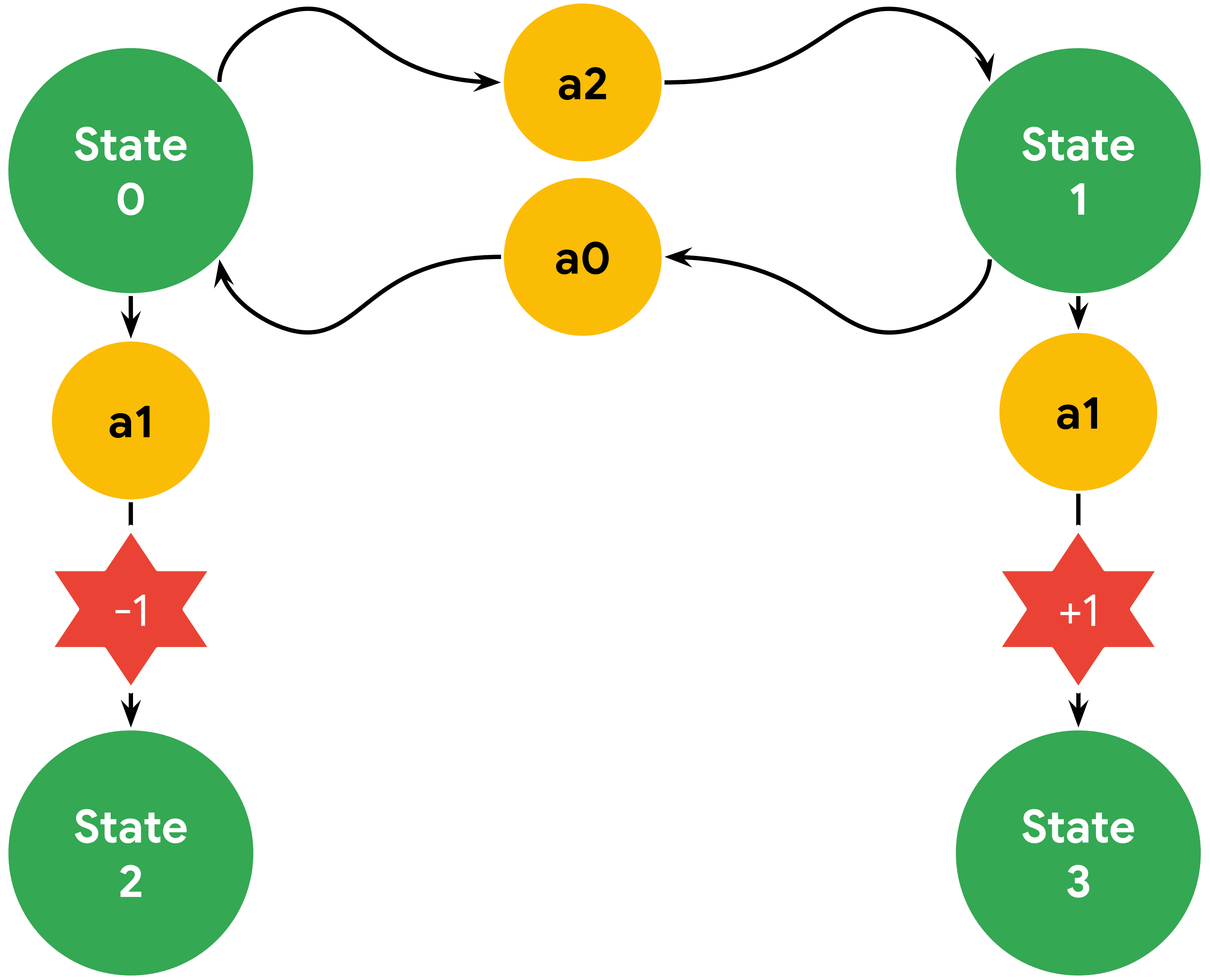
Policy Iteration

State Map	
0	1
2	3

Current Value	
-.9	.9
0	0

Policy Map	
1	1
-	-

Prime Value	
0	0
0	0



Google Cloud

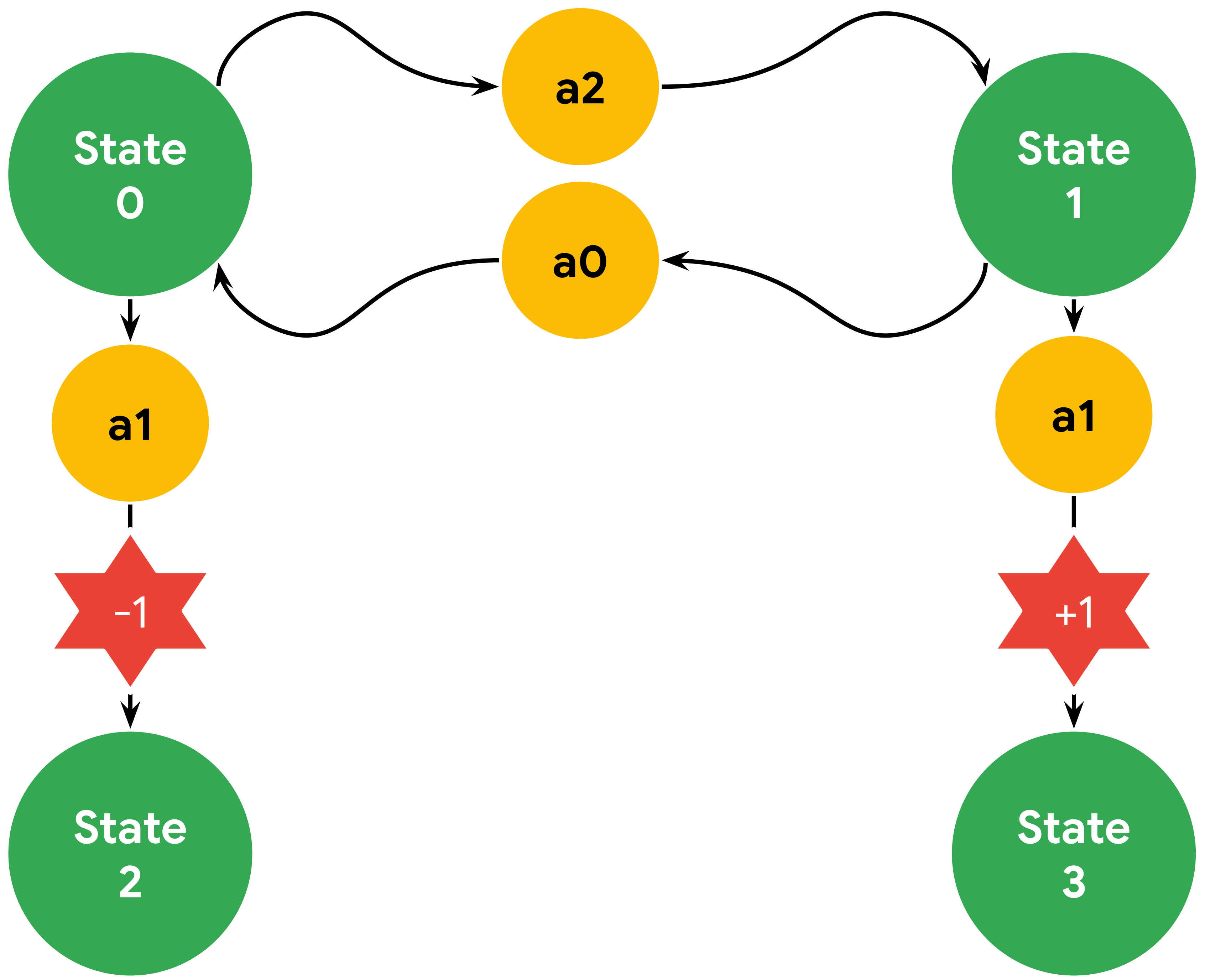
Policy Iteration

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



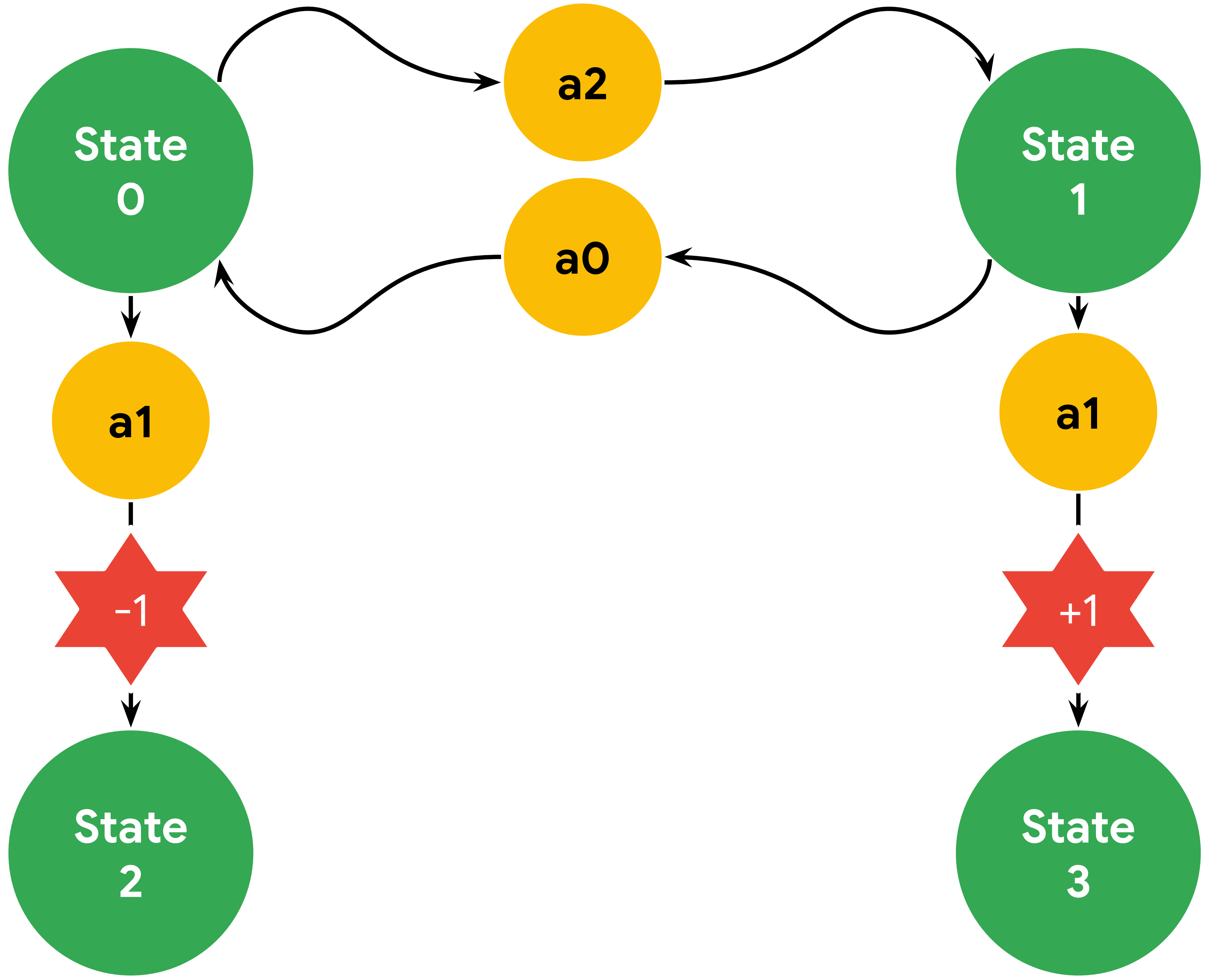
Policy Iteration

State Map	
0	1
2	3

Current Value	
-.9	.9
0	0

Policy Map	
2	**1**
-	-

Prime Value	
-1	1
0	0



Google Cloud

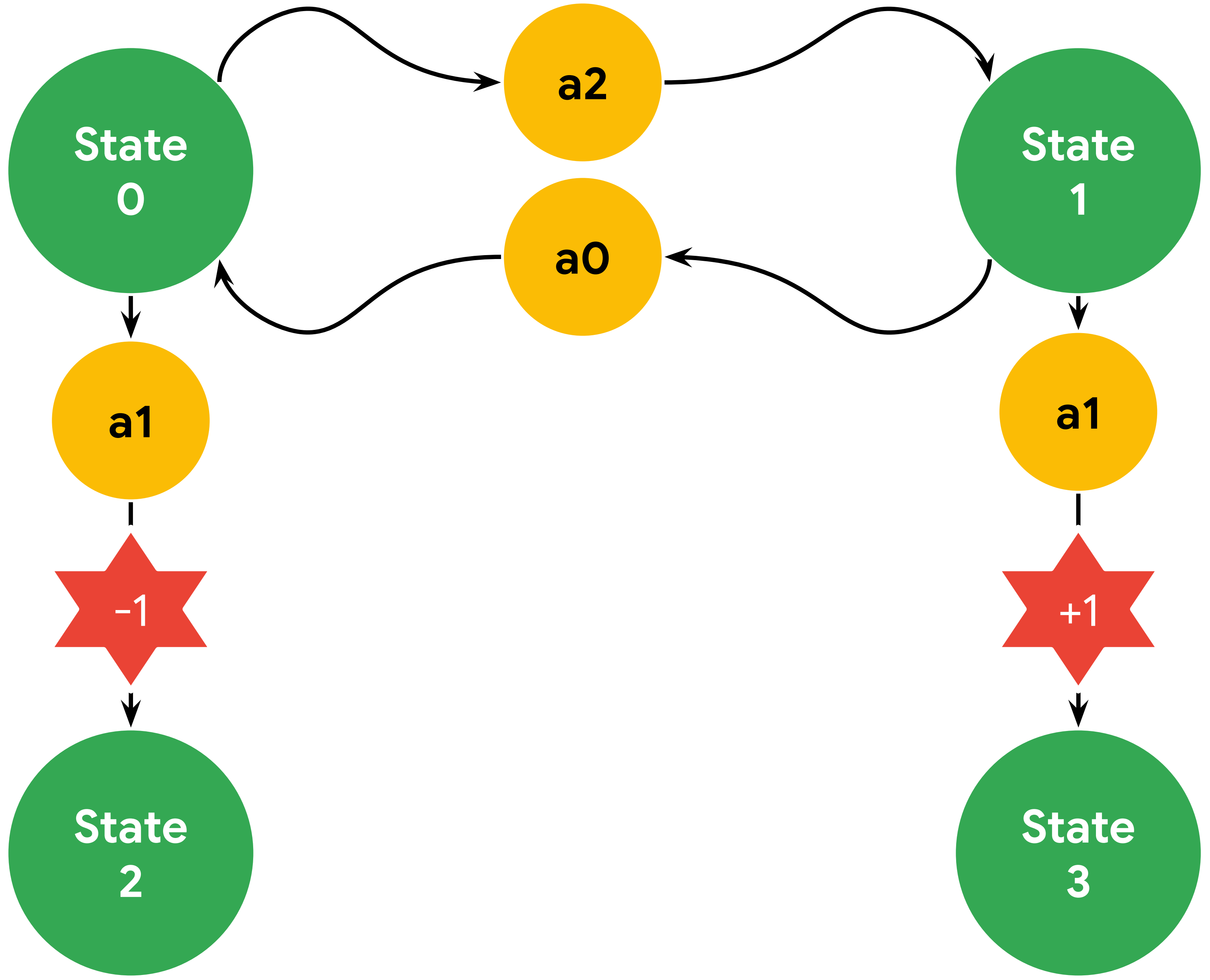
Policy Iteration (Iteration 2)

State Map	
0	1
2	3

Current Value	
.81	.9
0	0

Policy Map	
2	1
-	-

Prime Value	
0	0
0	0



Google Cloud

Modified Policy Iteration Code

```python
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
```

Google Cloud

Modified Policy Iteration Code

```python
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

```python
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
```

Google Cloud

Modified Policy Iteration Complexity

$$O(\underline{s^2 s'} + \underline{s^2 a s'})$$

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

Lake

0	0	0	0
0	-1	0	-1
0	0	0	-1
-1	0	0	1

Value Iteration

Iteration 7

.00	.00	.00	.00
.01	0	-.27	0
.03	.10	.10	0
0	.25	.52	0

Optimal Policy

0	3	3	3
0	-	0	-
3	1	0	-
-	2	1	-

Policy Iteration

Iteration 4

0	0	0	0
0	0	-.3	0
0	.03	.03	0
0	.14	.44	0

Optimal Policy

0	3	3	3
0	-	0	-
3	1	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

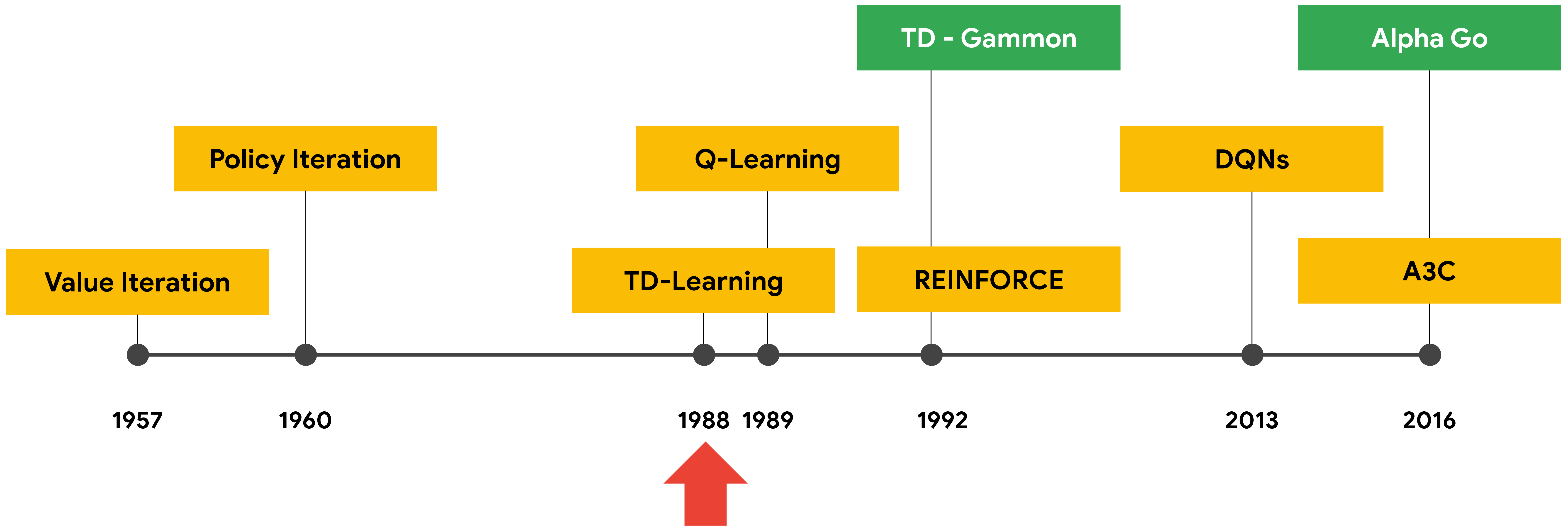
Value Iteration

Policy Iteration

TD(Lambda)

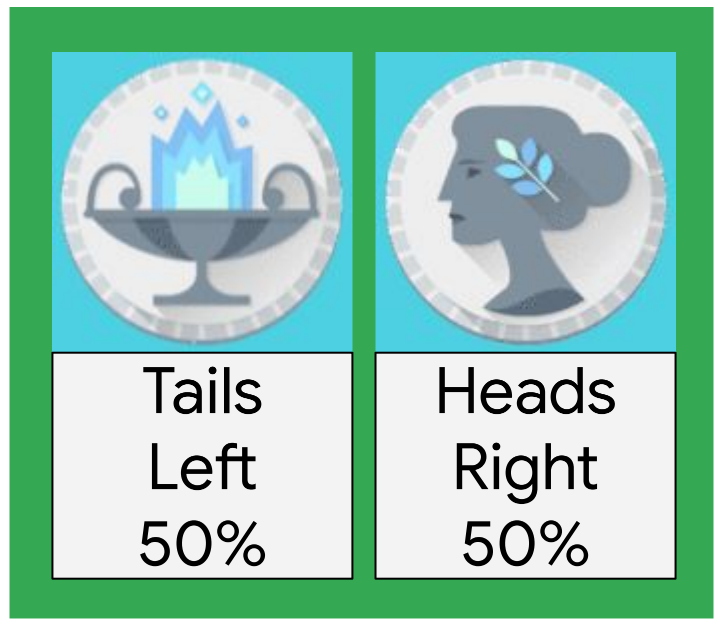
Q-Learning

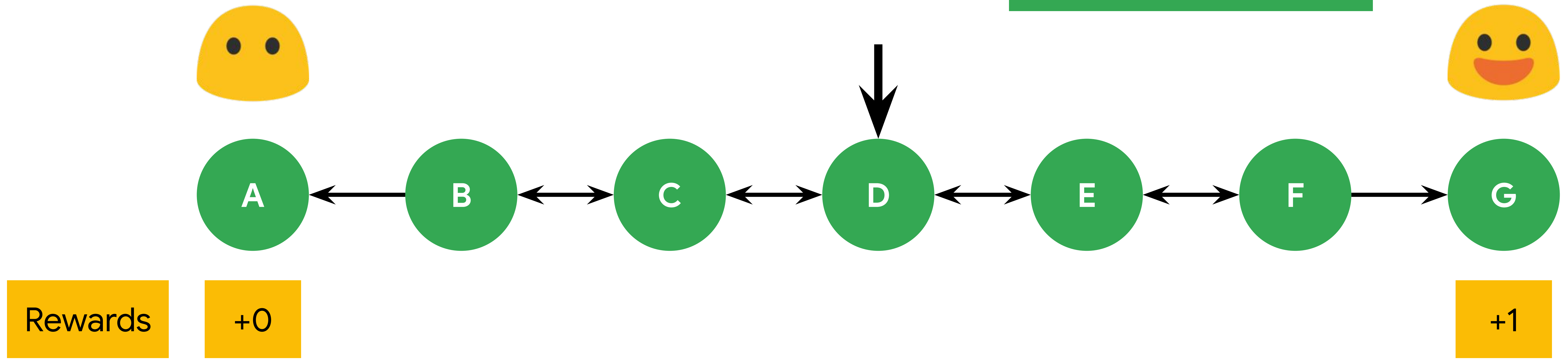
An RL Timeline



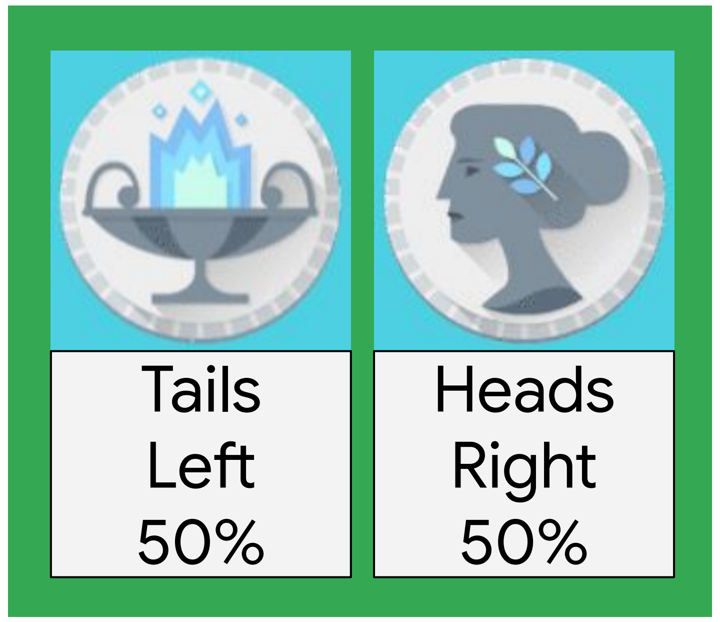
Google Cloud

A Random Walk

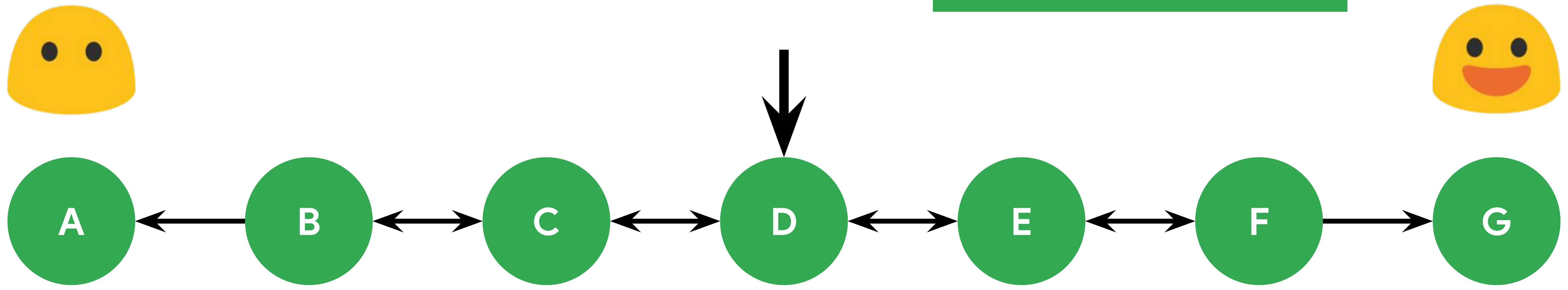




A Random Walk



$\gamma = 1$



	A	B	C	D	E	F	G
Rewards	+0						+1
Value Iteration	0	.17	.33	.5	.67	.83	0

Google Cloud

TD(0)

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

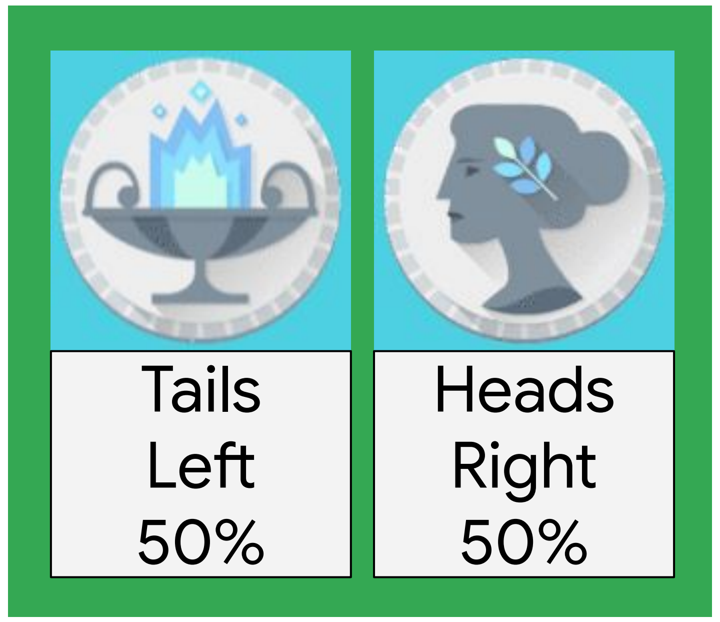
$$\underline{V(s_{t-1})} = \underline{V(s_{t-1})} + \underbrace{\alpha_t}_{\uparrow}(\underline{R(s_{t-1},a)} + \underline{\gamma V(s_t)} - \underline{V(s_{t-1})})$$

New Variable!

The Learning Rate

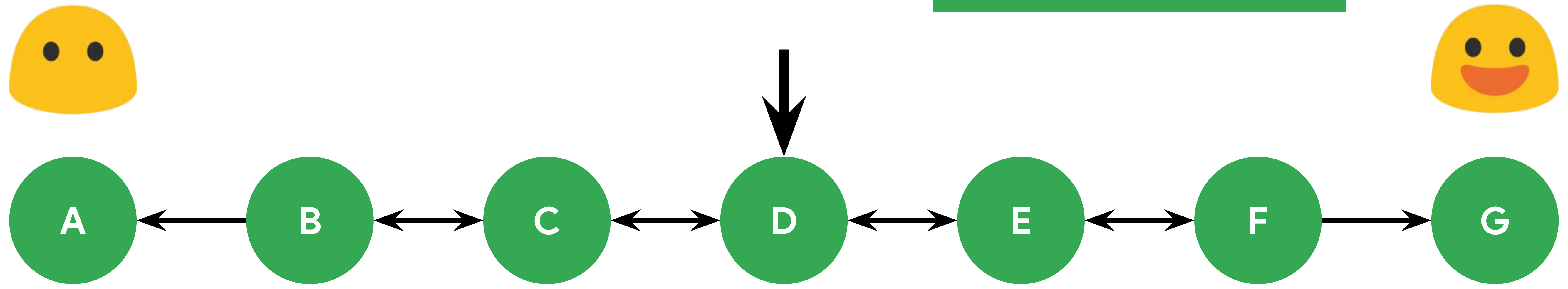
Google Cloud

TD(0) Random Walk



$\gamma = 1$

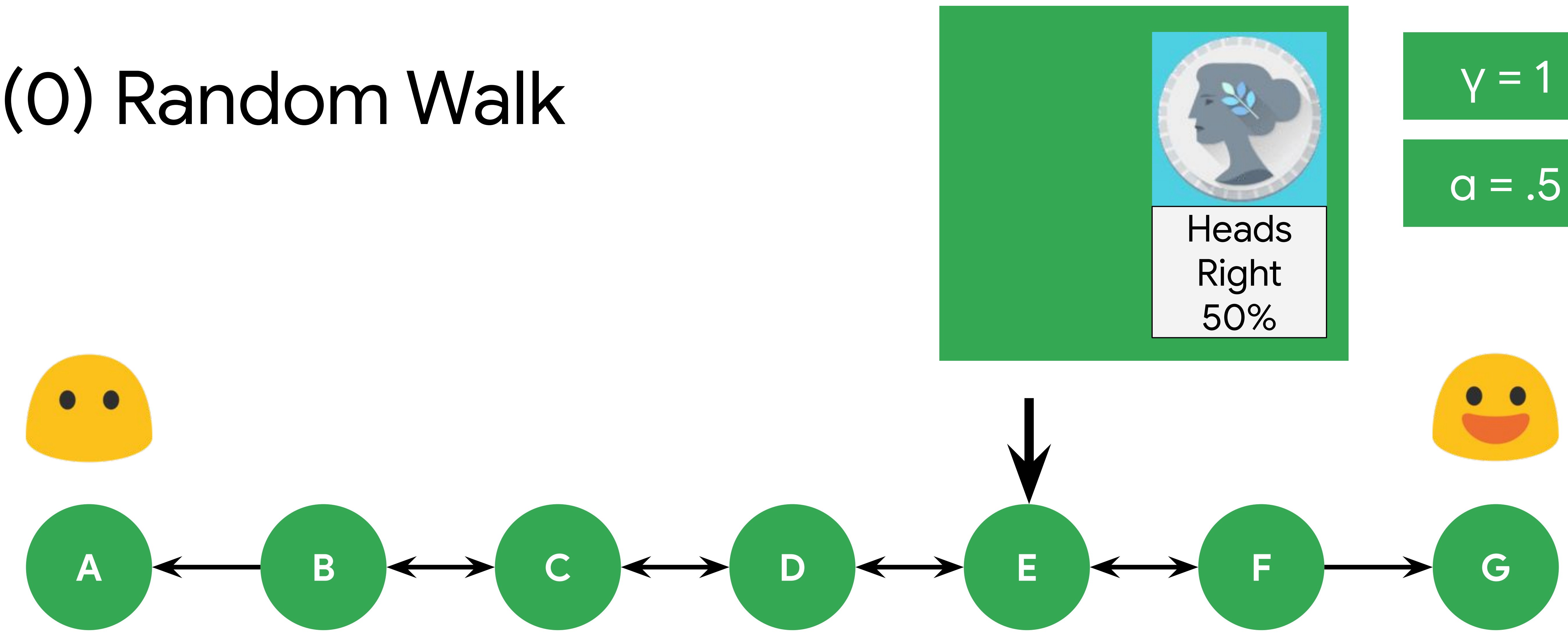
$\alpha = .5$



	A	B	C	D	E	F	G
Rewards	+0						+1
Value Iteration	0	.17	.33	.5	.67	.83	0
TD(0)	0	0	0	0	0	0	0

Google Cloud

TD(0) Random Walk



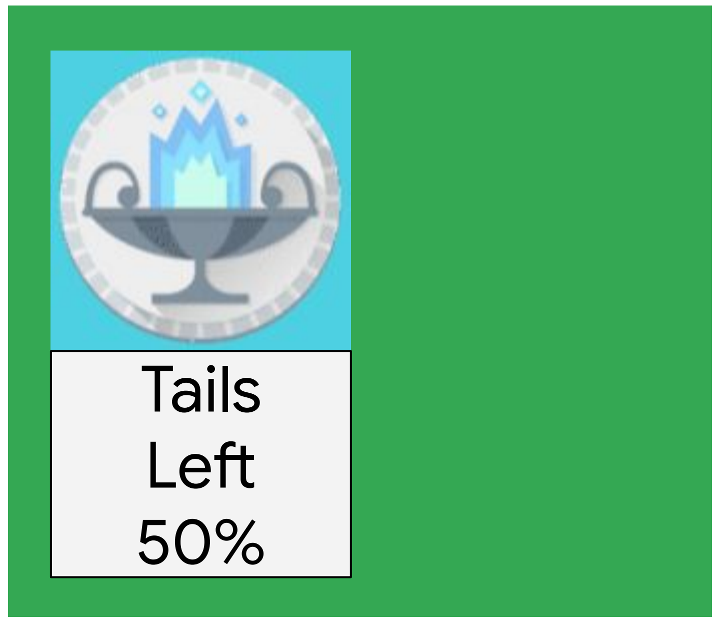
$\gamma = 1$

$\alpha = .5$

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

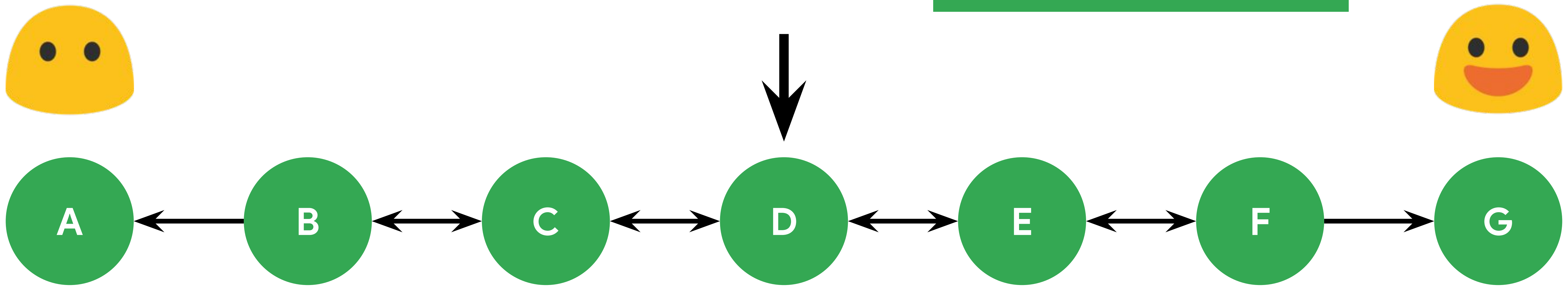
Google Cloud

TD(0) Random Walk



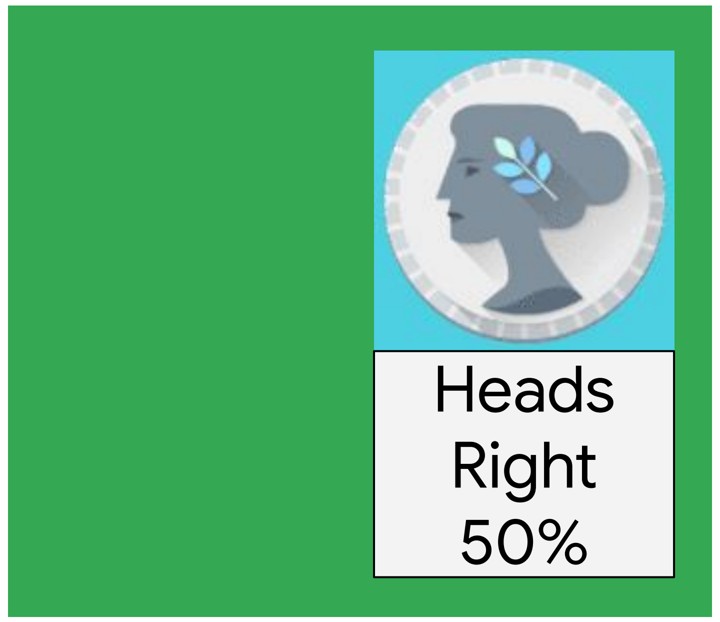
γ = 1

a = .5



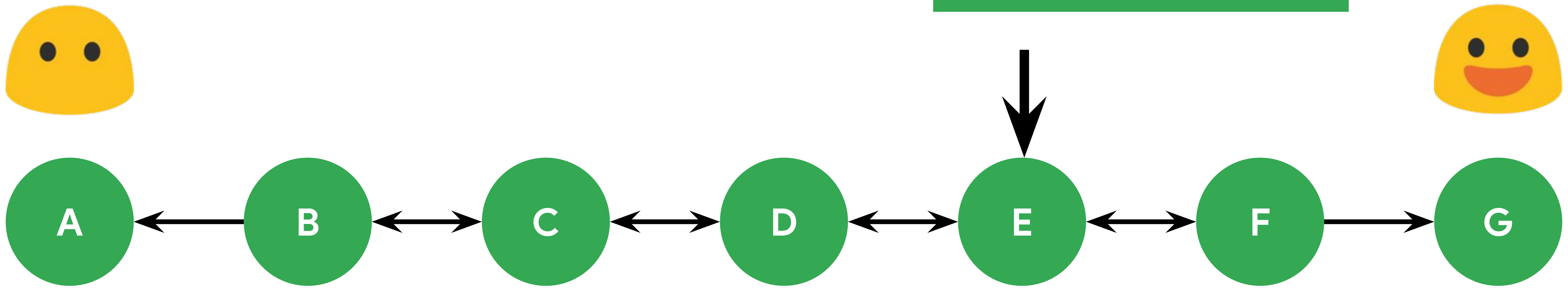
	A	B	C	D	E	F	G
Rewards	+0						+1
Value Iteration	0	.17	.33	.5	.67	.83	0
TD(0)	0	0	0	0	0	0	0

TD(0) Random Walk



$\gamma = 1$

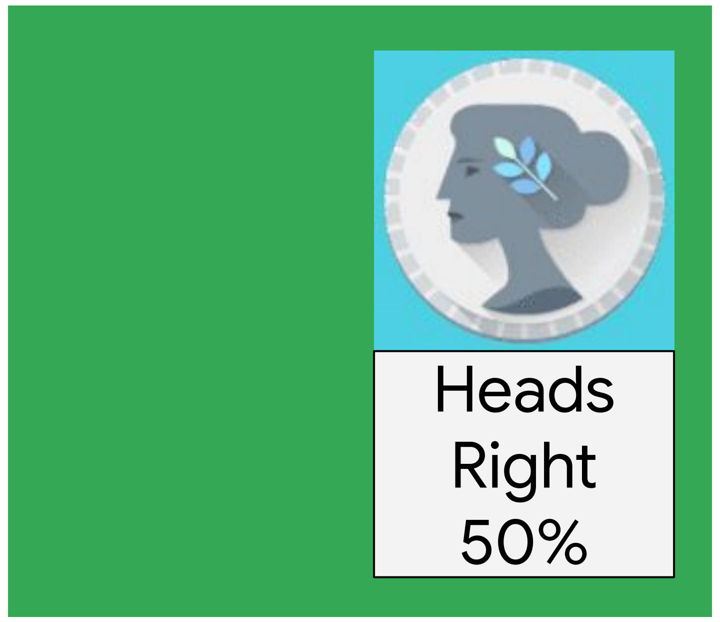
$\alpha = .5$



	A	B	C	D	E	F	G
Rewards	+0						+1
Value Iteration	0	.17	.33	.5	.67	.83	0
TD(0)	0	0	0	0	0	0	0

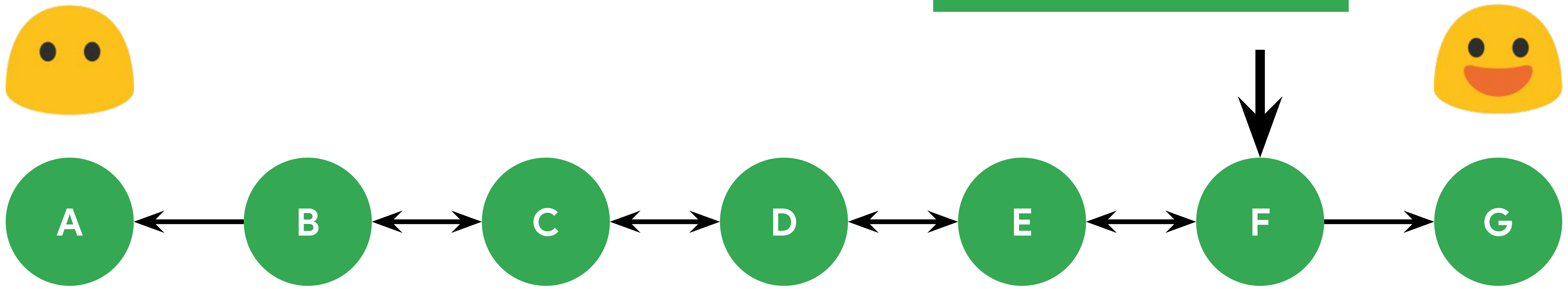
Google Cloud

TD(0) Random Walk



γ = 1

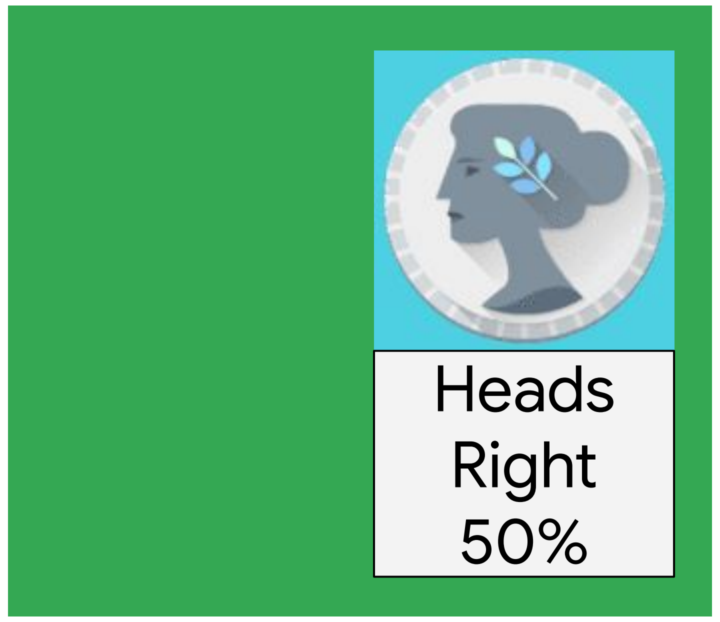
α = .5



	A	B	C	D	E	F	G
Rewards	+0						+1
Value Iteration	0	.17	.33	.5	.67	.83	0
TD(0)	0	0	0	0	0	0	0

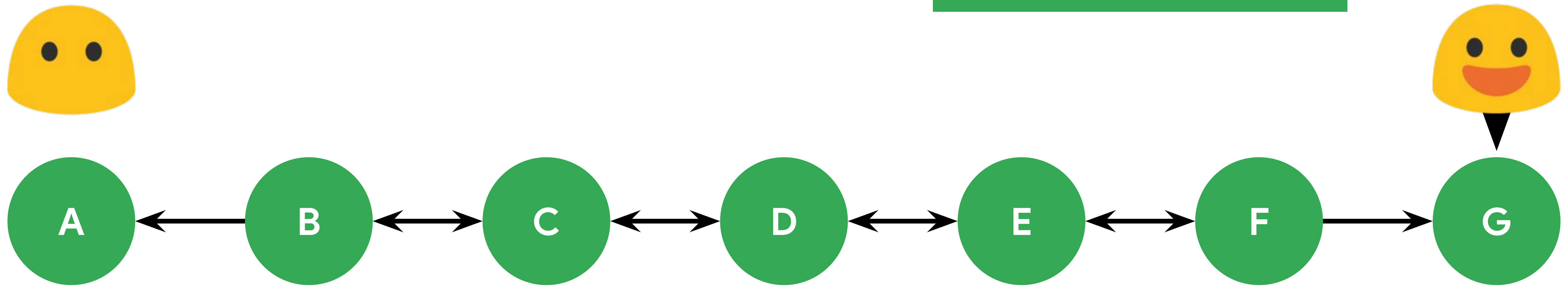
Google Cloud

TD(0) Random Walk



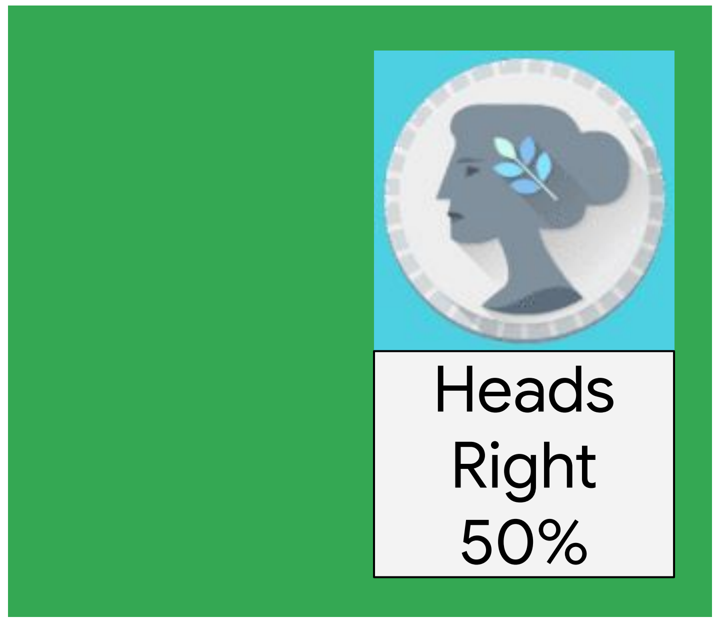
γ = 1

α = .5



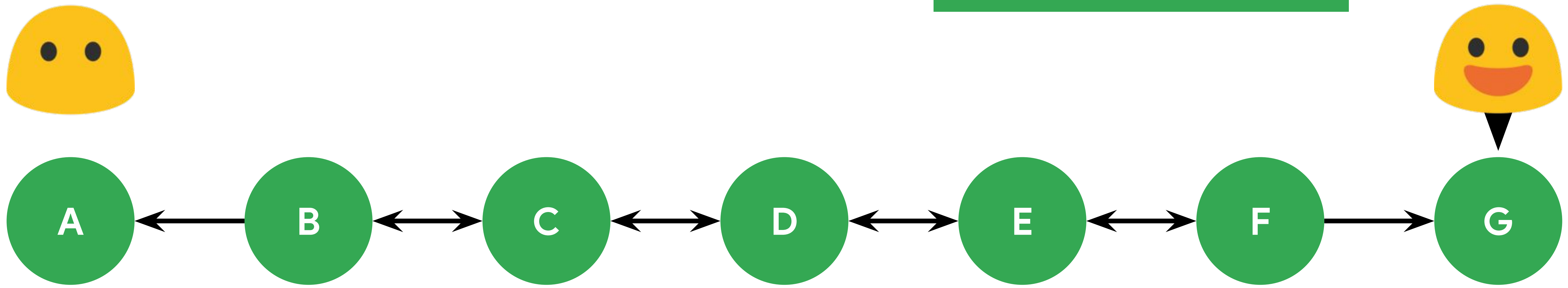
	A	B	C	D	E	F	G
Rewards	+0						+1
Value Iteration	0	.17	.33	.5	.67	.83	0
TD(0)	0	0	0	0	0	0	0

TD(0) Random Walk



$\gamma = 1$

$\alpha = .5$

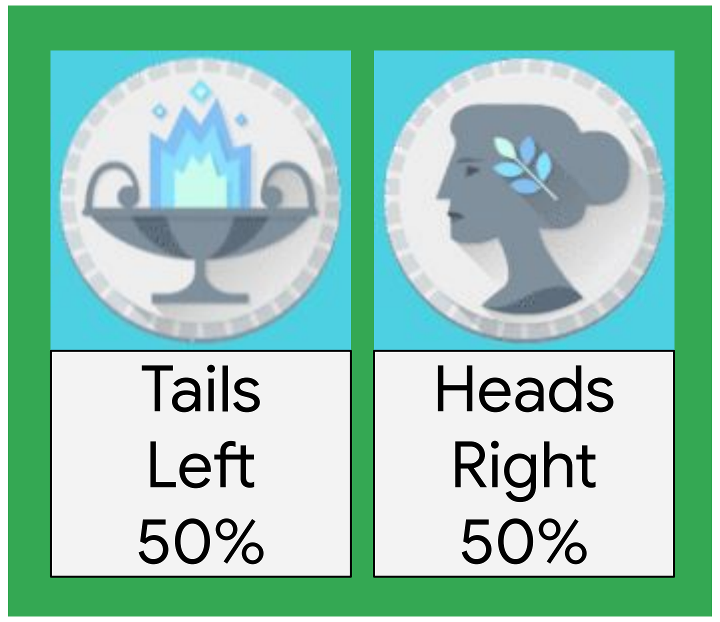


$$V(s_{t-1}) = V(s_{t-1}) + \alpha_t(R(s_{t-1}, a) + \gamma V(s_t) - V(s_{t-1}))$$

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

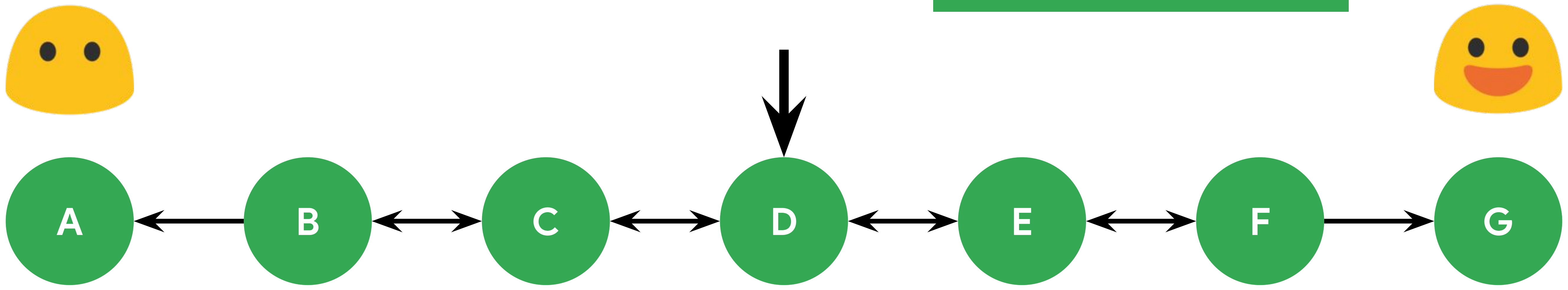
TD(0)	0	0	0	0	0	.5	0

TD(0) Random Walk



$\gamma = 1$

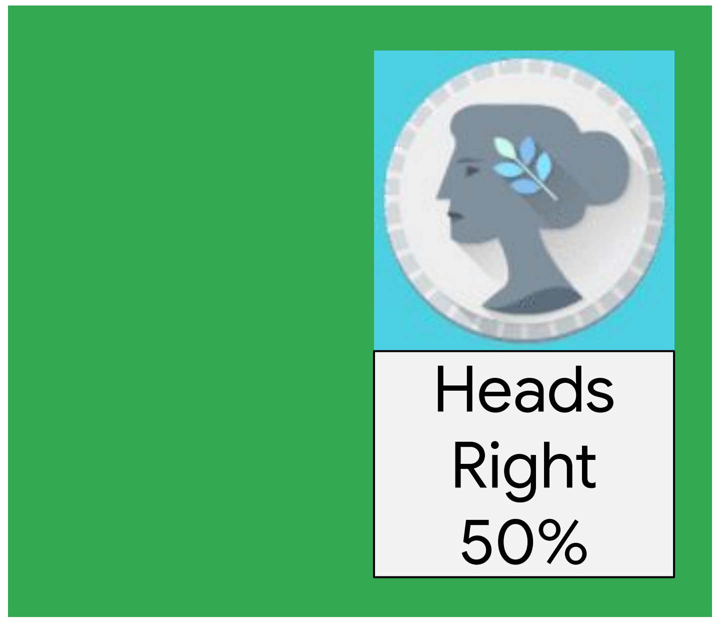
$\alpha = .5$



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

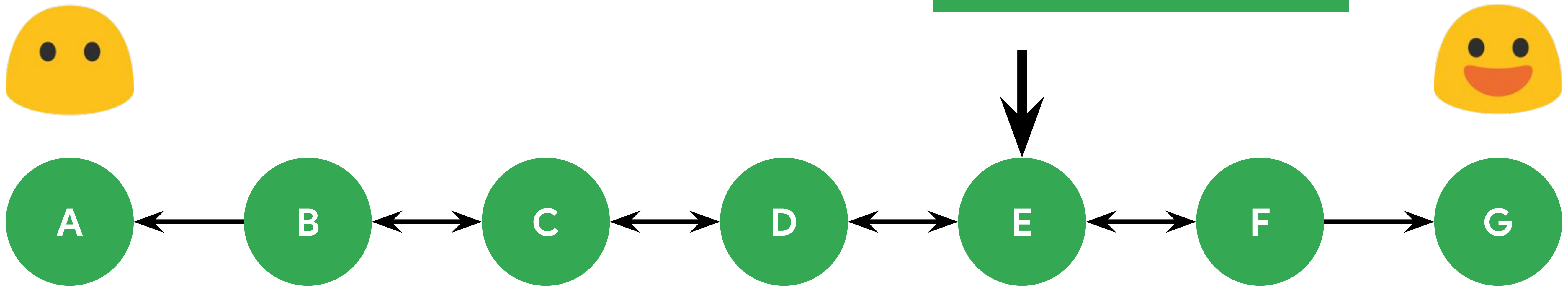
Google Cloud

TD(0) Random Walk



$\gamma = 1$

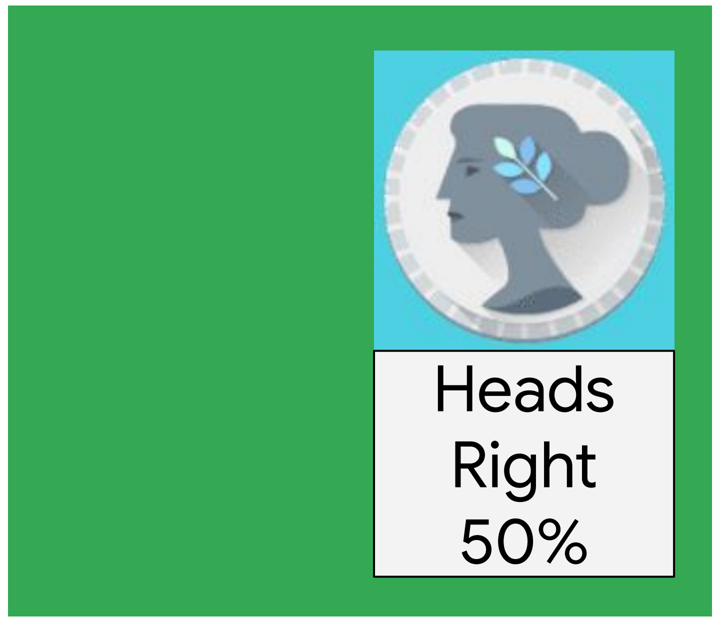
$\alpha = .5$



	A	B	C	D	E	F	G
Rewards	+0						+1
Value Iteration	0	.17	.33	.5	.67	.83	0
TD(0)	0	0	0	0	0	.5	0

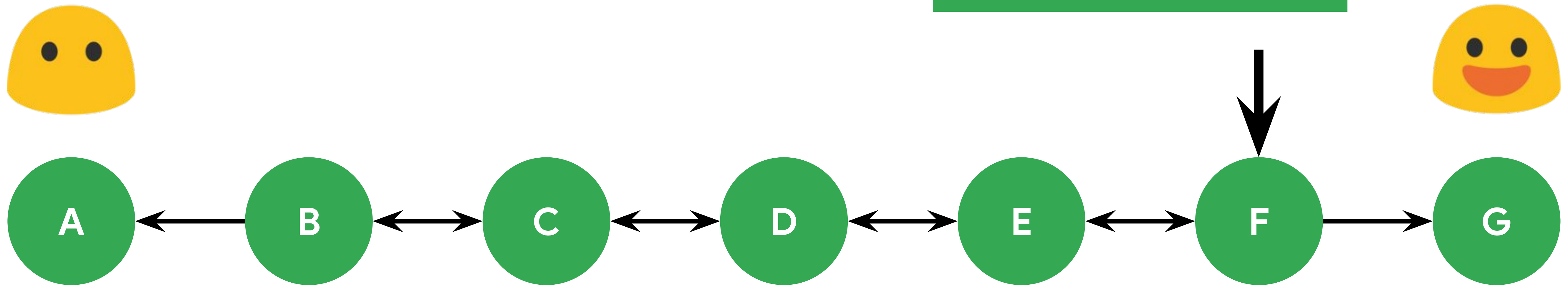
Google Cloud

TD(0) Random Walk



$\gamma = 1$

$\alpha = .5$



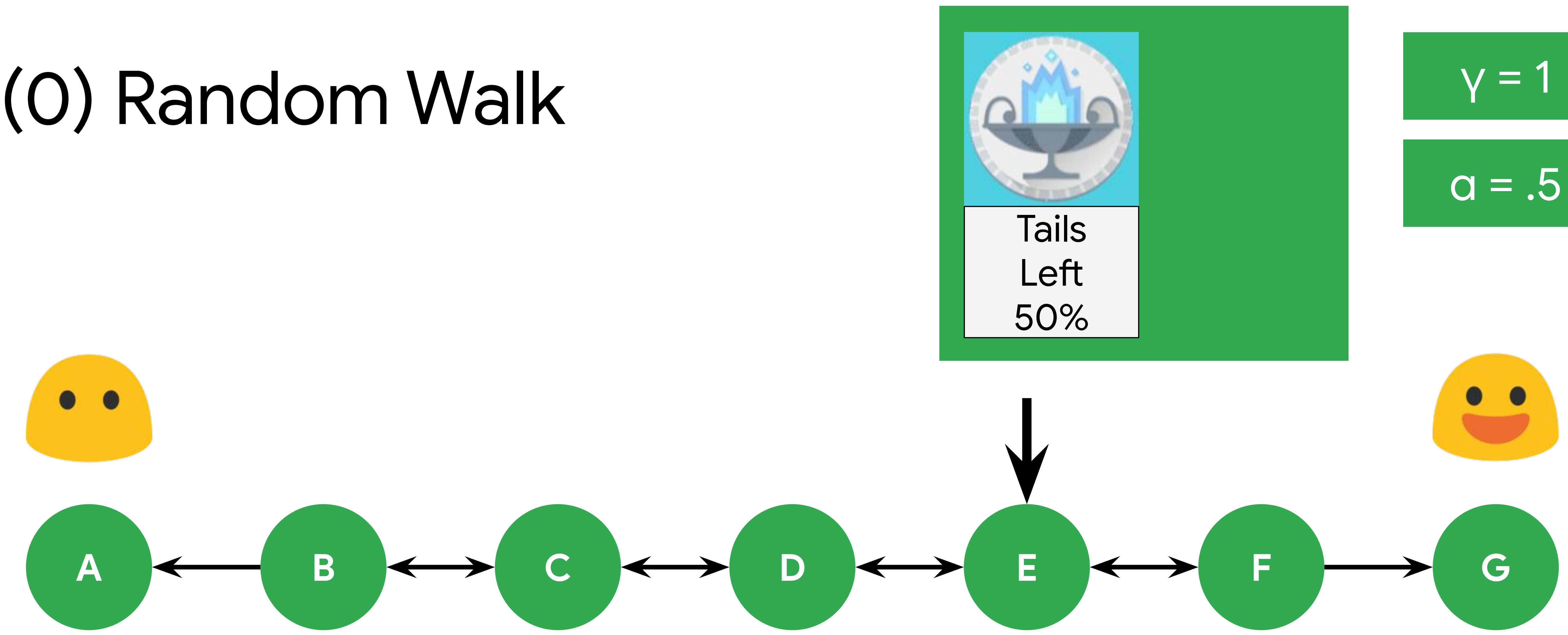
	A	B	C	D	E	F	G
Rewards	+0						+1
Value Iteration	0	.17	.33	.5	.67	.83	0
TD(0)	0	0	0	0	.25	.5	0

Google Cloud

TD(0) Random Walk

γ = 1

a = .5



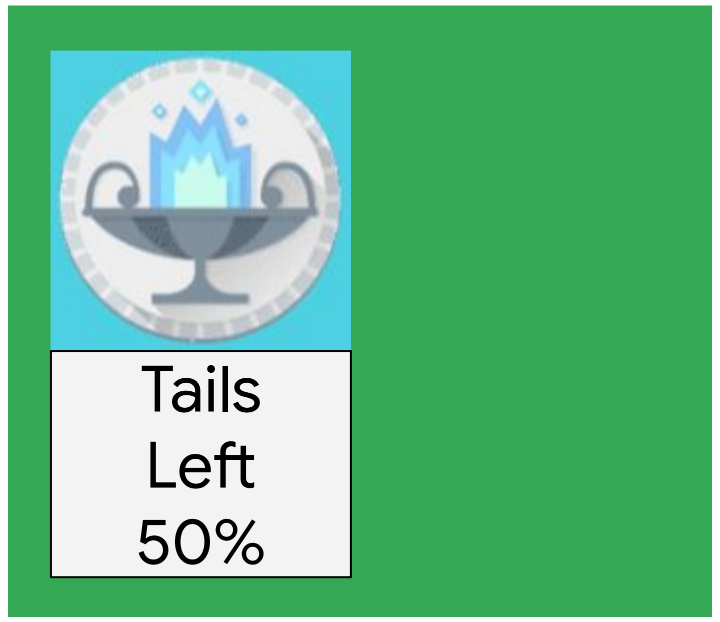
$$V(s_{t-1}) = V(s_{t-1}) + \alpha_t(R(s_{t-1},a) + \gamma V(s_t) - V(s_{t-1}))$$

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

TD(0)	0	0	0	0	.25	$\boxed{.375}$	0

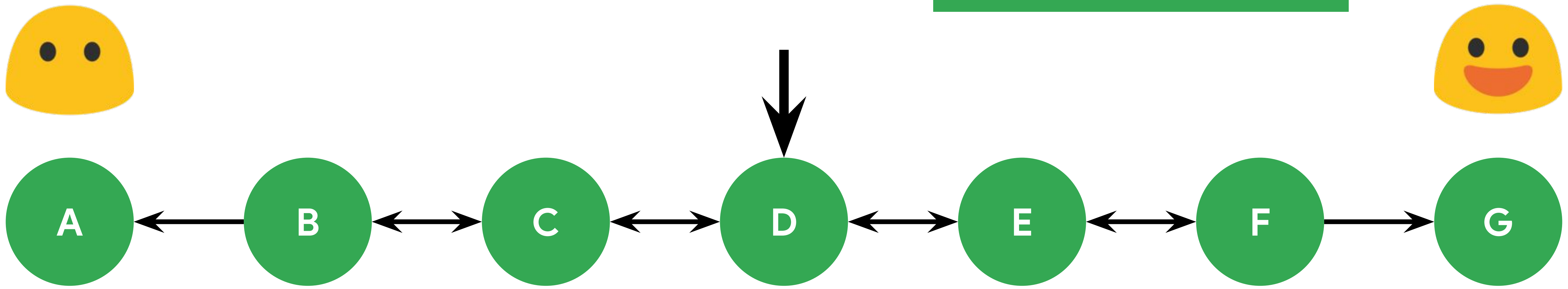
Google Cloud

TD(0) Random Walk



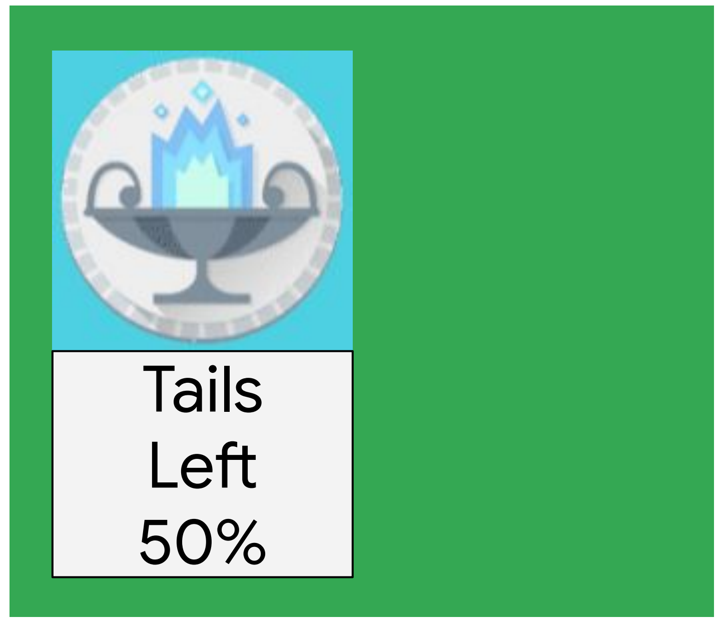
$\gamma = 1$

$\alpha = .5$



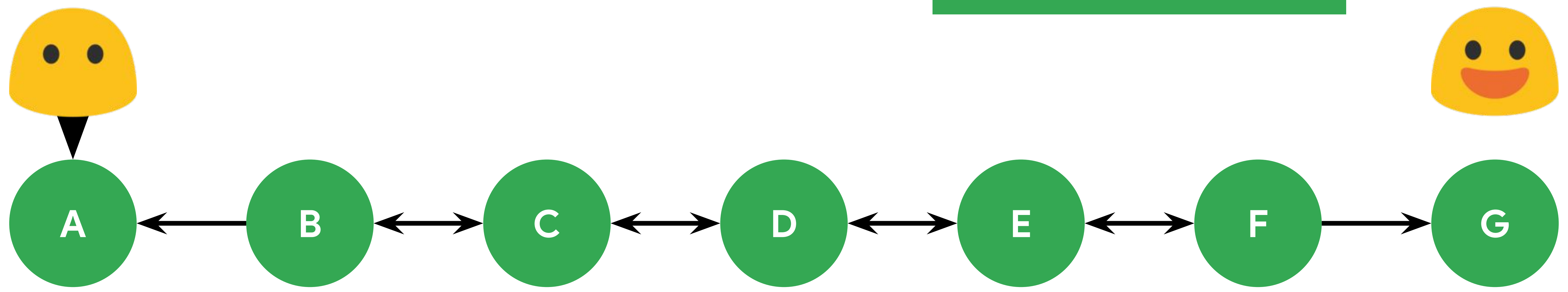
	A	B	C	D	E	F	G
Rewards	+0						+1
Value Iteration	0	.17	.33	.5	.67	.83	0
TD(0)	0	0	0	0	.125	.375	0

TD(0) Random Walk



$\gamma = 1$

$\alpha = .5$

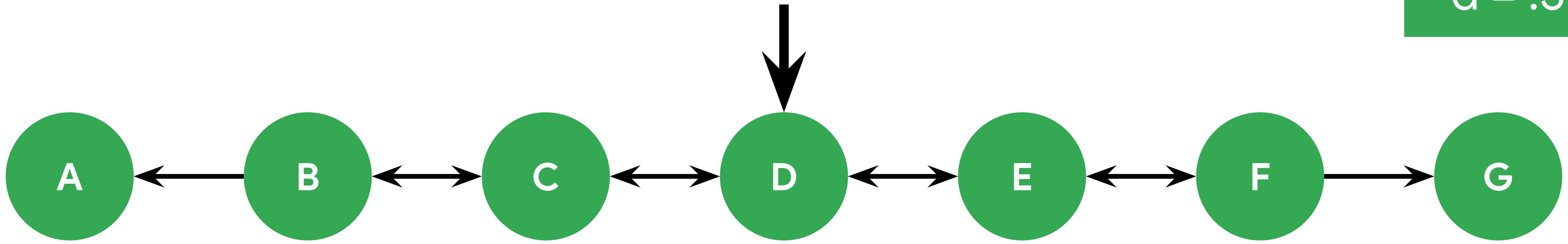


Rewards	+0						+1
Value Iteration	0	.17	.33	.5	.67	.83	0
TD(0)	0	.016	.031	.063	.125	.375	0

TD(1) Random Walk

γ = .9

α = .5

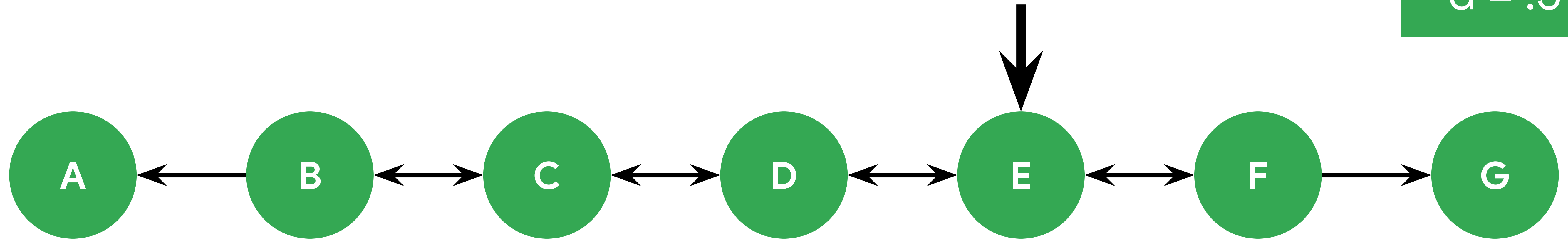


Rewards	+0						+1
Eligibility	0	0	0	1	0	0	0
TD(1)	0	0	0	0	0	.5	0

TD(1) Random Walk

$\gamma = .9$

$\alpha = .5$

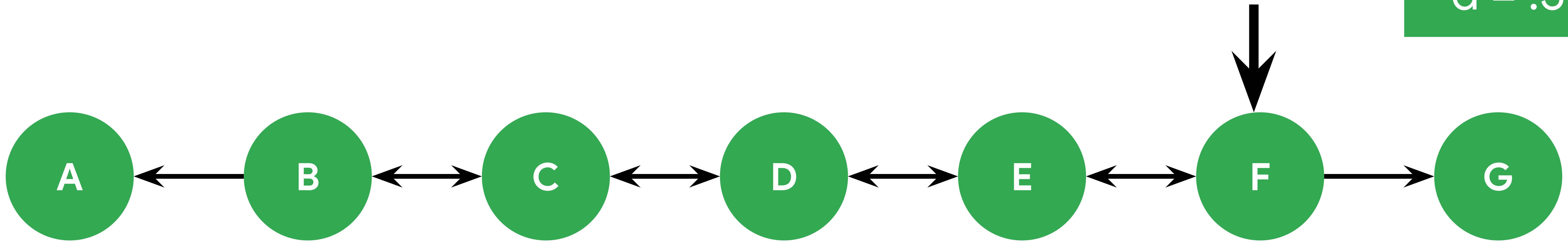


Rewards	+0						+1
Eligibility	0	0	0	γ	1	0	0
TD(1)	0	0	0	0	0	.5	0

TD(1) Random Walk

γ = .9

α = .5



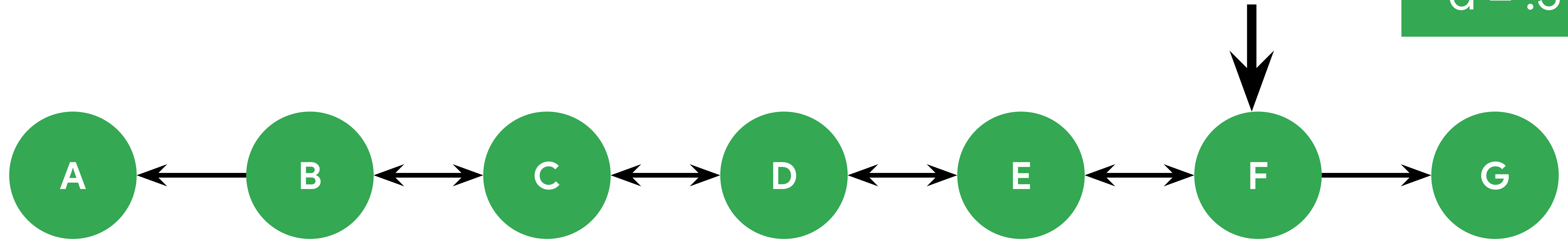
Rewards	+0						+1
Eligibility	0	0	0	γ	1	0	0
TD(1)	0	0	0	0	0	.5	0

Google Cloud

TD(1) Random Walk

γ = .9

α = .5

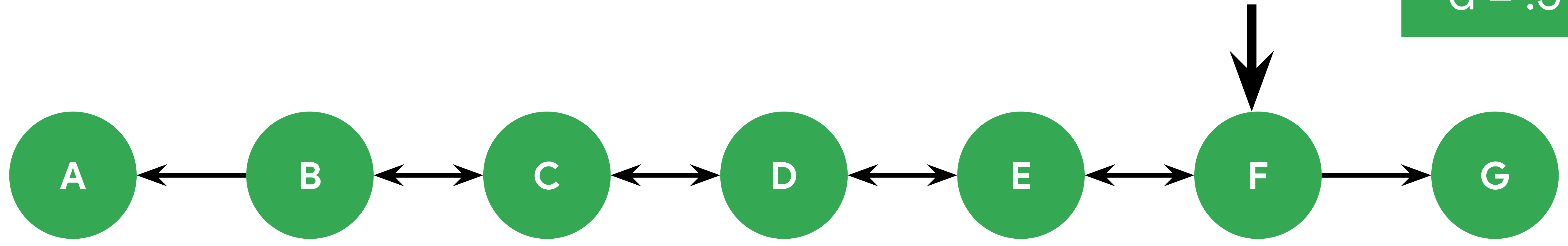


Rewards	+0						+1
Eligibility	0	0	0	γ	1	0	0
TD(1)	0	0	0	.202	.225	.5	0

TD(1) Random Walk

$\gamma = .9$

$\alpha = .5$



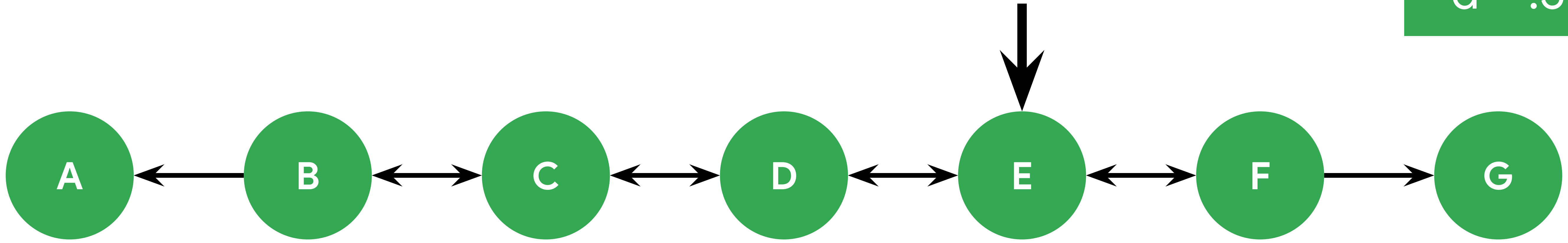
	A	B	C	D	E	F	G
Rewards	+0						+1
Eligibility	0	0	0	$\boxed{\gamma^2}$	$\boxed{\gamma}$	$\boxed{1}$	0
TD(1)	0	0	0	.202	.225	.5	0

Google Cloud

TD(1) Random Walk

$\gamma = .9$

$\alpha = .5$



Rewards	+0						+1
Eligibility	0	0	0	γ^2	γ	1	0
TD(1)	0	0	0	$\boxed{.182}$	$\boxed{.202}$	$\boxed{.475}$	0

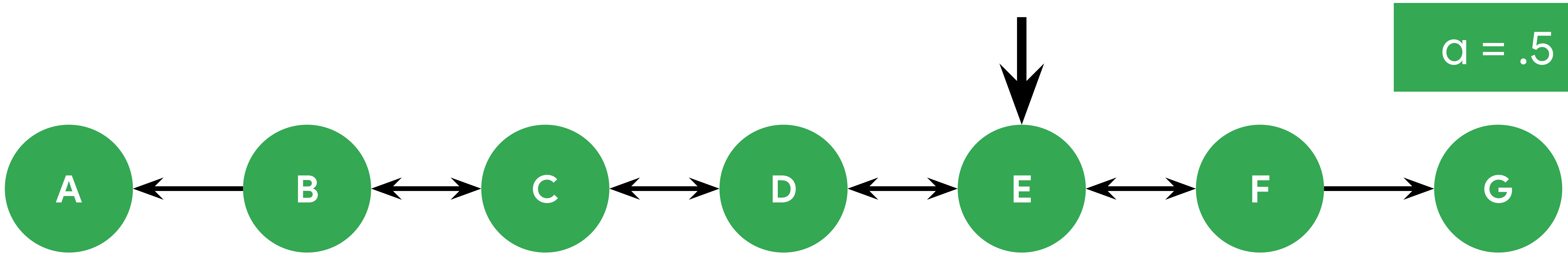
Changed by -.025

TD(λ) Random Walk

$\lambda = 1$

$\gamma = .9$

$\alpha = .5$



Rewards	+0						+1
Eligibility	0	0	0	$(\lambda\gamma)^2$	$\lambda\gamma$	1	0
TD(1)	0	0	0	.182	.202	.475	0

Changed by -.025

Google Cloud

Agenda

History Overview

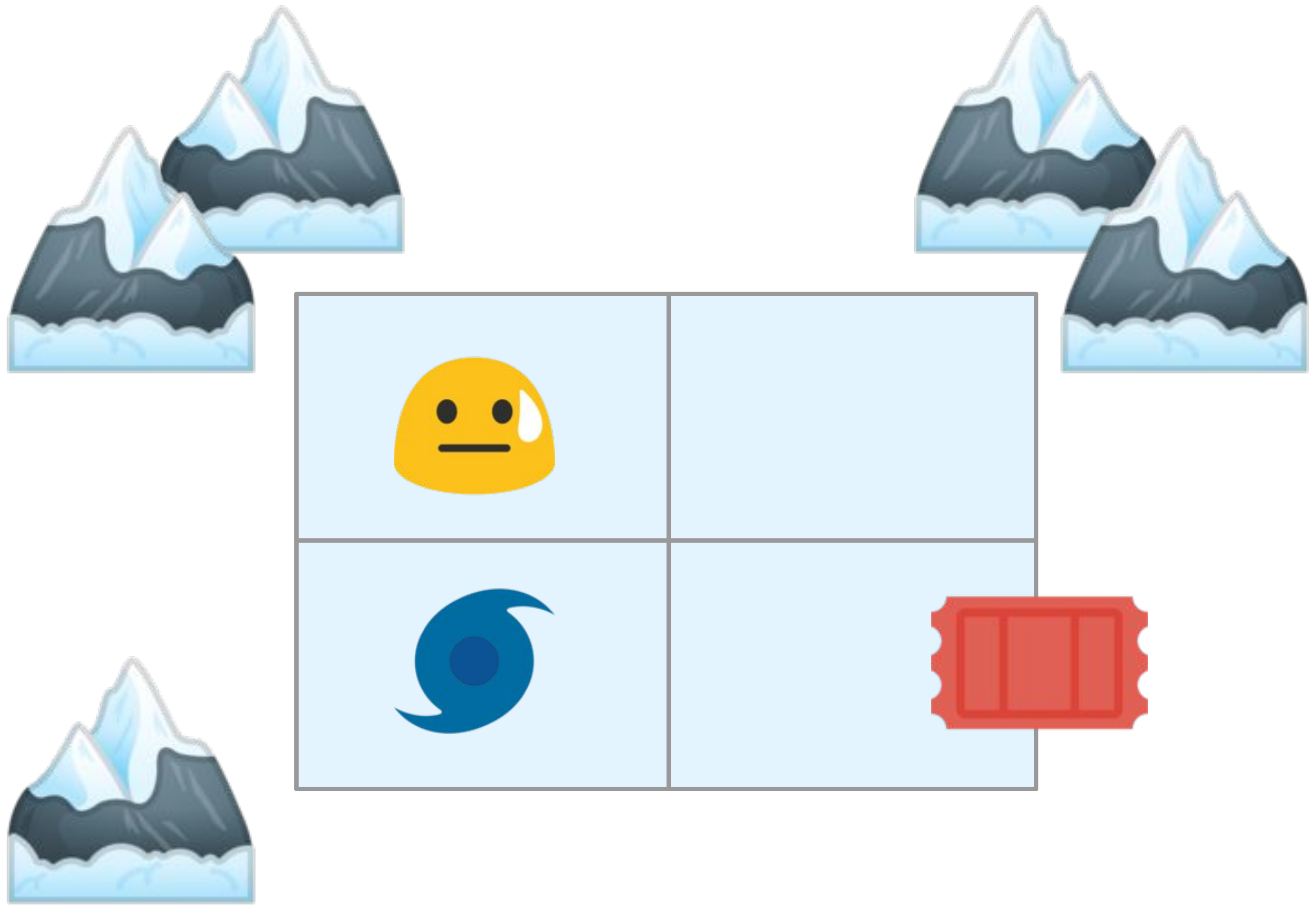
Value Iteration

Policy Iteration

TD(Lambda)

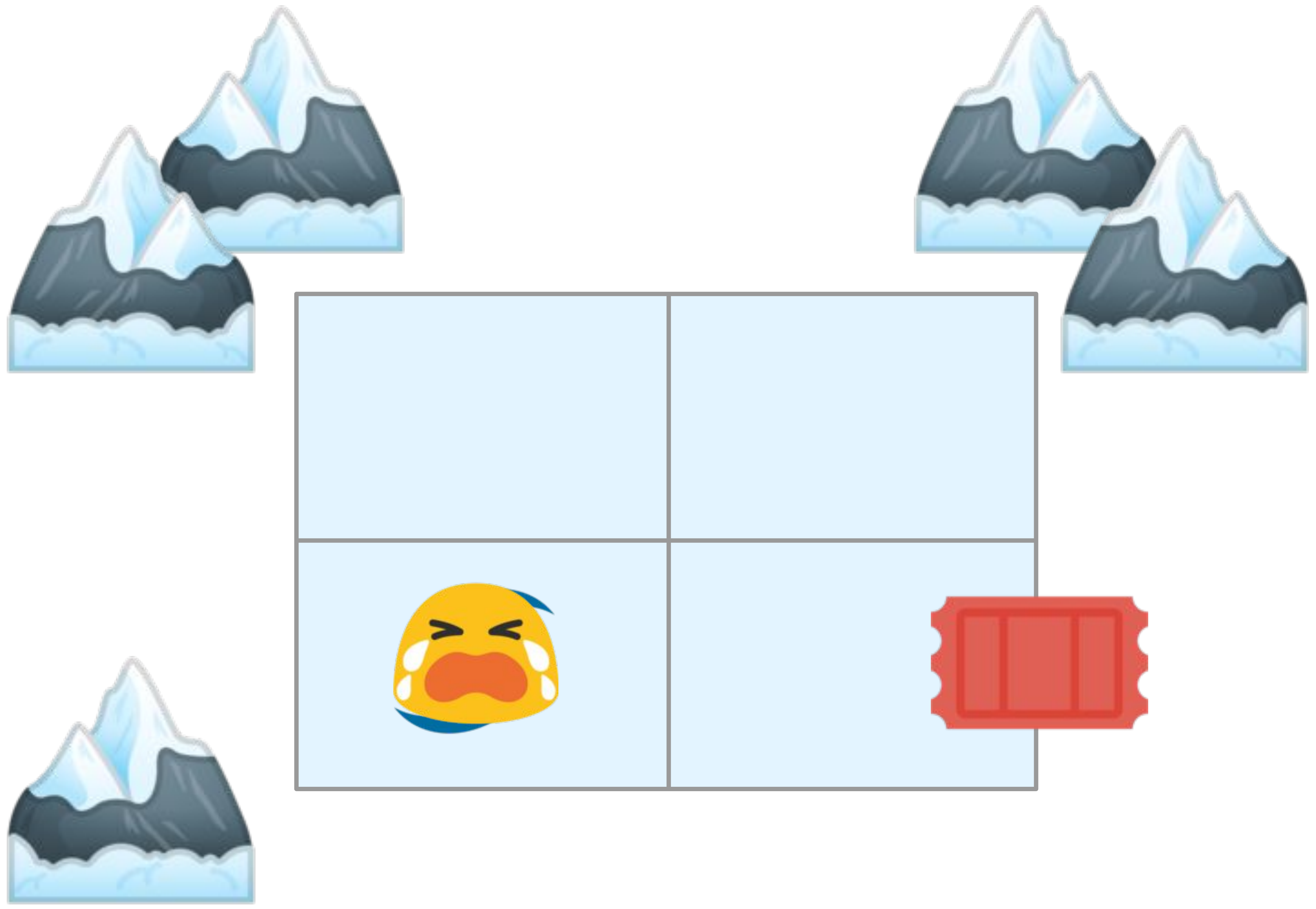
Q-Learning

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

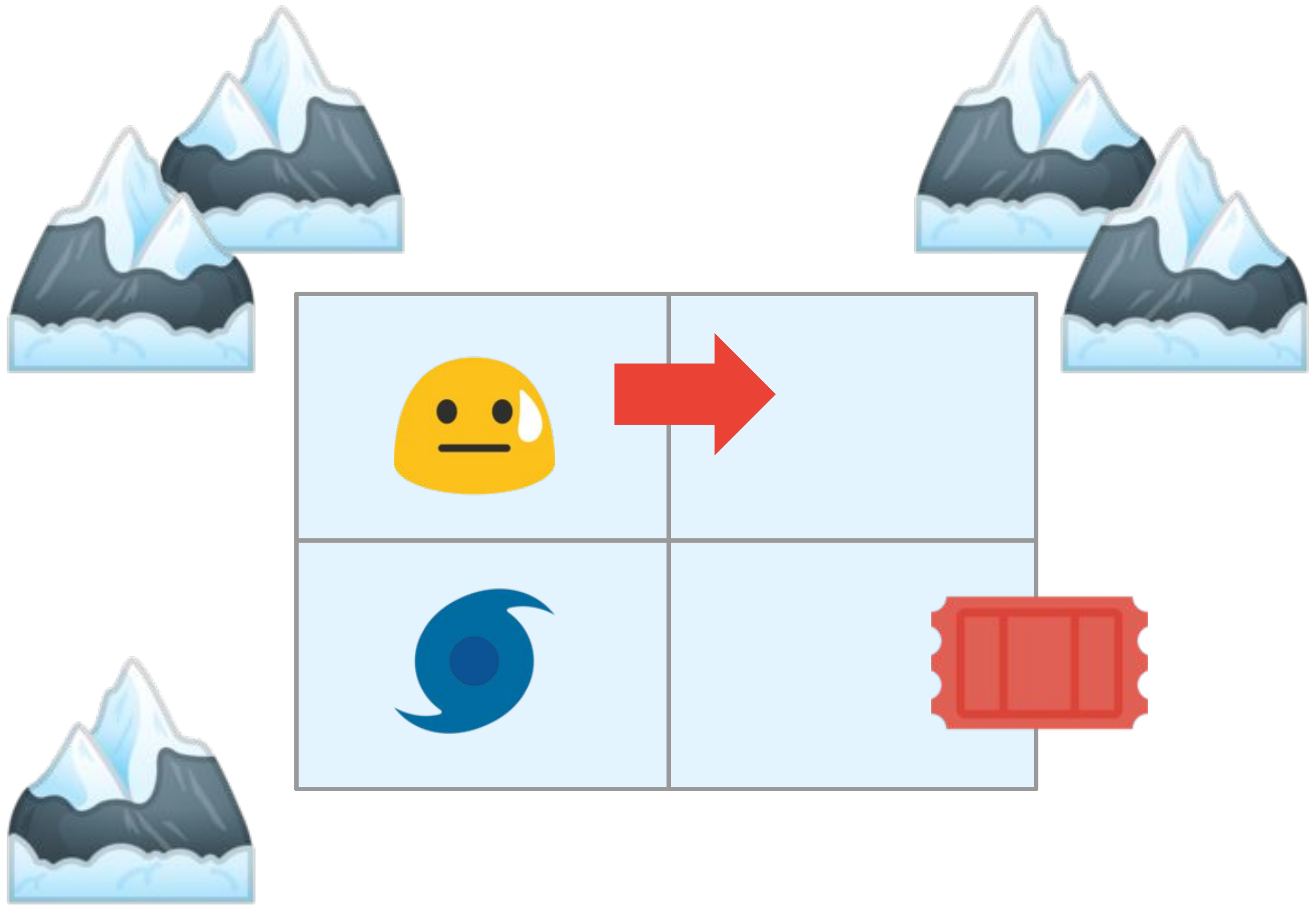


Q - table				
	Left	Down	Right	Up
0	0	-.5	0	0
1	0	0	0	0
2	0	0	0	0
3	0	0	0	0

$\gamma = .9$ $a = .5$

Google Cloud

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

γ = .9

α = .5

Google Cloud

Deep Q Learning

$$V(s_{t-1}) = V(s_{t-1}) + \square_t\left(R(s_{t-1},a_{t-1}) + \gamma \cdot V(s_t) - V(s_{t-s})\right)$$

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

Deep Q Learning

$$V(s_{t-1}) = V(s_{t-1}) + \square_t\left(R(s_{t-1},a_{t-1}) + \gamma \cdot V(s_t) - V(s_{t-s})\right)$$

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

To compare the equation on what we had before with TD Lambda. The Q function is extremely similar except it now accounts for the state action pair is besides just the state. There is one thing to note which is now that we're finding the value of a state action pair which action should we use from State Prime. Watkins logic was this. We'll use the action that we would use if we were in state prime. Which would be the action that gives us the maximum value. So I'll just look at the Q table row that corresponds to State Prime and use the maximum value.

Google Cloud

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):
        ...
```

Our agent needs three key things:
1- away to initialize the Q table
2- a way to update it with new information and
3- a way to choose an action based on the policy.

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):
        ...
```

If we know the total number of states and actions initializing our Q table is not bad at all. We just tell numpy the number of states and actions.
if we don't know those things then no problem. We can make python dictionaries that map states and actions to rows and columns. If we come across the state or action that is not in those dictionaries. Then we expand the size of our Q table and add the new indexes to our mappings.

Google Cloud

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):
        ...
```

It's the TD0 update rule with the max election for State Prime to represent the action we would take in that state. The key here is the line where we calculate the future value. We take the max value corresponding to the Q table row for State Prime.

Google Cloud

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
```

Google Cloud

Finally we'll add in a new way to act given the current situation we're in. This one is deceptively tricky. First, we'll grab the row corresponding to our current state we could use numpy argMax function to find the action index corresponding to the maximum value. But that's going to biase our agents actions. how? if we have ties for maximum values, numpy will only return the first occuring index. Instead, we will use a argMax to find the indexes of all the values that are equal to the maximum, then we will randomly select one from those.

On Purpose Mistakes?

On-Policy vs Off-Policy. the difference is in off policy algorithms will do exploration

There's one last observation Watkins had about animals that he included in Q learning. In this research he learned that animals will purposely make mistakes when they're in a safe place in order to improve their understanding of the environment. So far all the algorithms we've learned are called on policy. That means given the information currently available to us we've gone with the best note action. Watkins introduced off policy, which is to purposely do something different than the best-known action for the sake of exploration.

Google Cloud

Anatomy of an Agent

There are a few ways to incorporate this exploration versus exploitation. Turns out one of the easiest ways is also one of the most popular. We'll introduce a new variable called the random rate. It's also called Epsilon and some circles this represents the fraction of times we want to choose a completely random action.

```
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
```

Then in the act function will roll a random decimal between 0 and 1 and see if it's lower than a random rate, we'll roll a random action (Exploration), else if it isn't, we'll find the best action based on a Q table like before (Exploitation).
We'll only do this when we're training just like humans or robots and it need a safe environment to try new things to make mistakes.
But when it comes to a moment that mistakes will count, it will do what it knows is best.
Finally, let's put it all together.

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)
```

So the tricky thing here is adding in the environment. Thankfully OpenAi gym makes it super easy for us. All I have to do is pass in the name of the game FrozenLake-v0 and it will build an environment for agent to interact with.

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)
```

Google Cloud

File Name:
T-AIFORF-I-p3_M1_l10_benefits_of_using_reinforcement_learning_in_your_trading_strategy_part1

Content Type: Video - Lecture Presenter

Presenter: Jack Farmer

ESTABLISHED 1922
NEW YORK INSTITUTE OF FINANCE

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

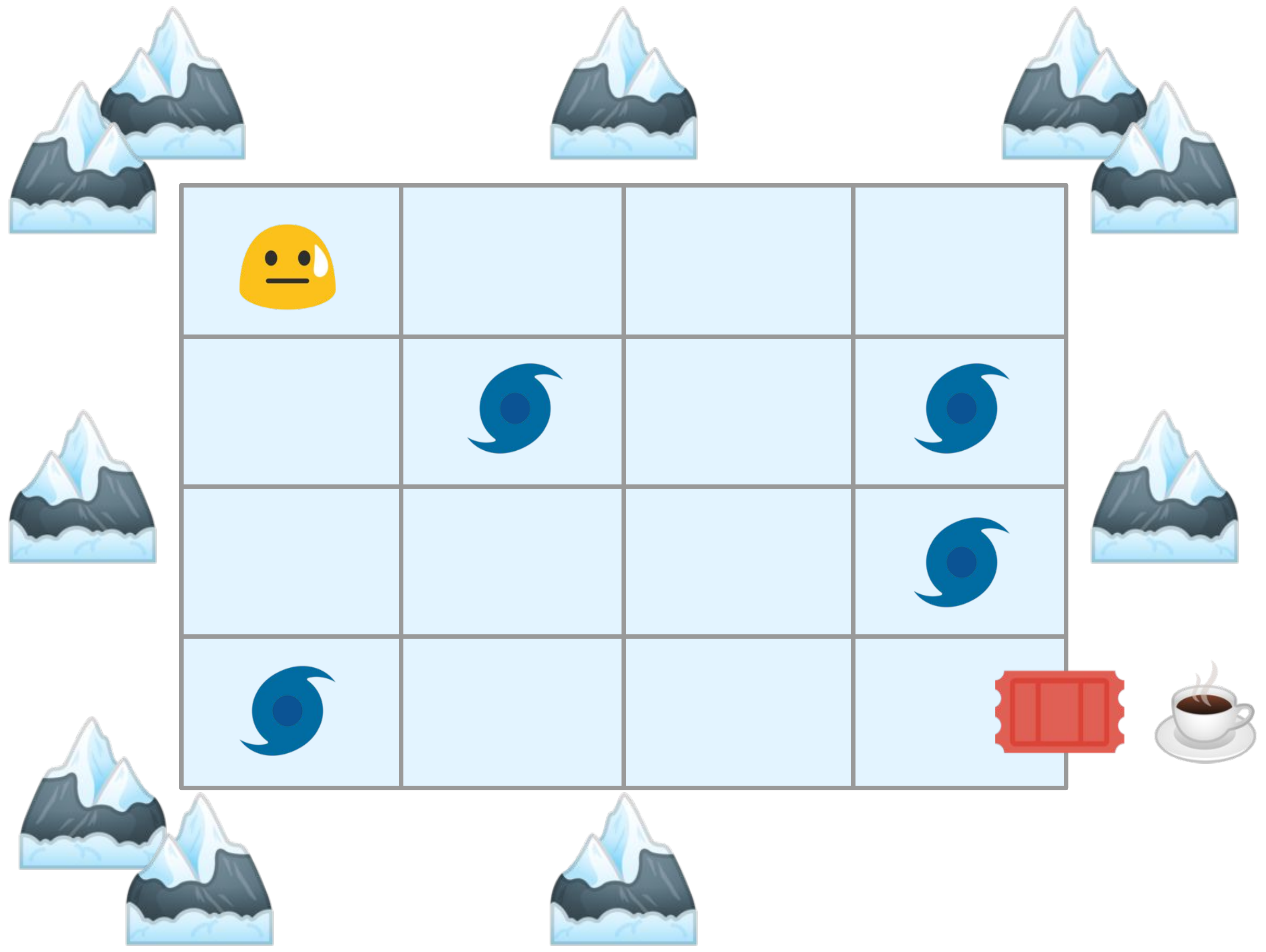
What is Deep Reinforcement Learning?

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

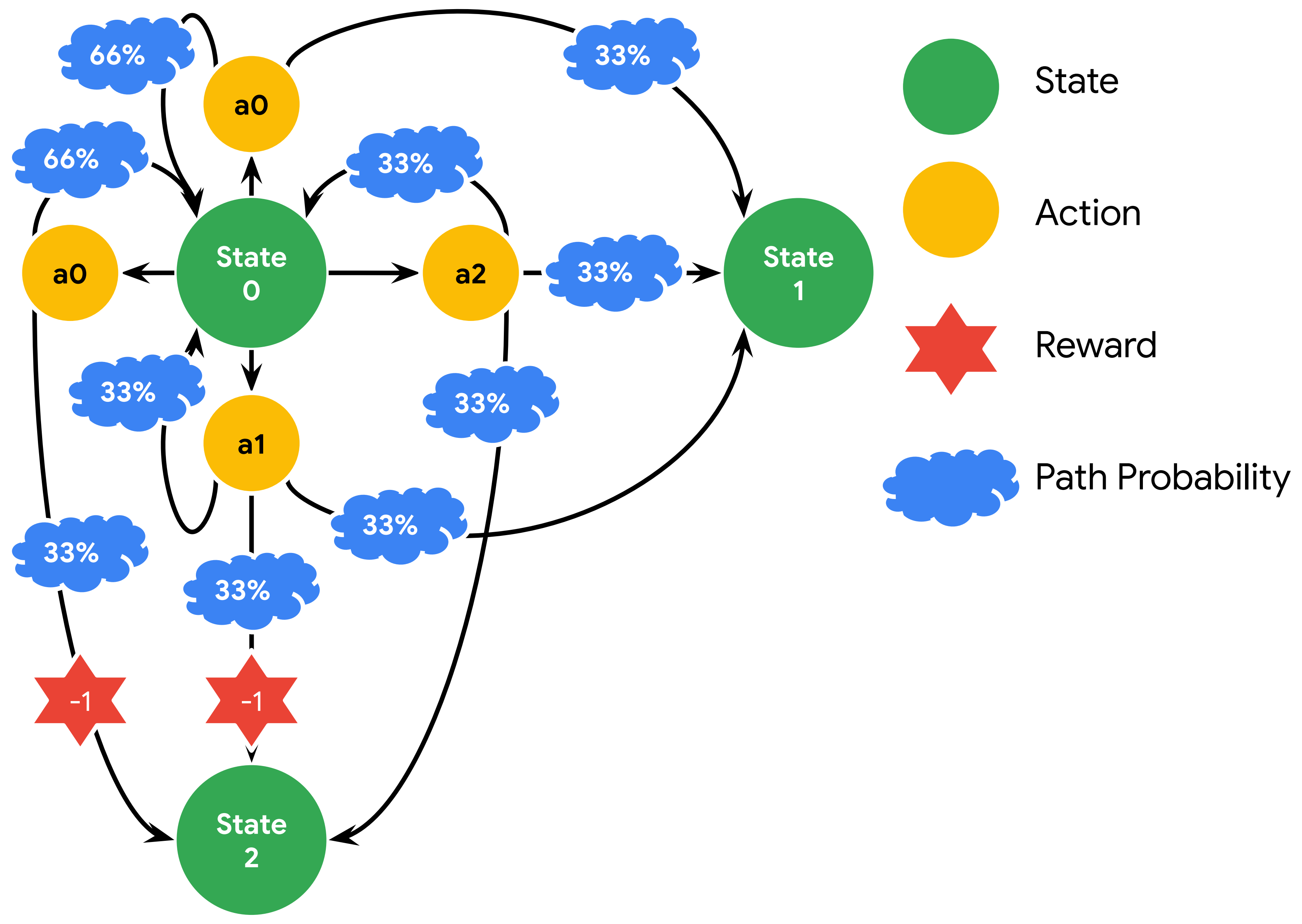


DRL Agent

- 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

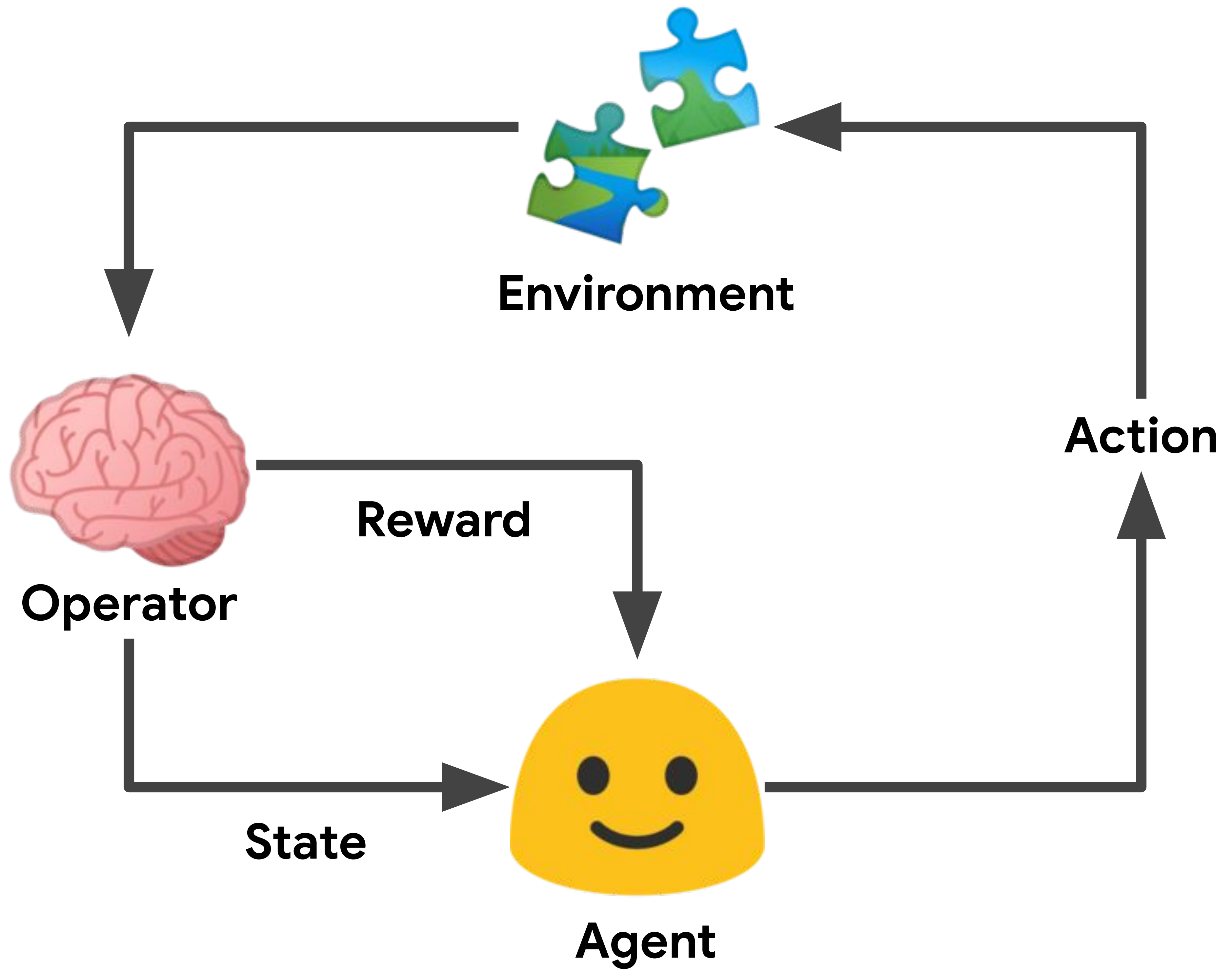
- Strategies require error-free handling of large volumes of data
- Agents' actions may result in longer-term consequences that other ML techniques are unable to measure
- And also have short-term impacts on the current market conditions which makes the trading environment highly unpredictable

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

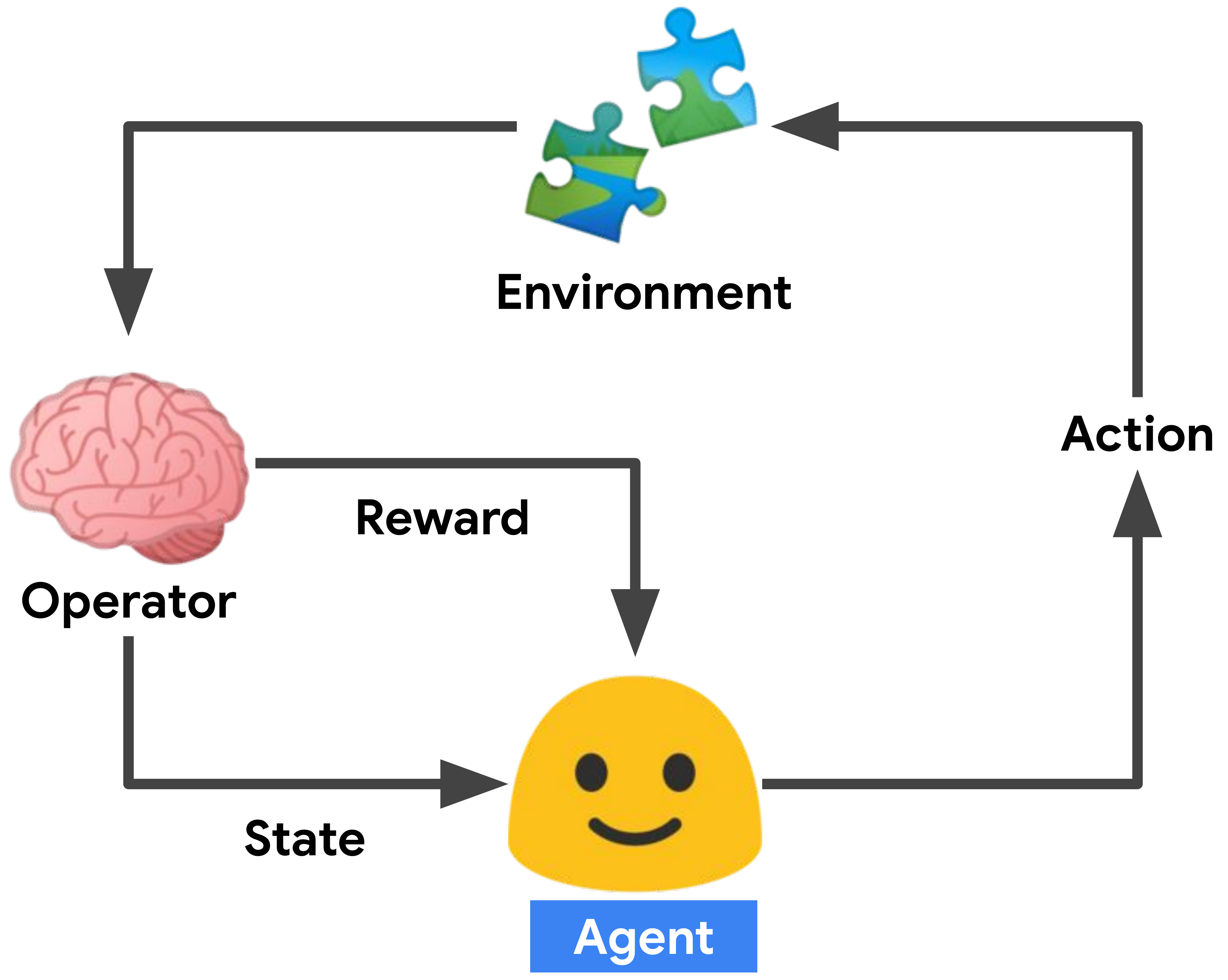
DRL Trading Algorithm Components

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



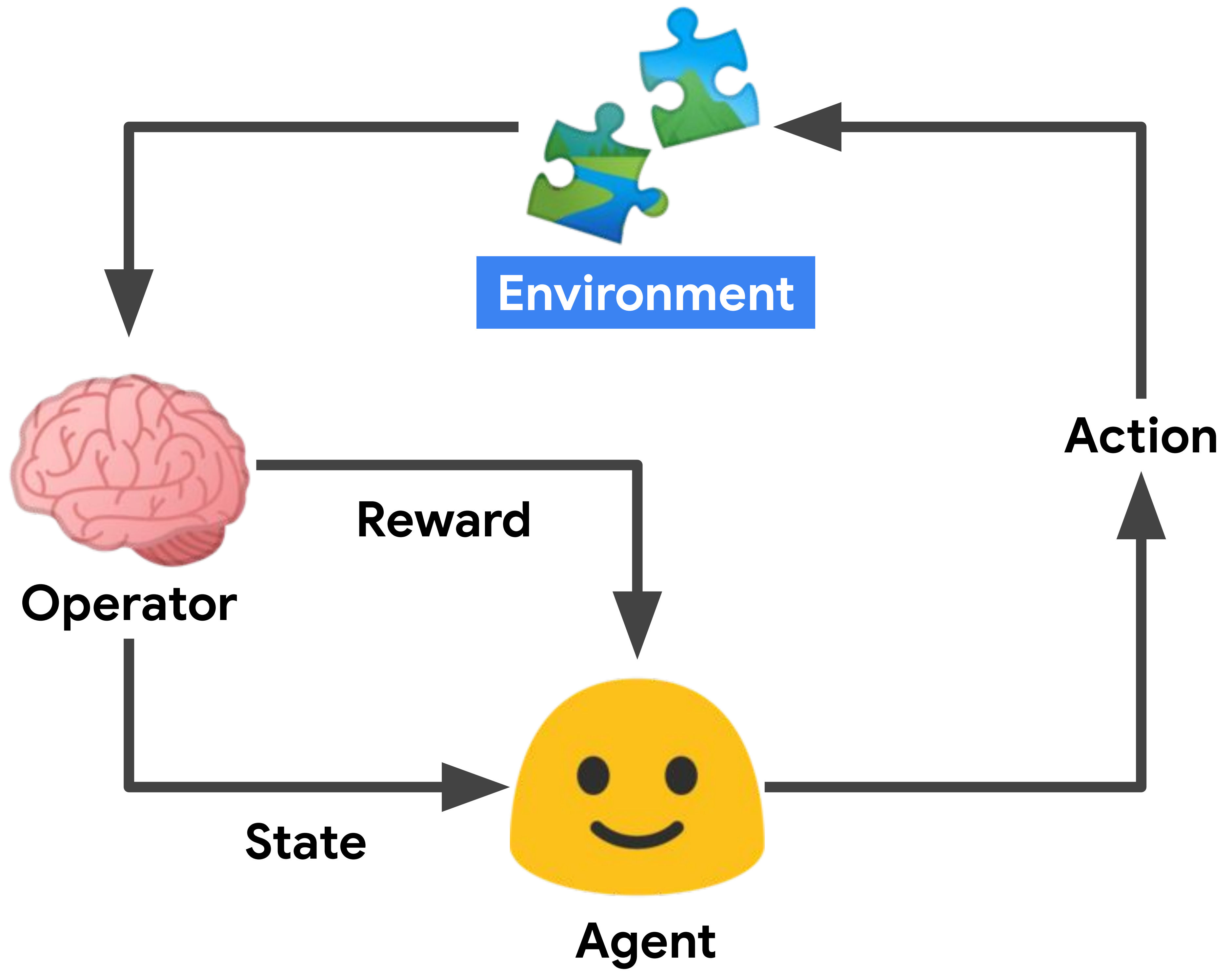
DRL Trading Algorithm Components

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



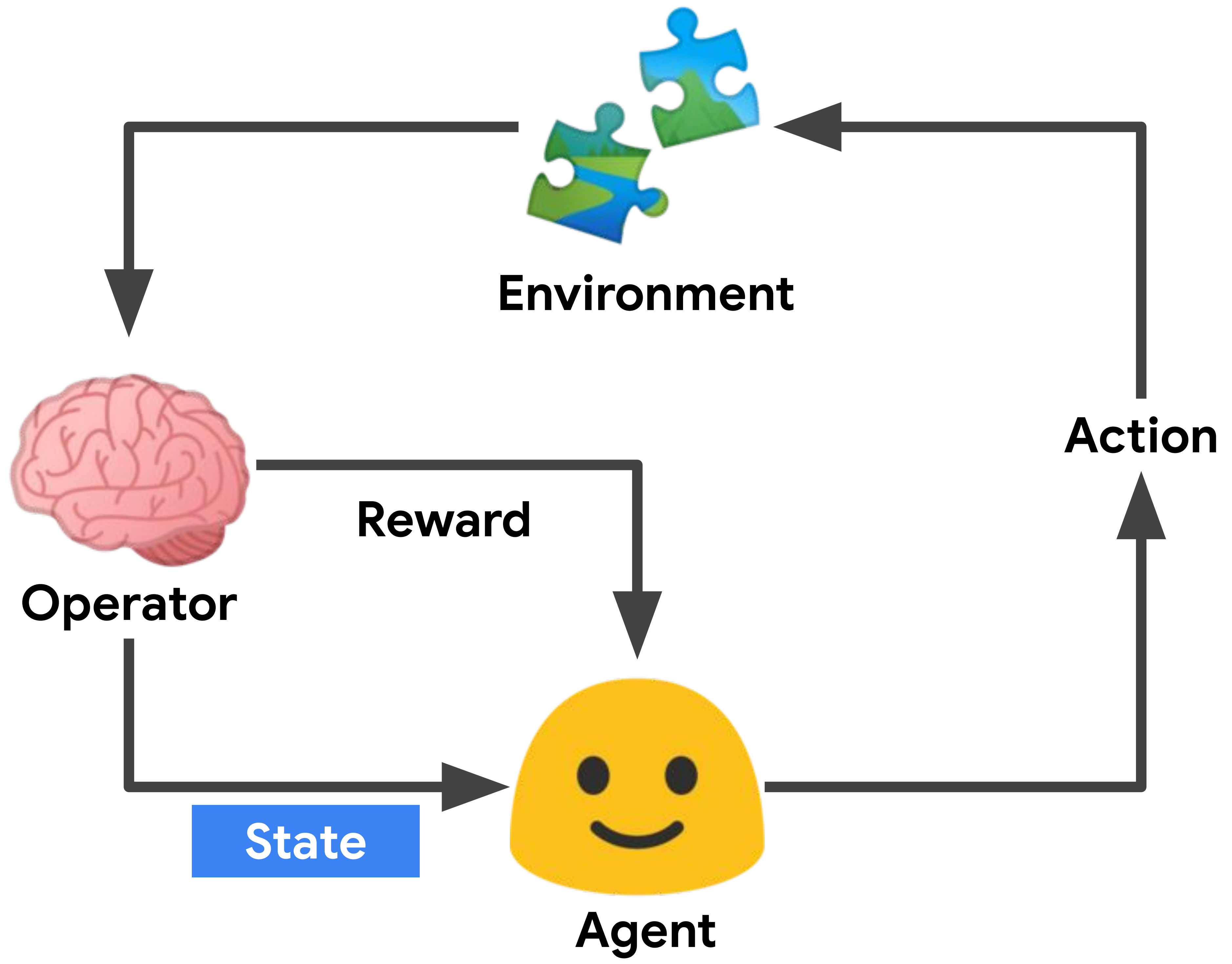
DRL Trading Algorithm Components

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



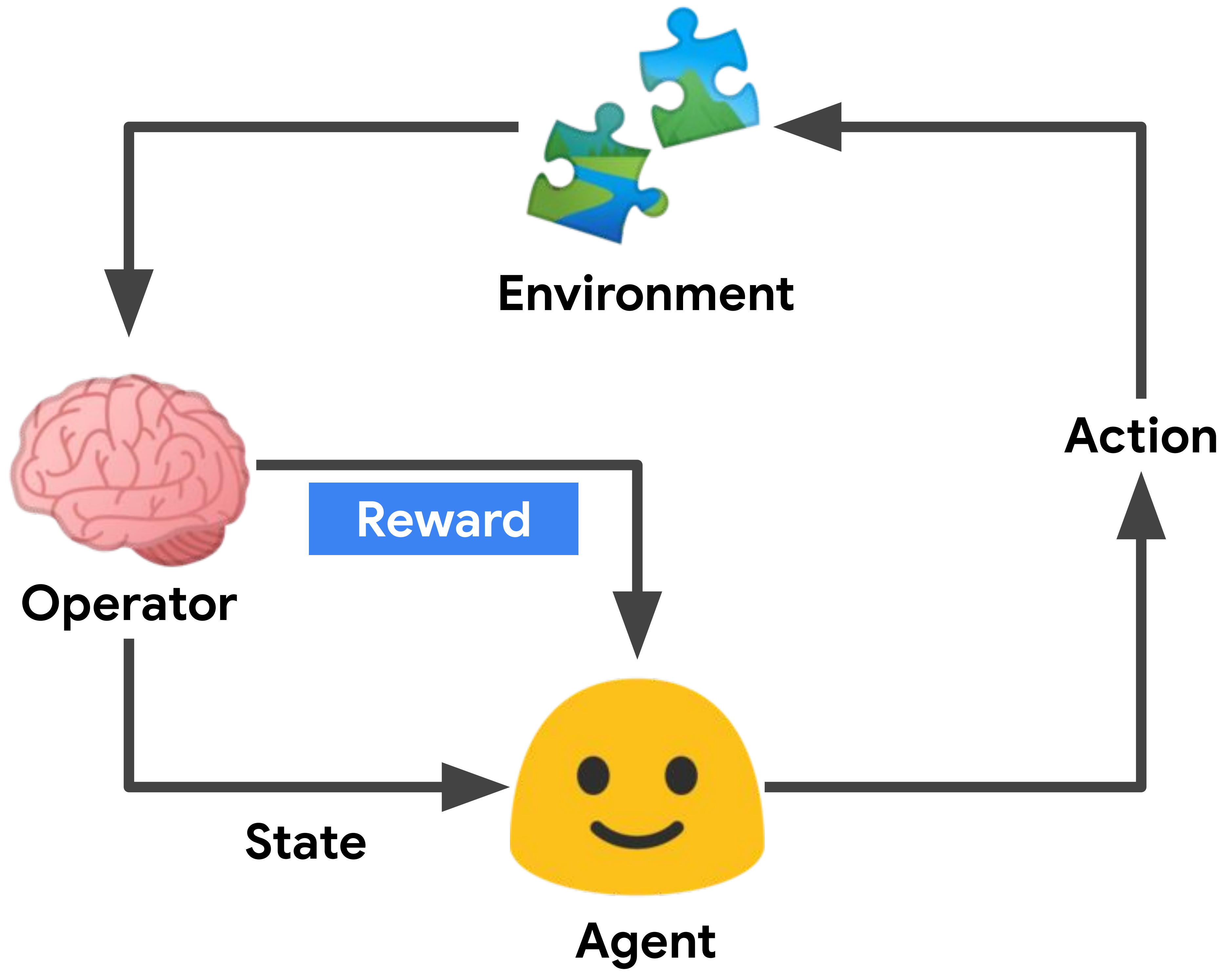
DRL Trading Algorithm Components

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



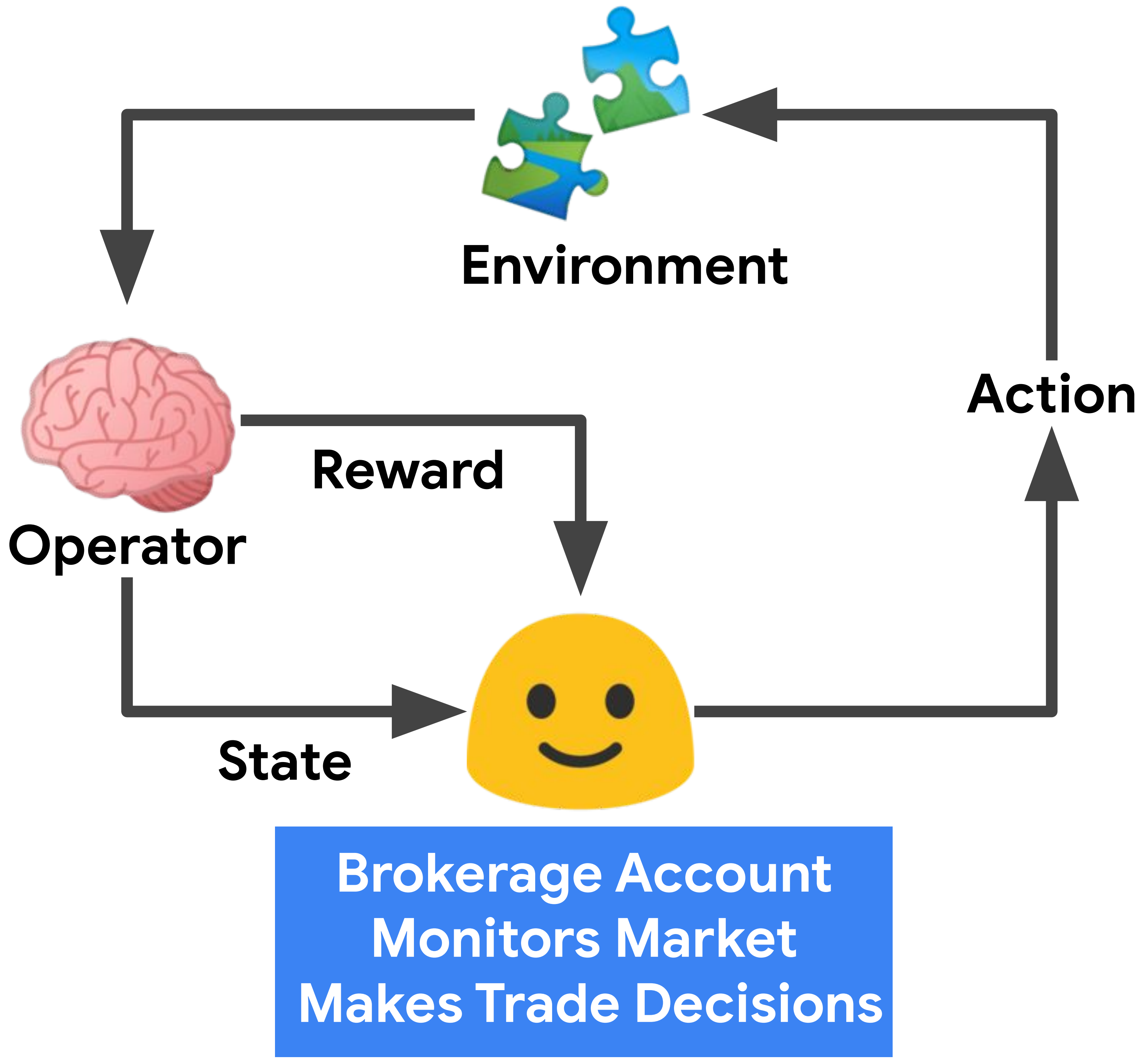
DRL Trading Algorithm Components

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



DRL Agent

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



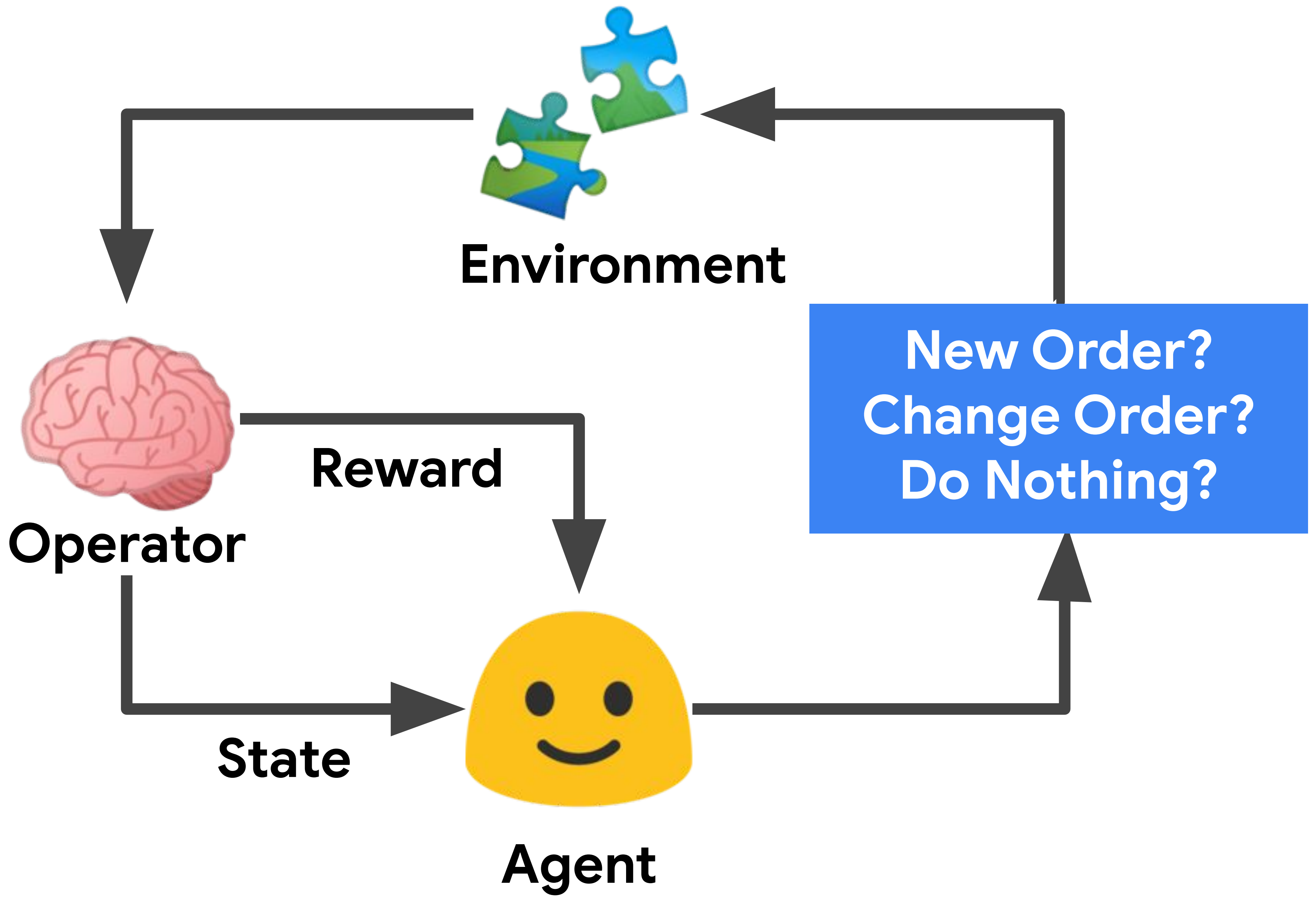
Agent/Algo Methodology

1. Make Trading Decision ⇒ Order
   Filled or **Not Filled?**
2. Assess New Market Conditions
3. Make Decision

   ⇒ New Order?

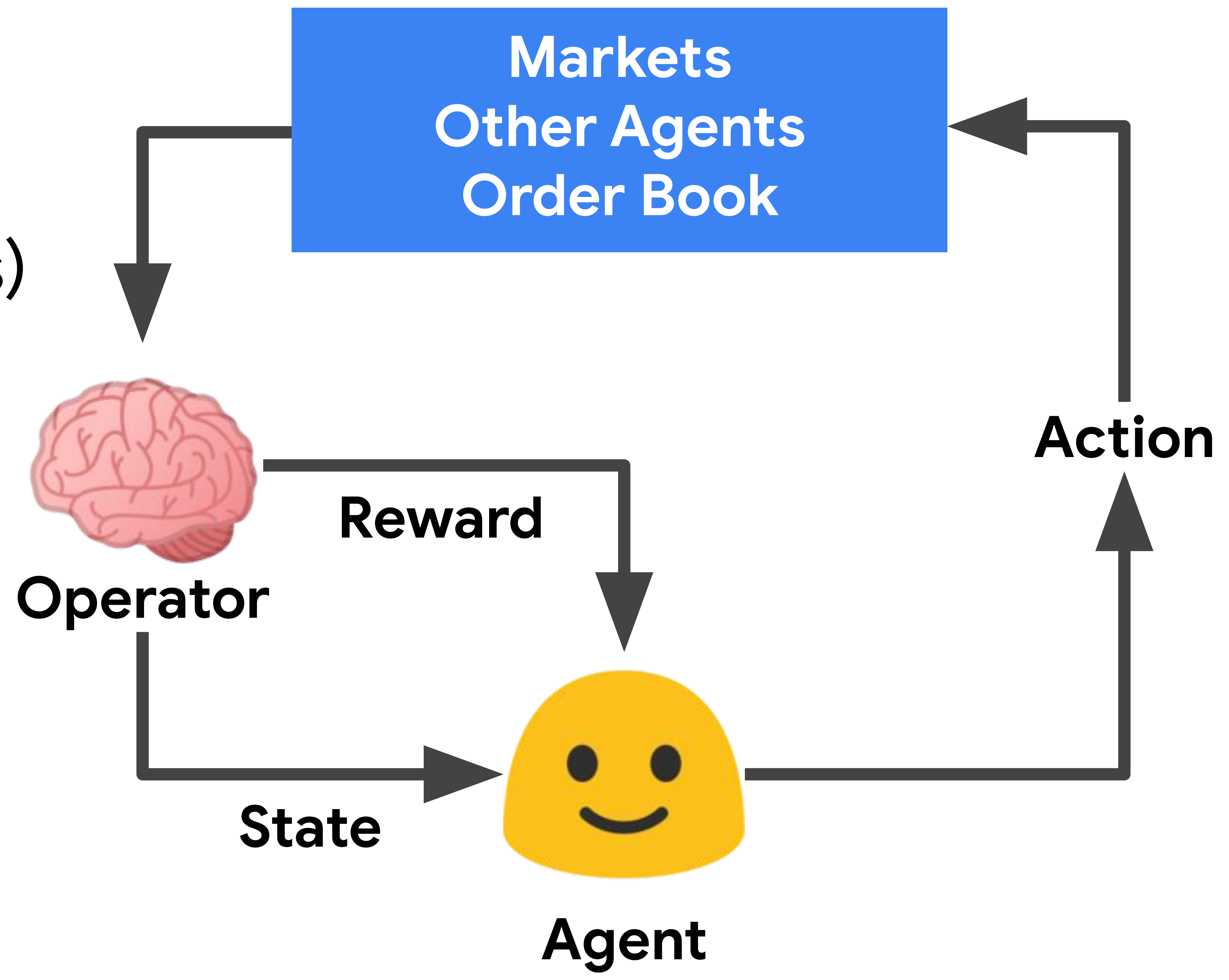
   ⇒ Change Order?

   ⇒ Do nothing?



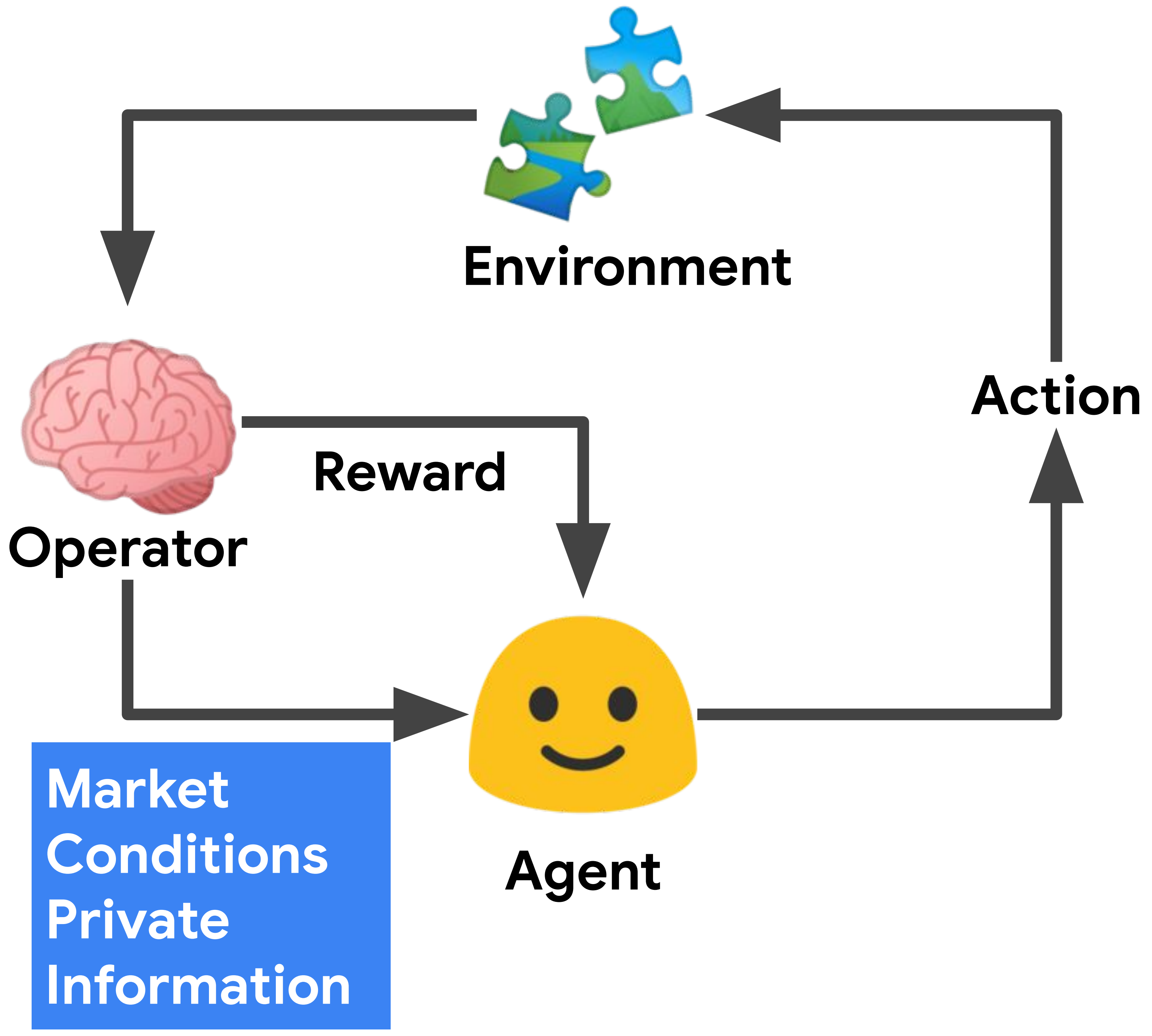
DRL Environment

- 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



DRL Reward

- Specification is key to the success of trading algo

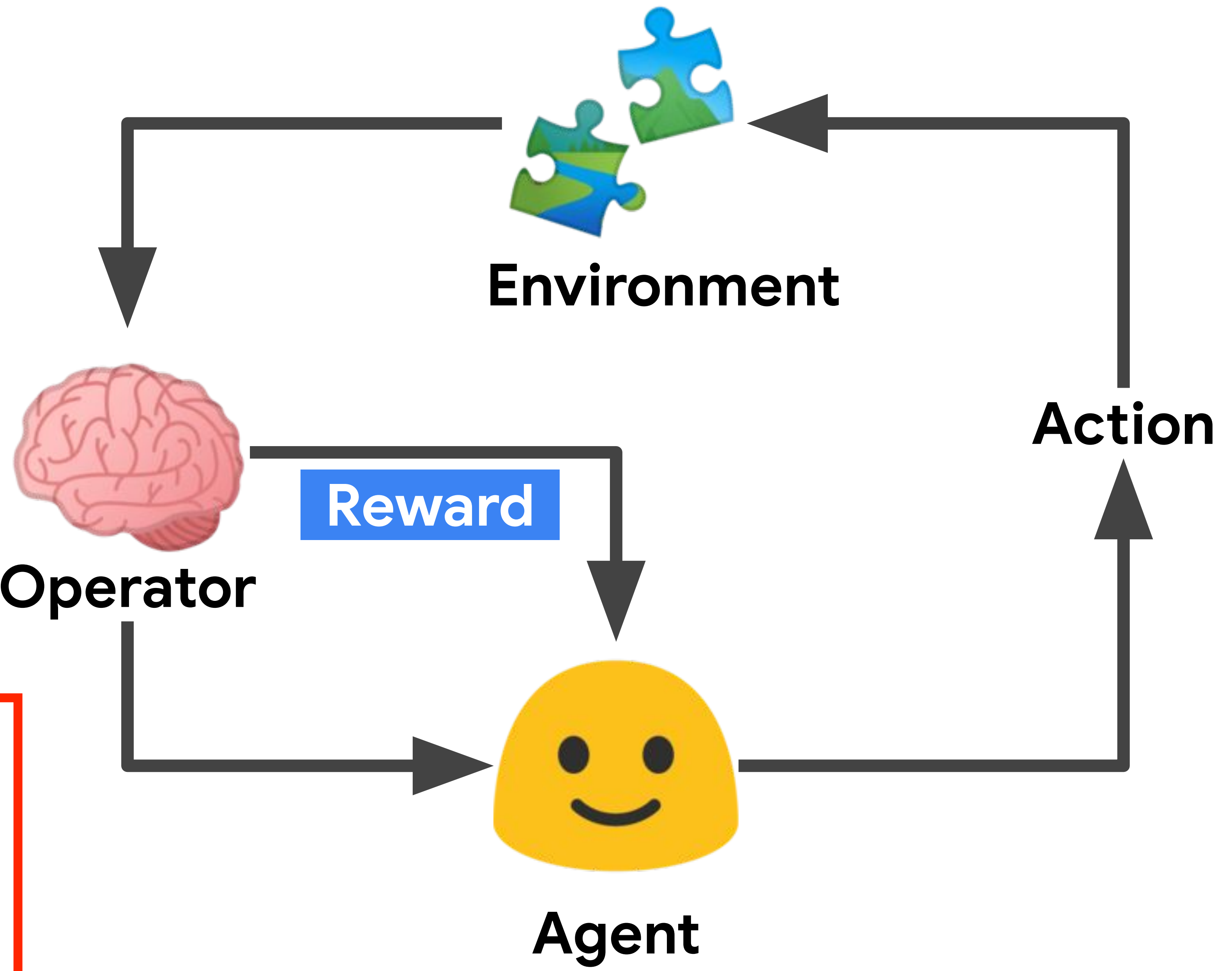
  Absolute Reward Maximization

  ⇒ High PnL Volatility

  ⇒ Unmanageable Drawdowns

- Optimization default is **Sharpe Ratio:**

  Strategy Return / PnL Volatility



In reality, traders strive for an optimal Sharpe ratio, which has proven to be the most efficient reward goal for DRL algorithms.

File Name:
T-AIFORF-I-p3_M1_l11_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 it 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