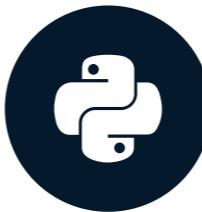


Markov Decision Processes

REINFORCEMENT LEARNING WITH GYMNASIUM IN PYTHON



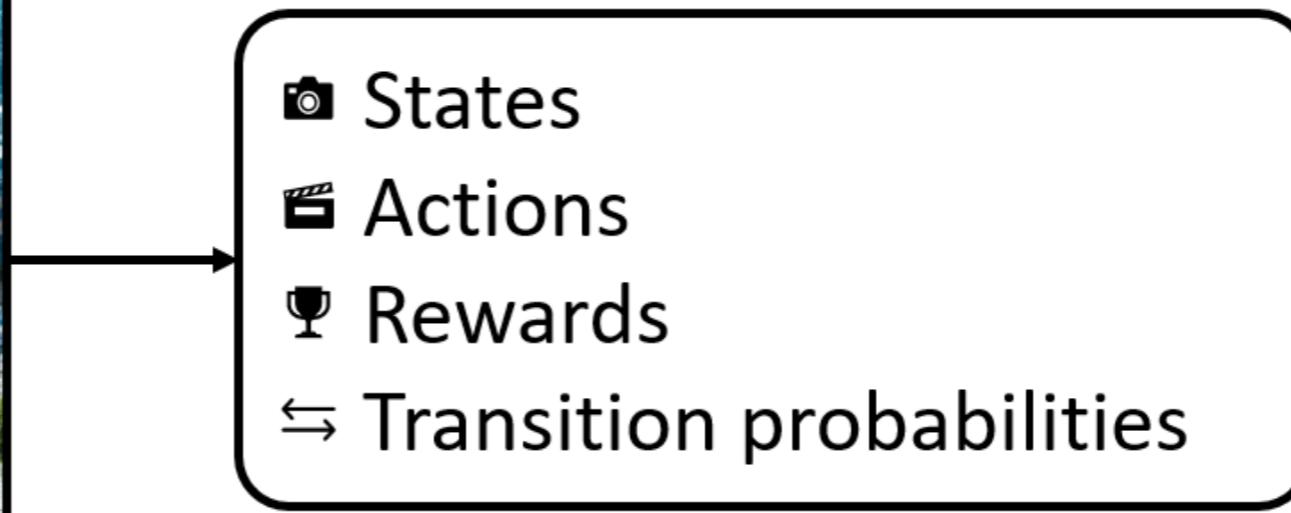
Fouad Trad
Machine Learning Engineer

MDP

- Models RL environments mathematically



Complex environment

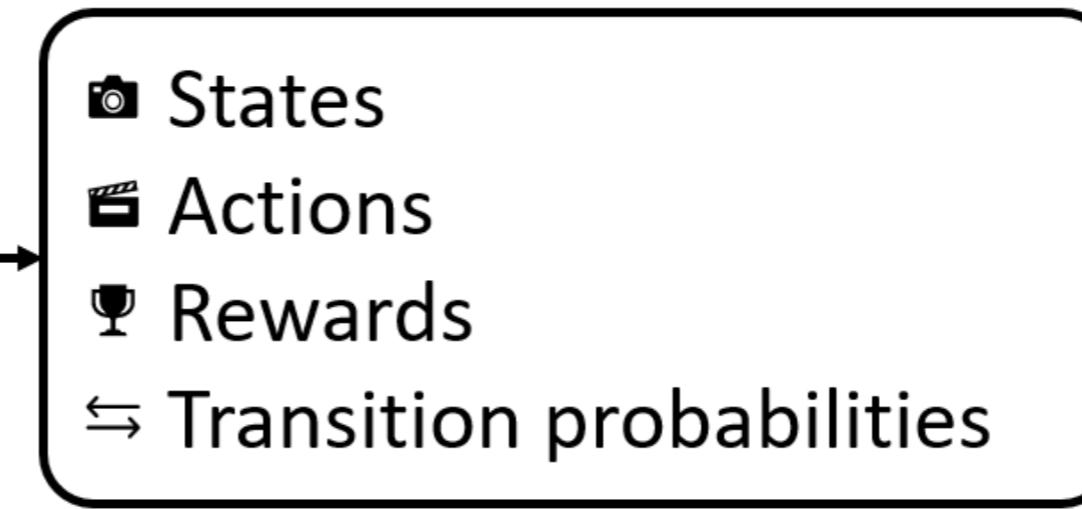


MDP

- Models RL environments mathematically



Complex environment



MDP

*Solve with
model-based
RL techniques*

Markov property

- Future state depends only on *current state* and *action*



Frozen Lake as MDP

- Agent must reach goal without falling into holes



Frozen Lake as MDP - states

- Positions agent can occupy



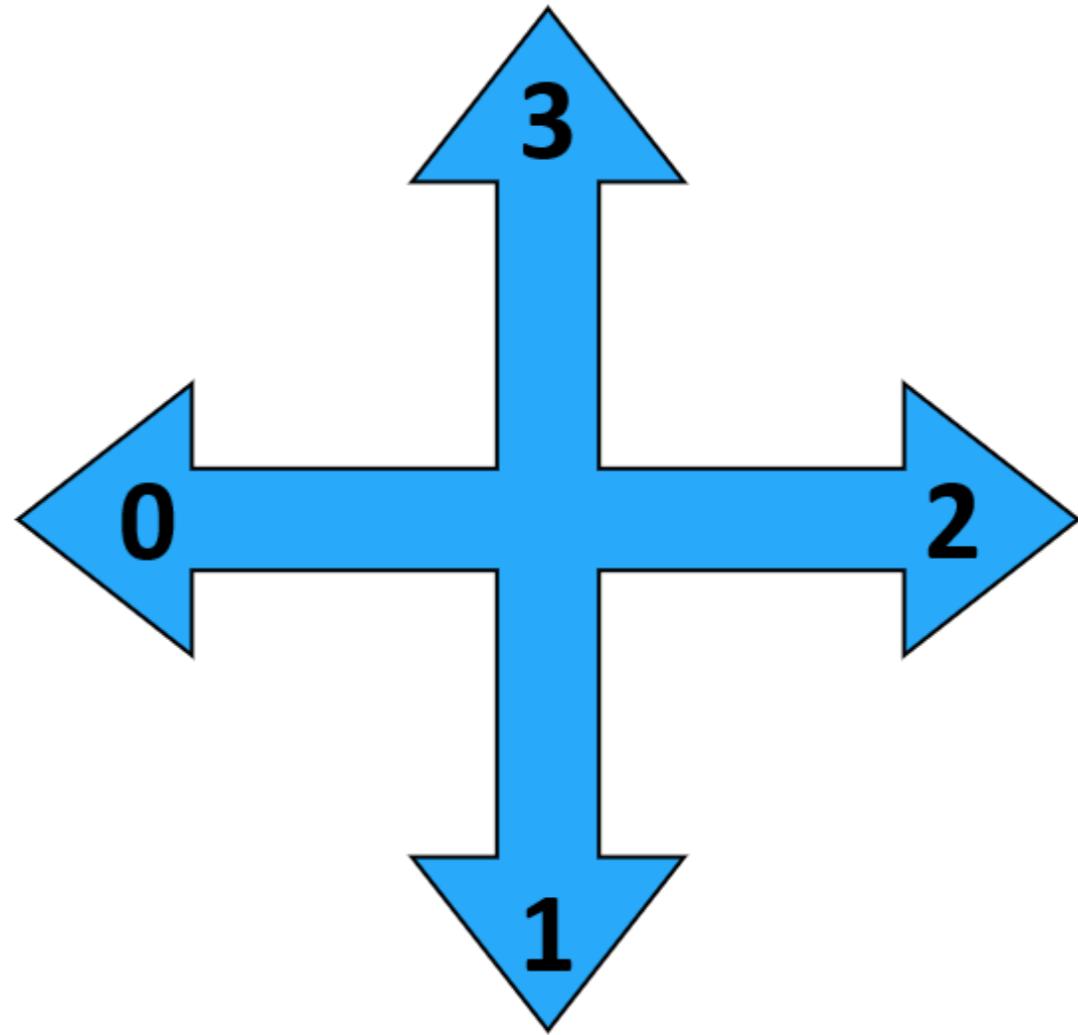
Frozen Lake as MDP - terminal states

- Lead to episode termination



Frozen Lake as MDP - actions

- Up, down, left, right



Frozen Lake as MDP - transitions

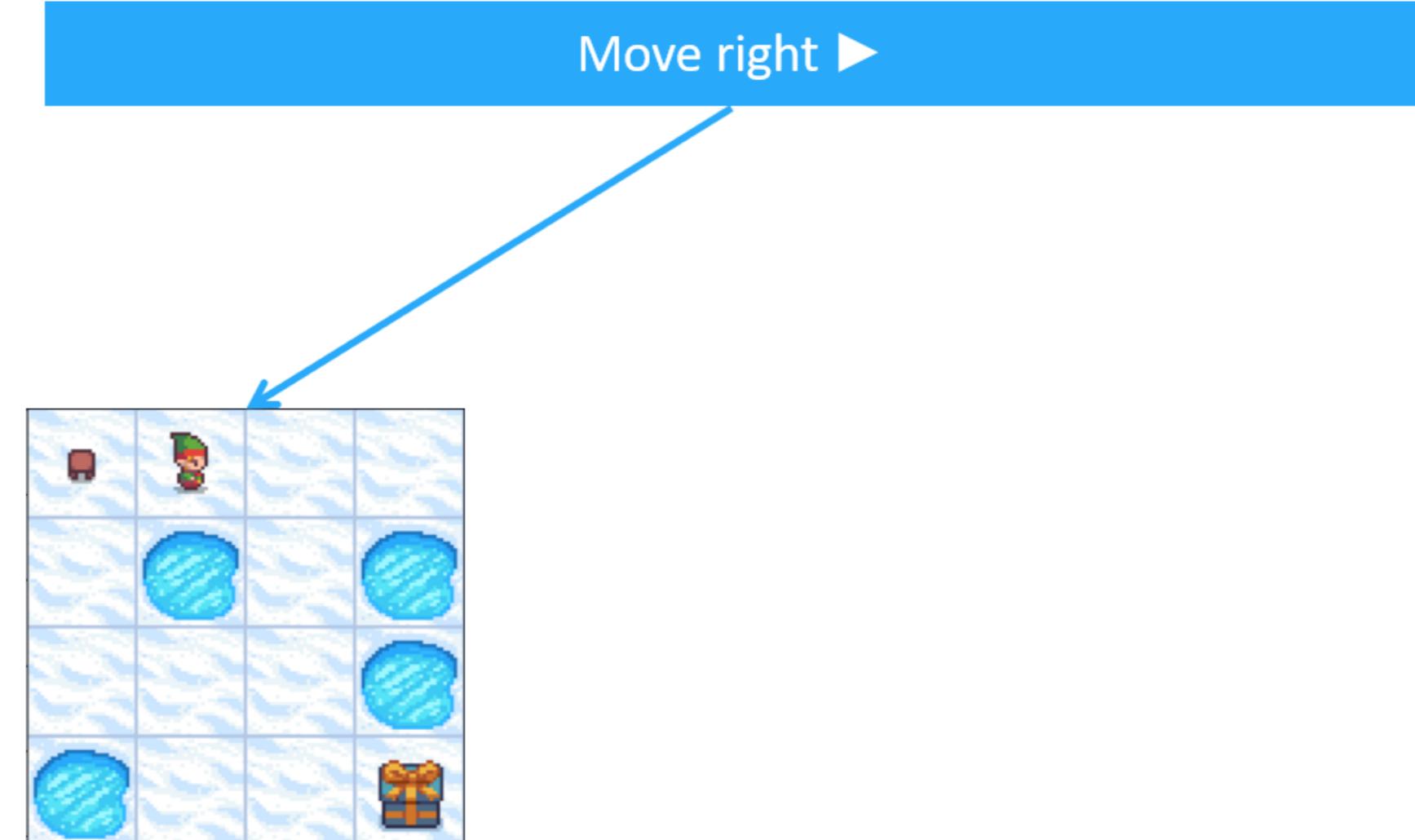
- Actions don't necessarily lead to expected outcomes



Move right ►

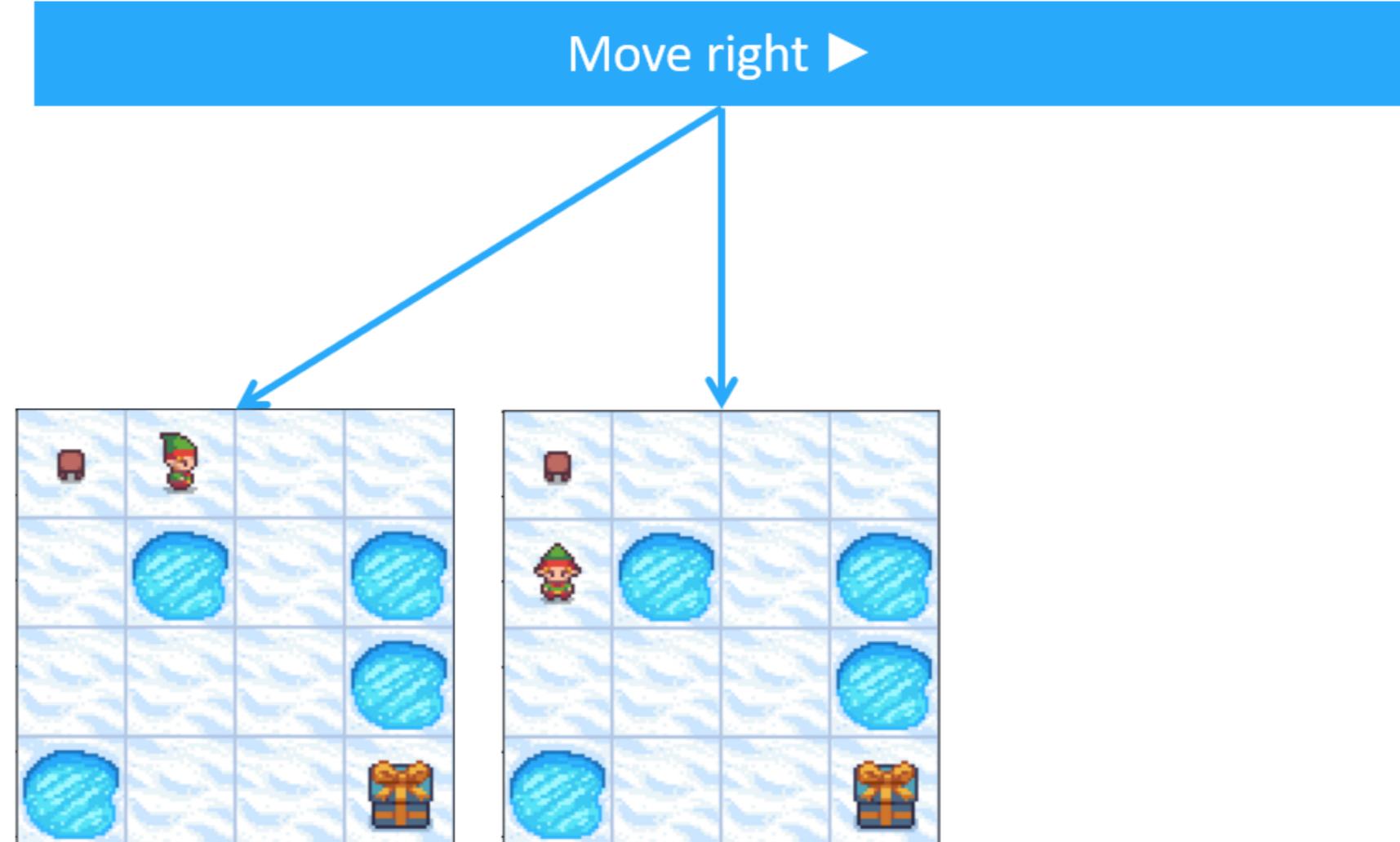
Frozen Lake as MDP - transitions

- Actions don't necessarily lead to expected outcomes



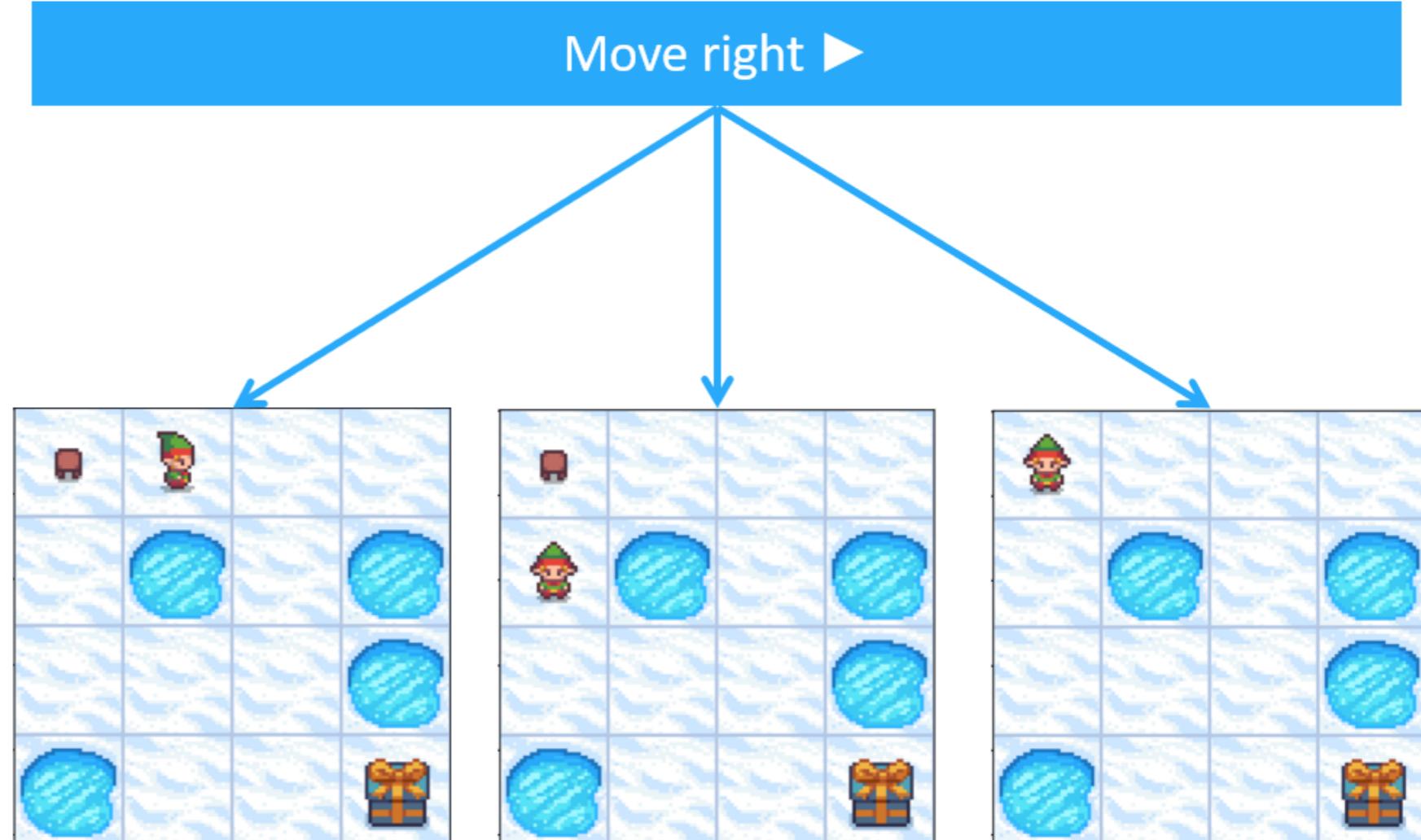
Frozen Lake as MDP - transitions

- Actions don't necessarily lead to expected outcomes



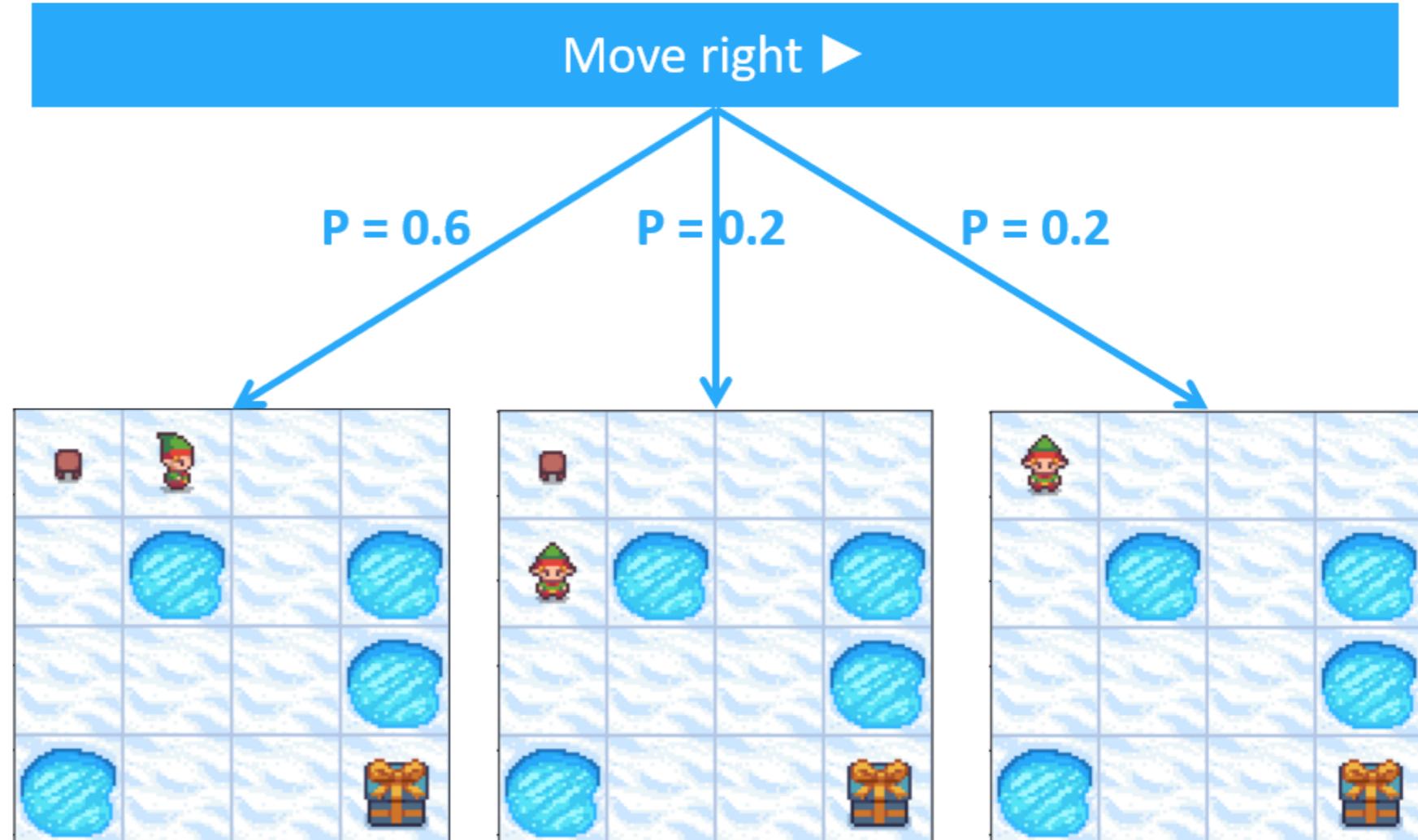
Frozen Lake as MDP - transitions

- Actions don't necessarily lead to expected outcomes



Frozen Lake as MDP - transitions

- Actions don't necessarily lead to expected outcomes



- Transition probabilities: likelihood of reaching a state given a state and action

Frozen Lake as MDP - rewards

- Reward only given in goal state



Positive reward

Gymnasium states and actions

```
import gymnasium as gym

env = gym.make('FrozenLake', is_slippery=True)
print(env.action_space)
print(env.observation_space)
print("Number of actions: ", env.action_space.n)
print("Number of states: ", env.observation_space.n)
```

Discrete(4)

Discrete(16)

Number of actions: 4

Number of states: 16

Gymnasium rewards and transitions

`env.unwrapped.P` : dictionary where keys are state-action pairs

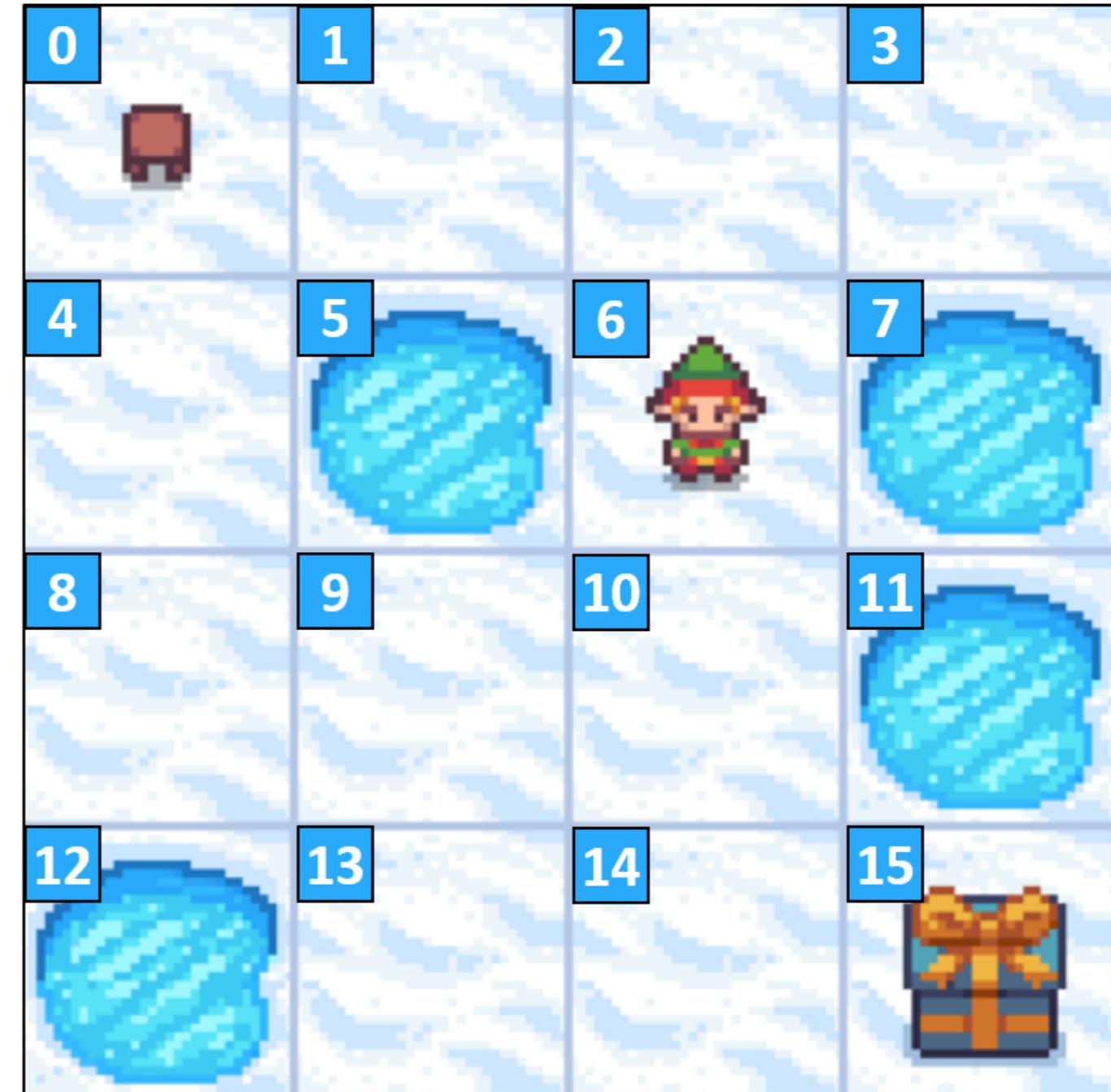
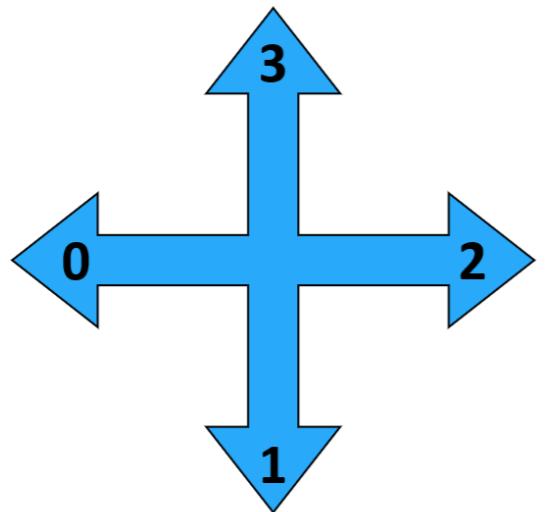
```
print(env.unwrapped.P[state][action])
```

```
[  
    (probability_1, next_state_1, reward_1, is_terminal_1),  
    (probability_2, next_state_2, reward_2, is_terminal_2),  
    etc.  
]
```

Gymnasium rewards and transitions - example

```
state = 6  
action = 0  
print(env.unwrapped.P[state][action])
```

```
[(0.3333333333333333, 2, 0.0, False),  
(0.3333333333333333, 5, 0.0, True),  
(0.3333333333333333, 10, 0.0, False)]
```

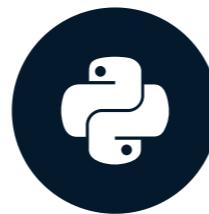


Let's practice!

REINFORCEMENT LEARNING WITH GYMNASIUM IN PYTHON

Policies and state-value functions

REINFORCEMENT LEARNING WITH GYMNASIUM IN PYTHON



Fouad Trad
Machine Learning Engineer

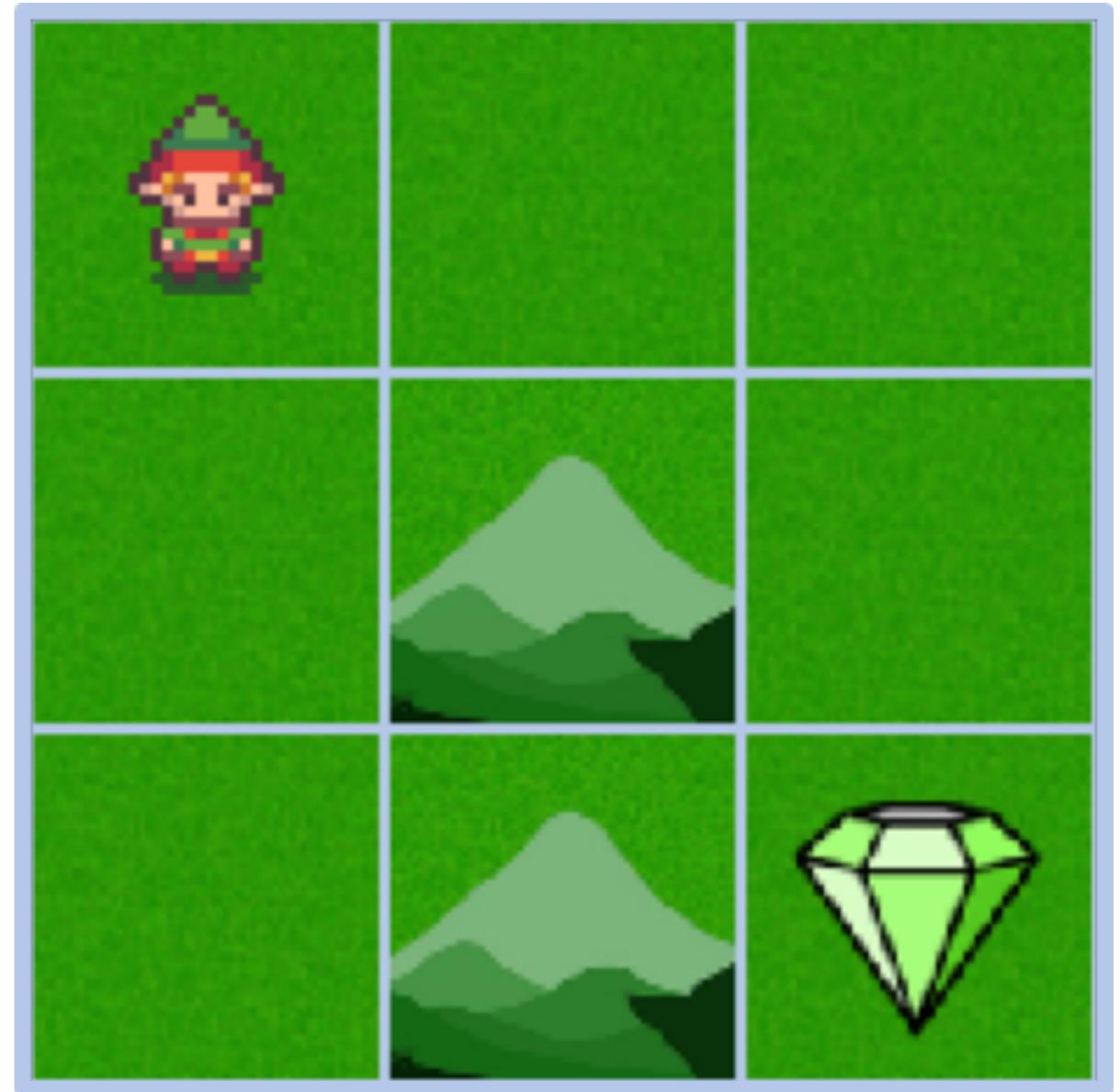
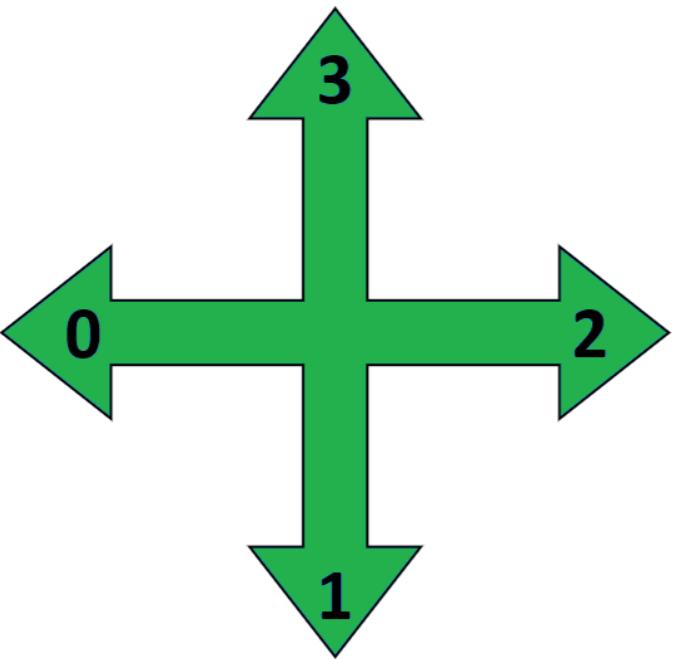
Policies

- RL objective → formulate effective policies
- Specify which action to take in each state to maximize return



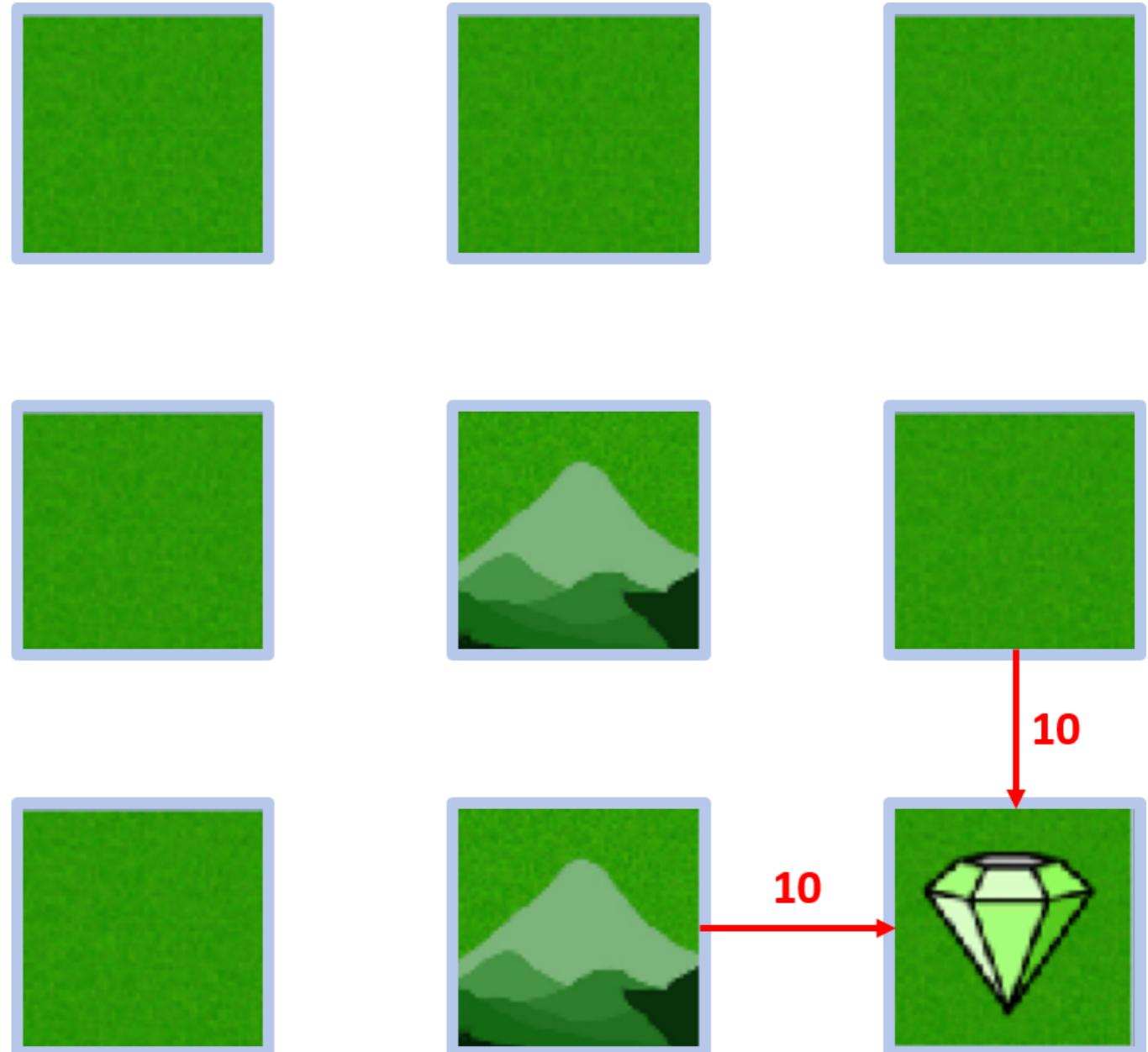
Grid world example

- Agent aims to reach diamond while avoiding mountains
- Nine states
- Deterministic movements



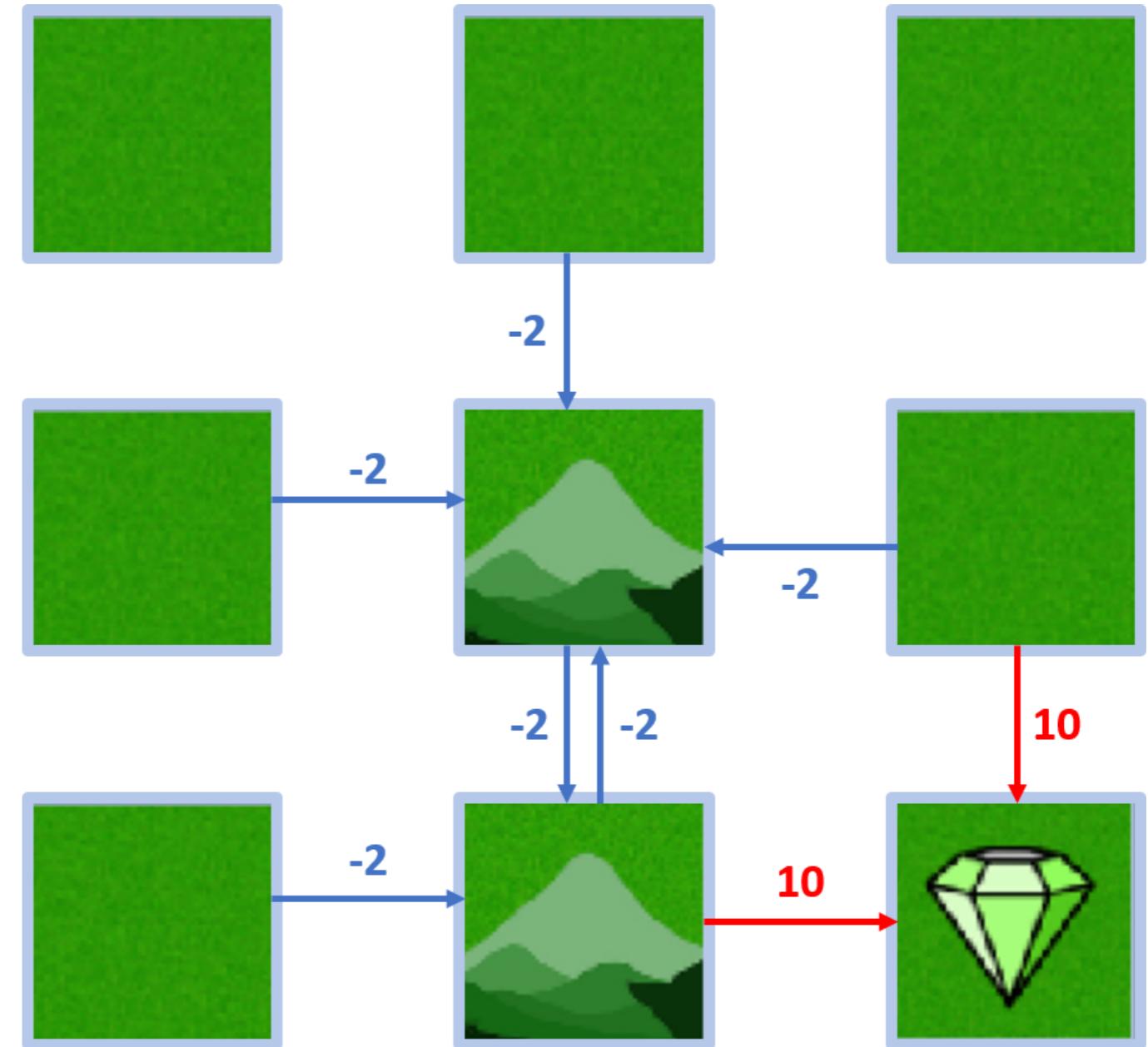
Grid world example - rewards

- Given based on states:
 - Diamond: +10



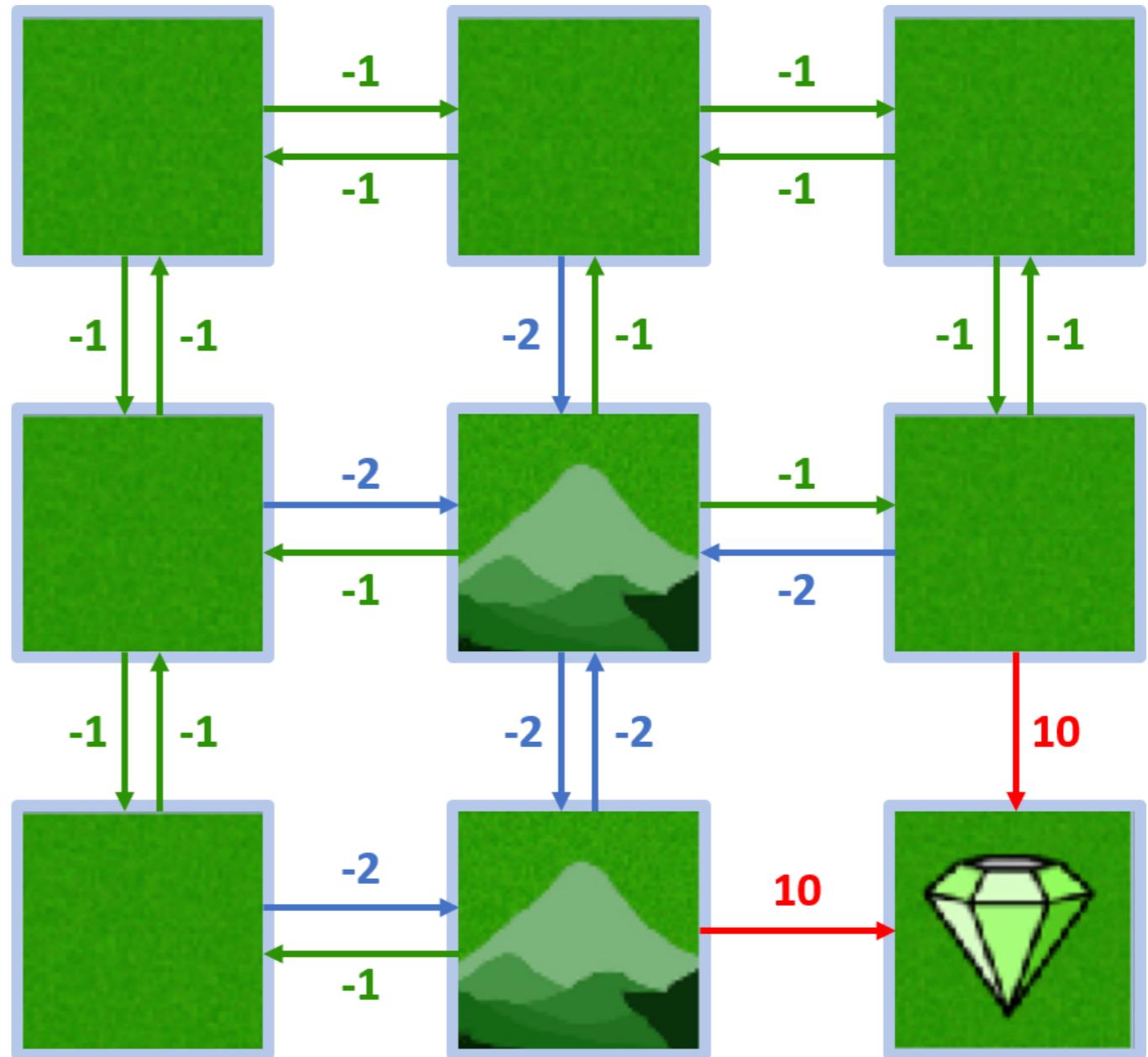
Grid world example - rewards

- Given based on states:
 - Diamond: +10
 - Mountain: -2



Grid world example - rewards

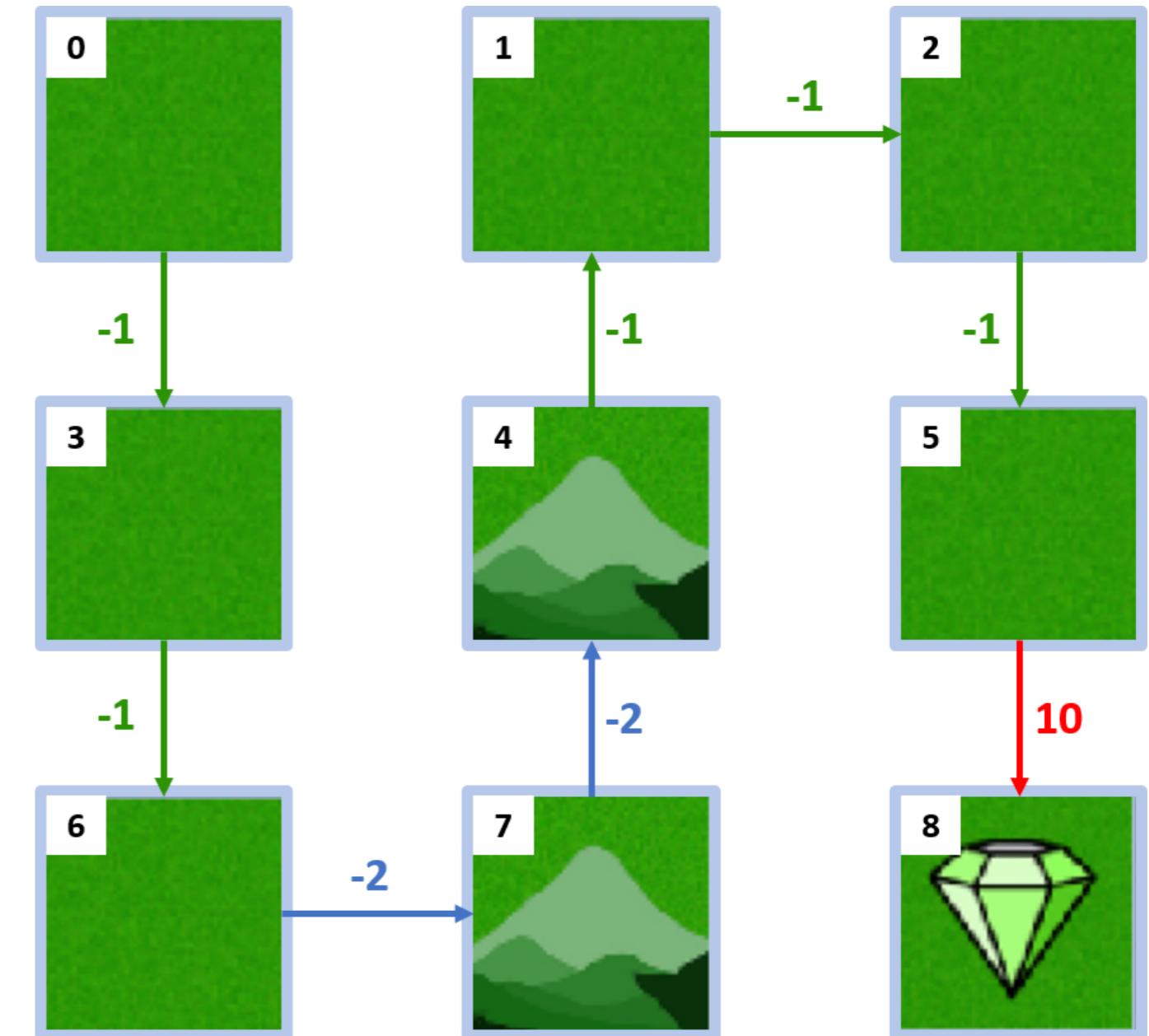
- Given based on states:
 - Diamond: +10
 - Mountain: -2
 - Other states: -1



Grid world example: policy

```
# 0: left, 1: down, 2: right, 3: up
policy = {
    0:1, 1:2, 2:1,
    3:1, 4:3, 5:1,
    6:2, 7:3
}

state, info = env.reset()
terminated = False
while not terminated:
    action = policy[state]
    state, reward, terminated, _, _ = env.step(action)
```



State-value functions

- Estimate the state's worth
- Expected return starting from state, following policy

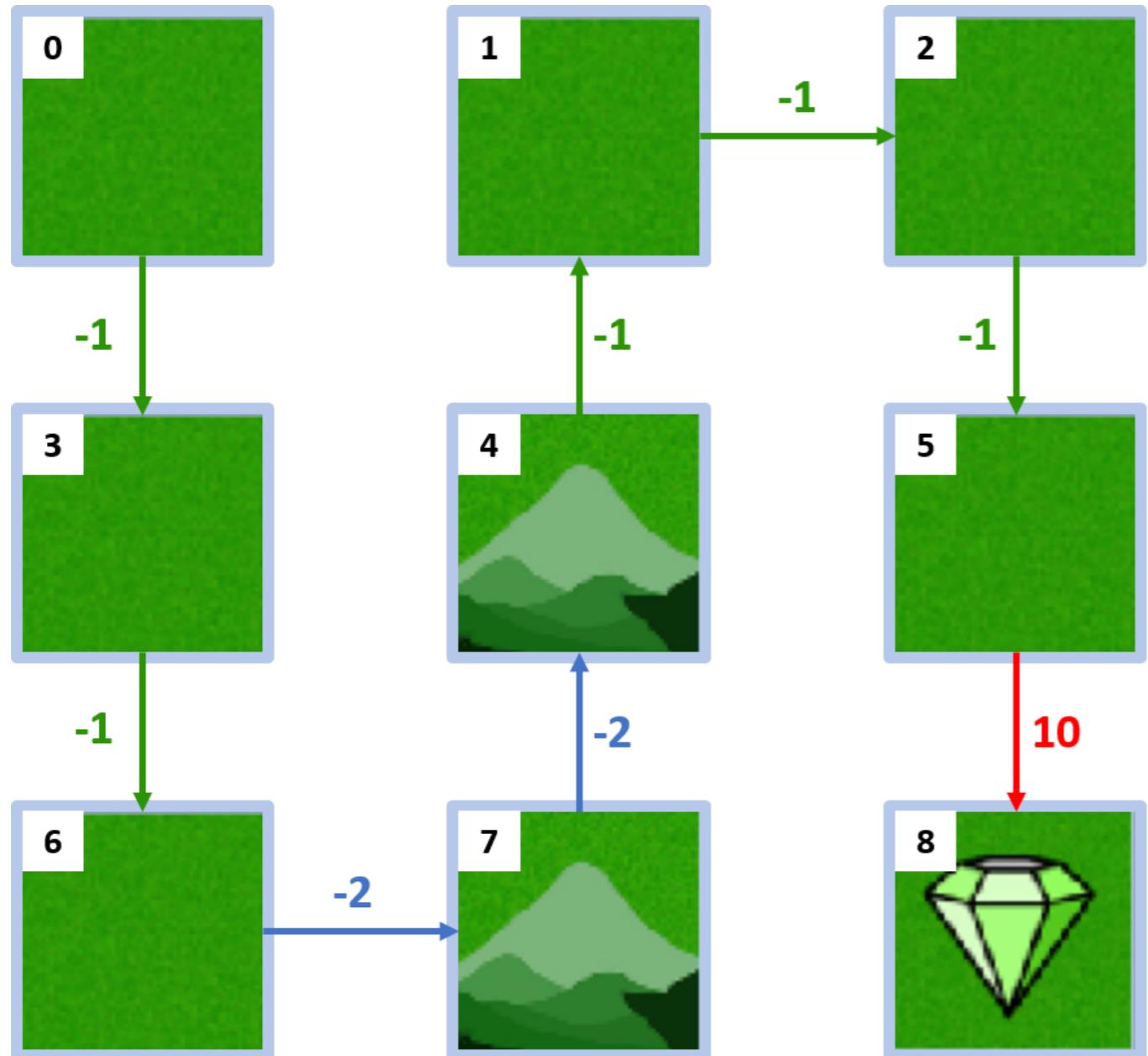
$$V(s) = \underline{r_{s+1} + \gamma r_{s+2} + \gamma^2 r_{s+3} + \cdots + \gamma^{n-1} r_{s+n}}$$

State – value of s

Sum of discounted rewards collected by

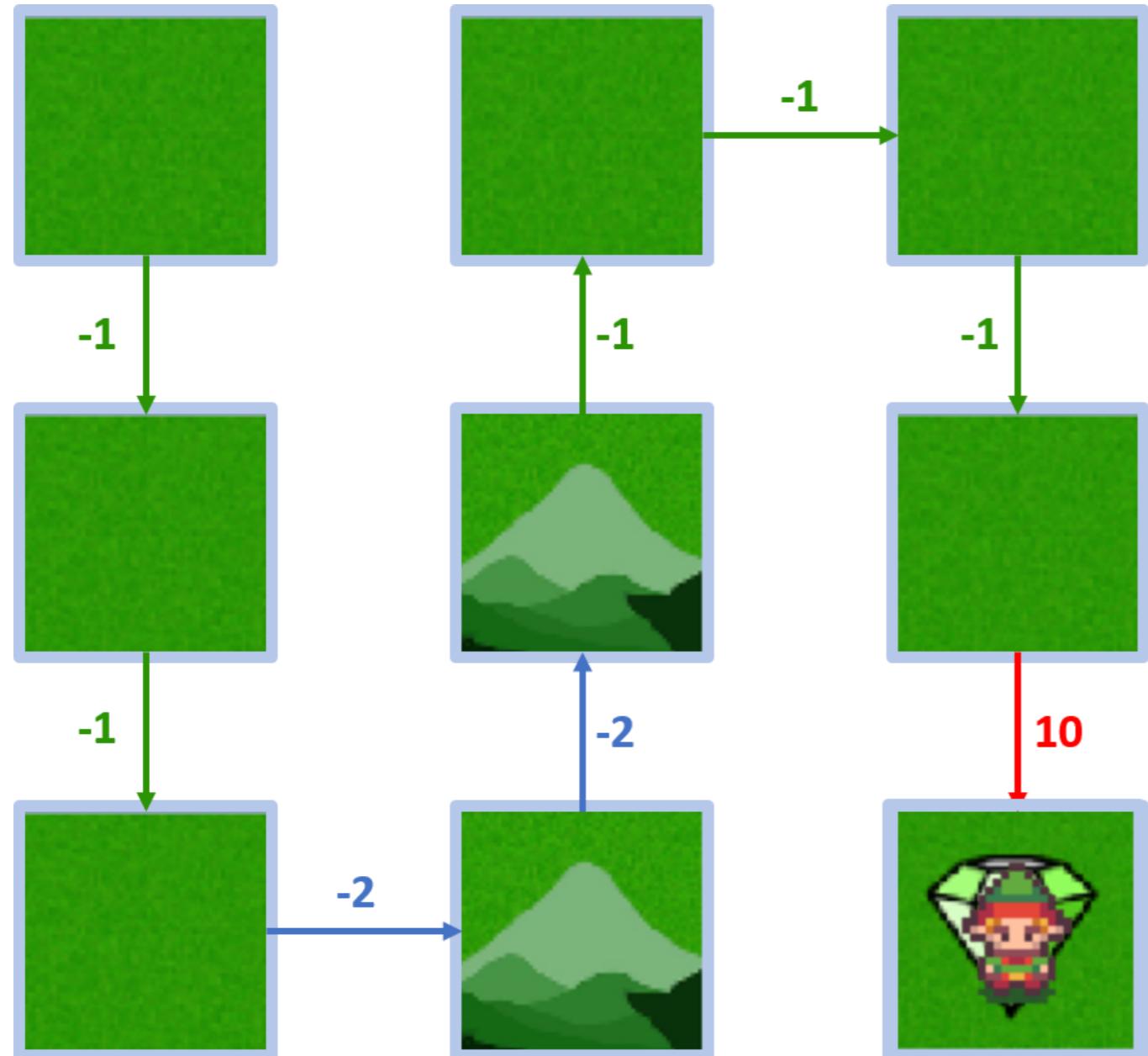
- starting in state s
- and following the policy

Grid world example: State-values

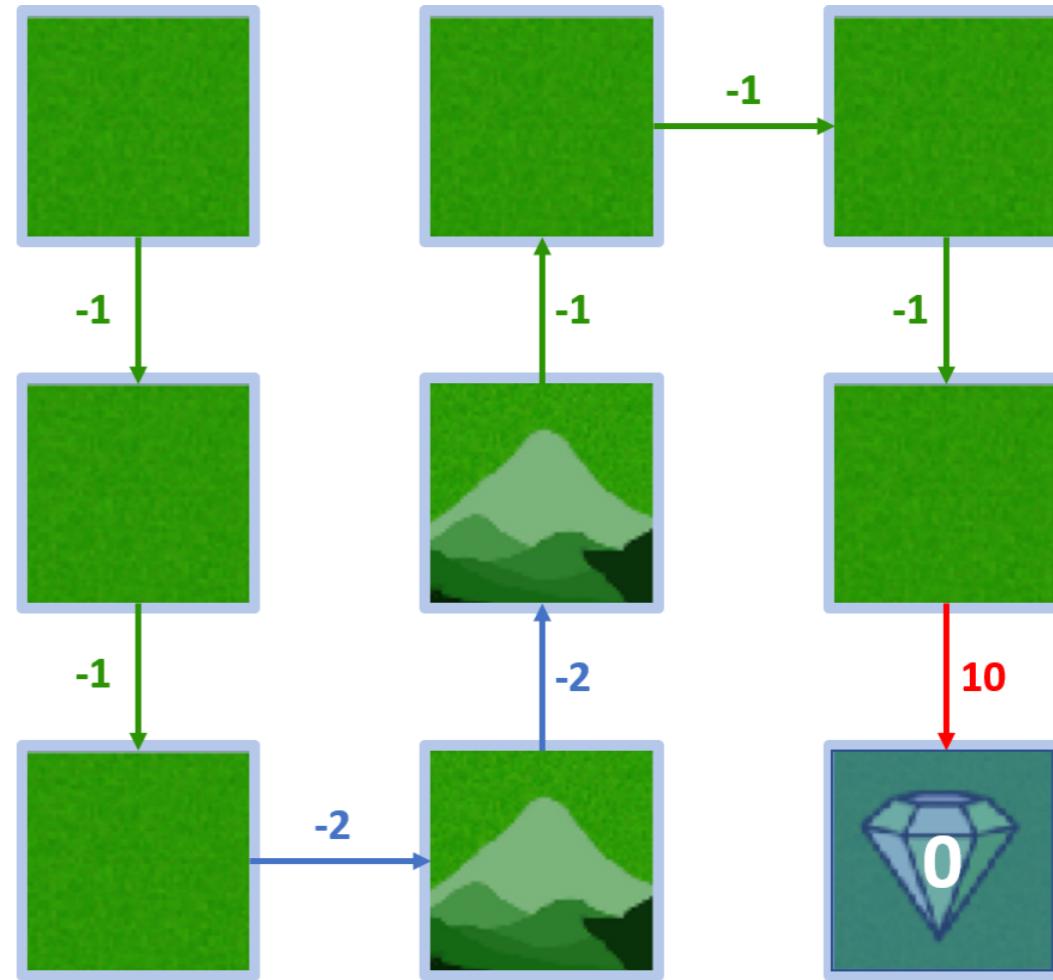


- Nine states → nine state-values
- Discount factor: $\gamma = 1$

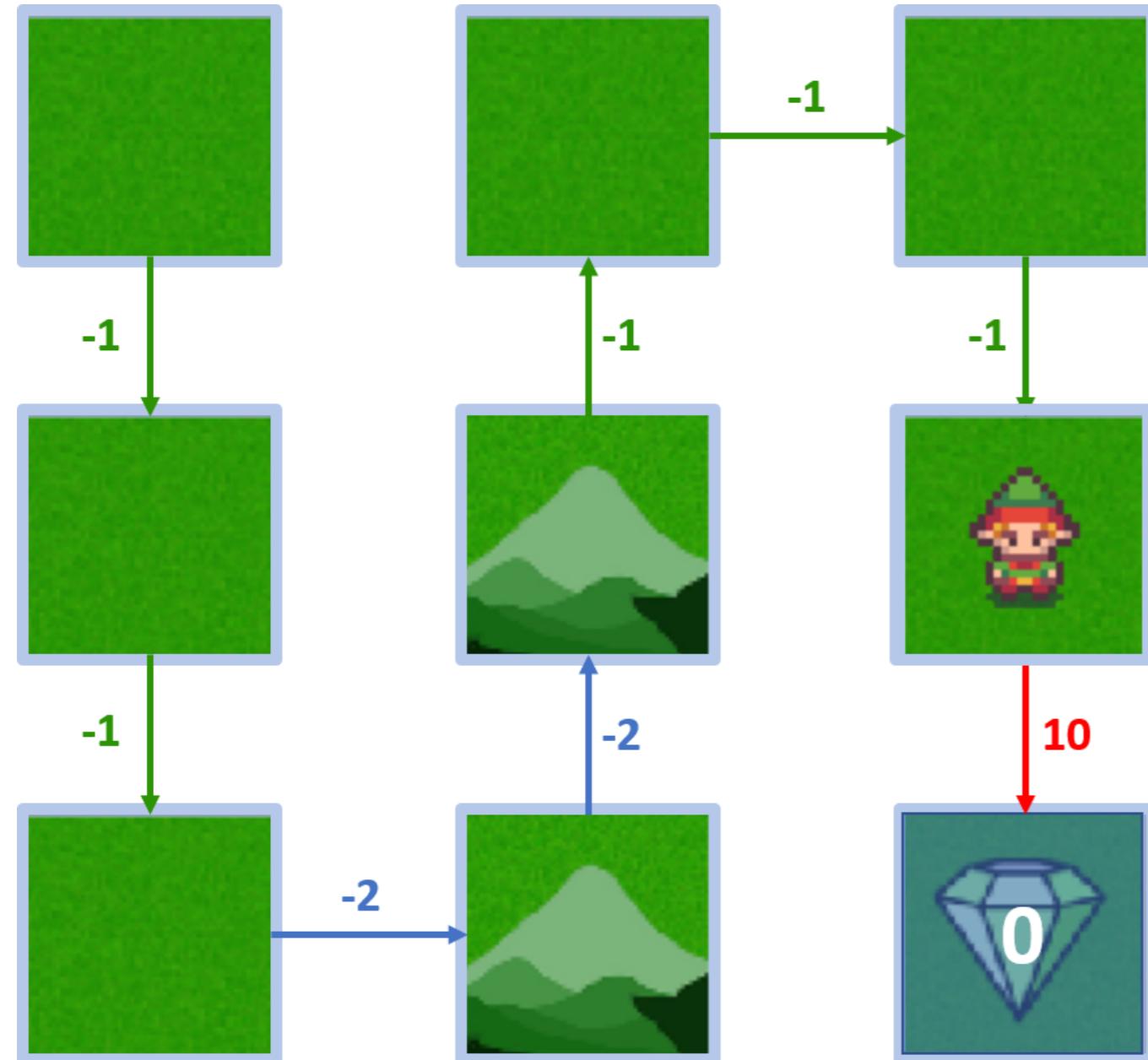
Value of goal state



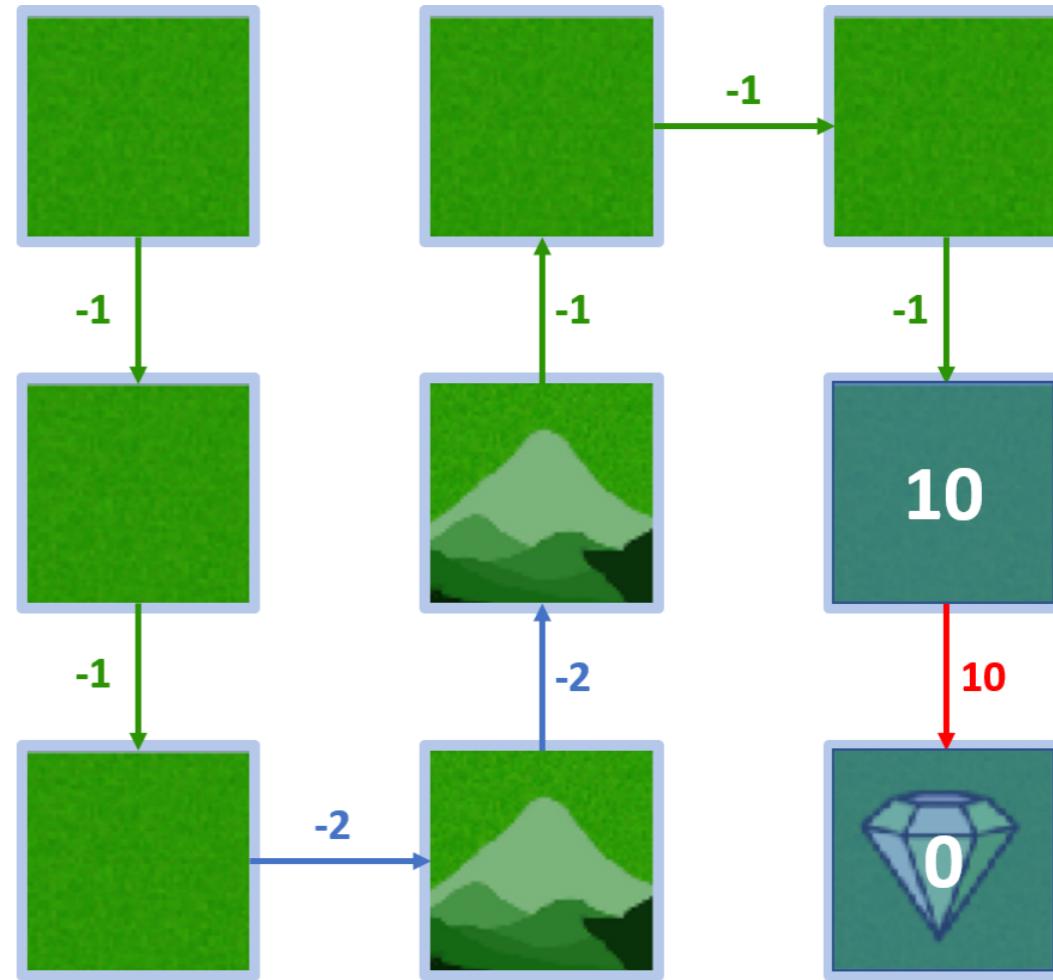
- Starting in goal state, agent doesn't move
- $V(goal\ state) = 0$



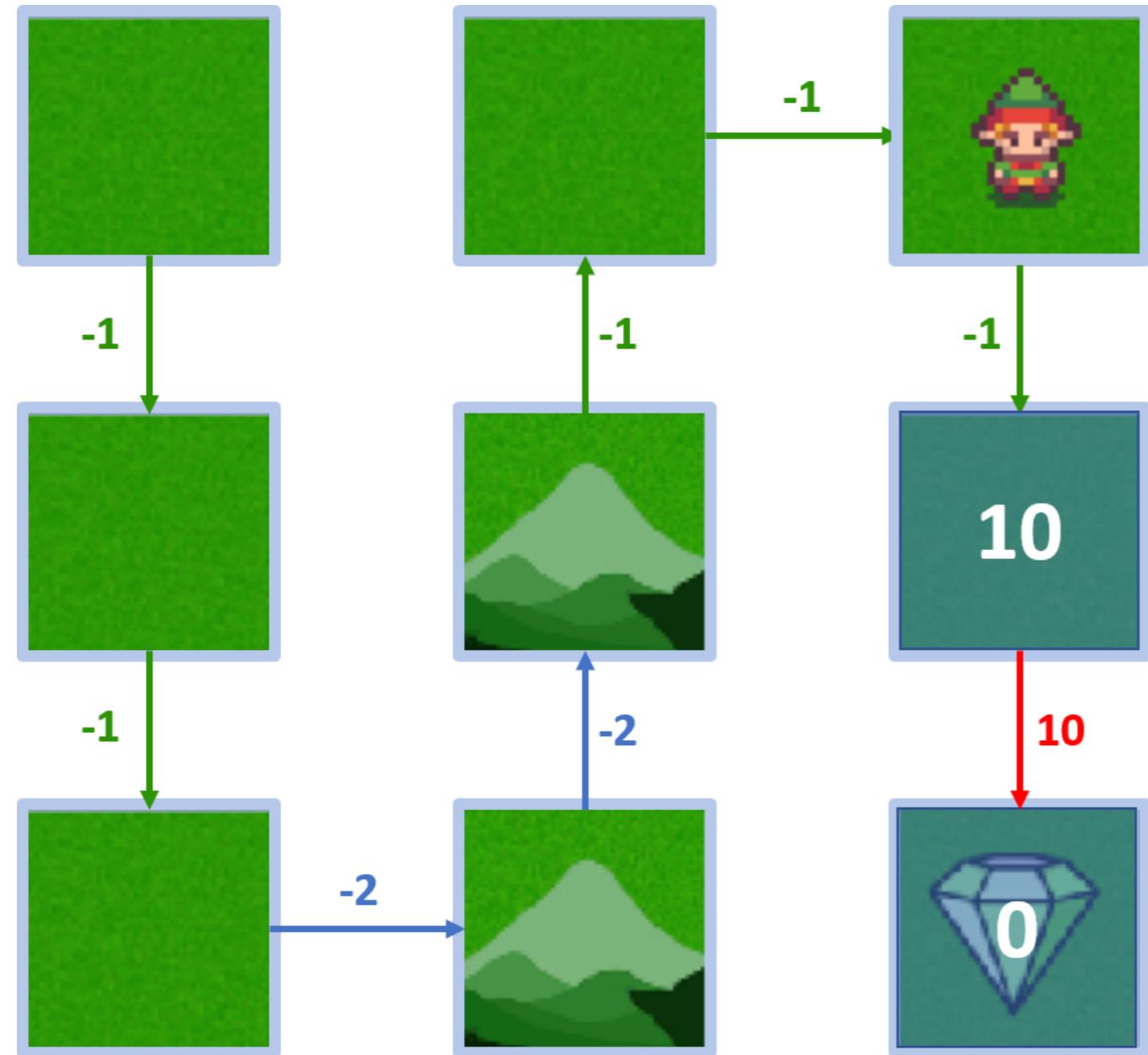
Value of state 5



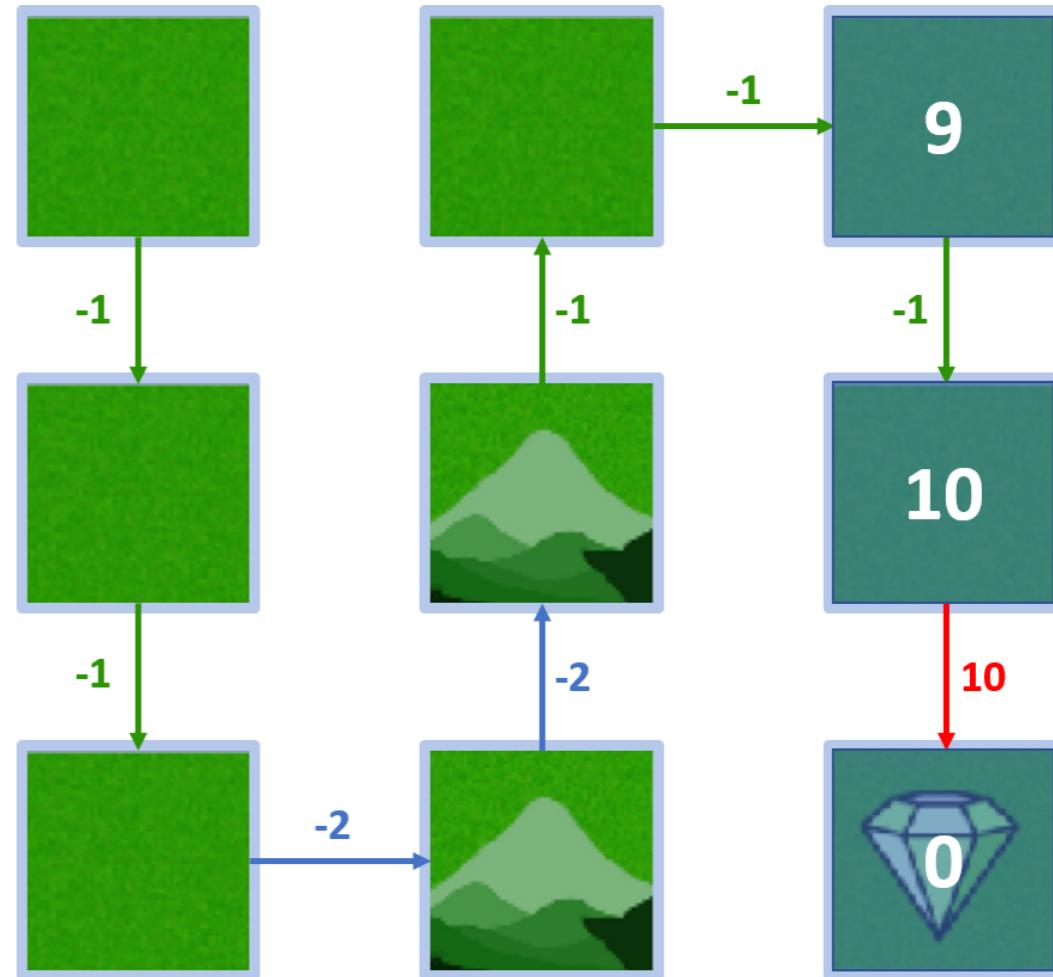
- Starting in 5, agent moves to goal
- $V(5) = 10$



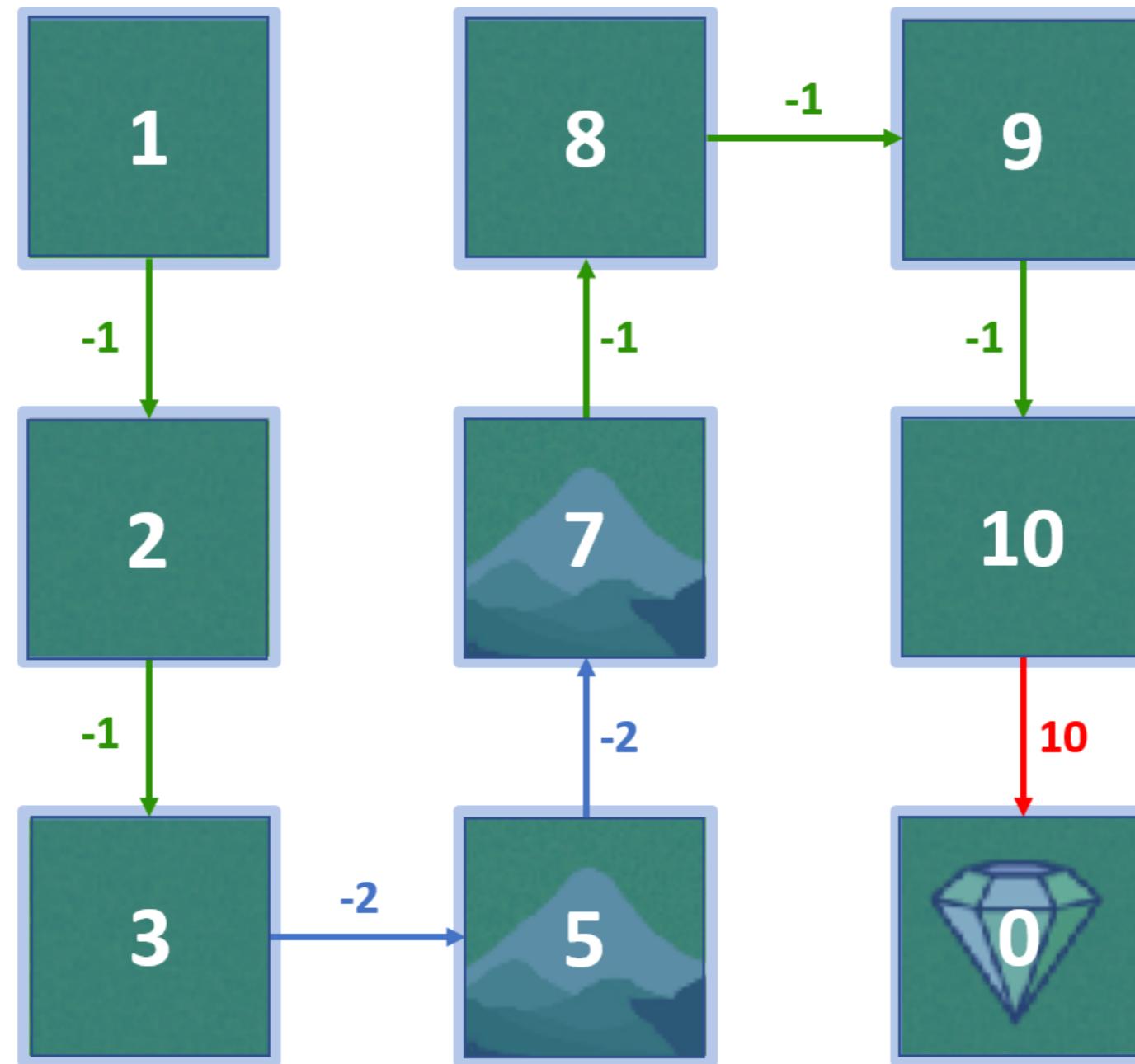
Value of state 2



- Starting in 2, rewards: $-1, 10$
- $V(2) = (1 \times -1) + (1 \times 10) = 9$



All state values



Bellman equation

- Recursive formula
- Computes state-values

$$\underline{V(s) = r_{s+1} + \gamma V(s + 1)}$$

*State – value
of s*

Sum of:
– immediate reward
– discounted value of the next state

Computing state-values

```
def compute_state_value(state):
    if state == terminal_state:
        return 0

    action = policy[state]
    _, next_state, reward, _ = env.unwrapped.P[state][action][0]
    return reward + gamma * compute_state_value(next_state)
```

$$\underline{V(s) = r_{s+1} + \gamma V(s + 1)}$$

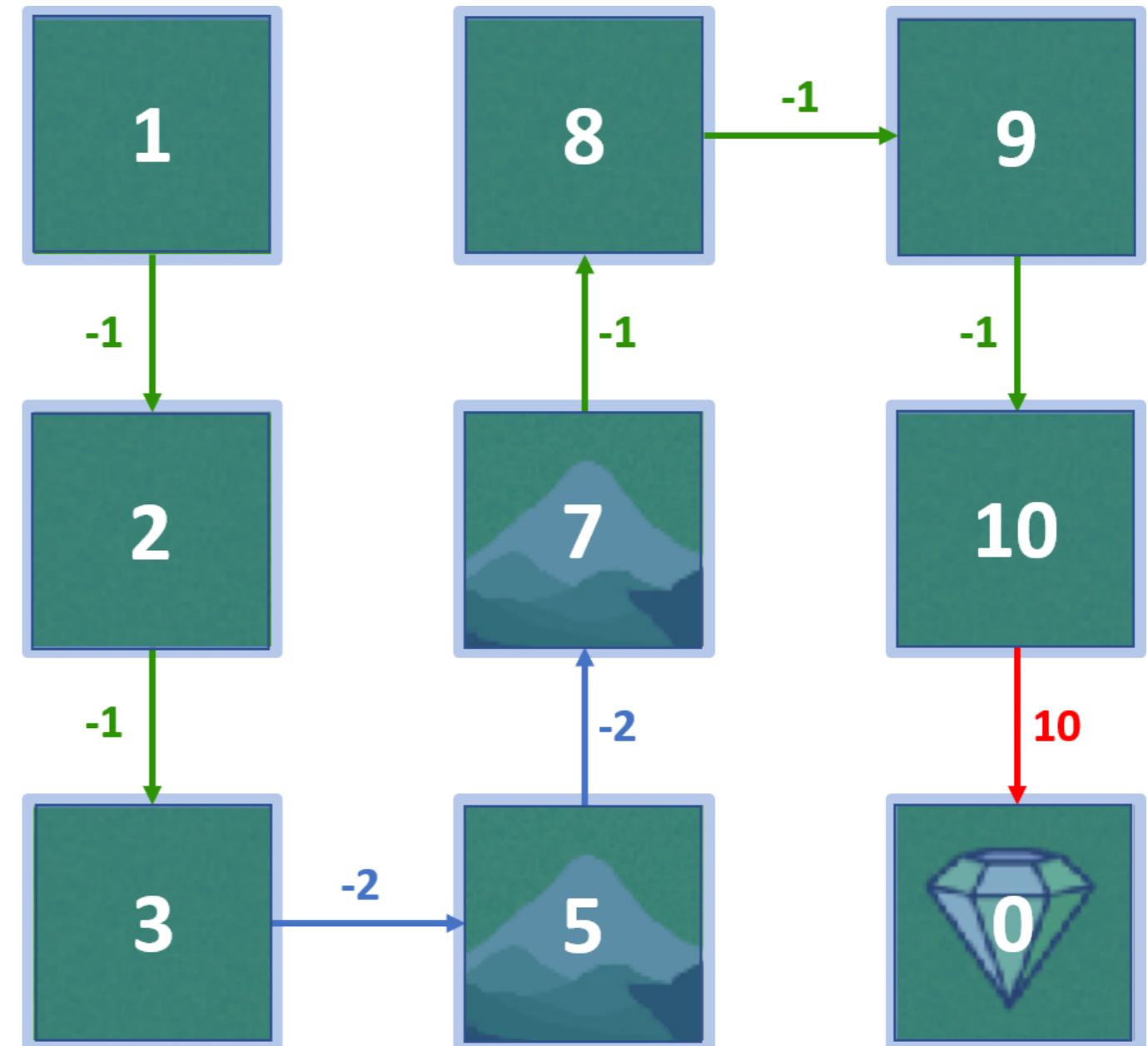
*State – value
of s*

Sum of:
– immediate reward
– discounted value of the next state

Computing state-values

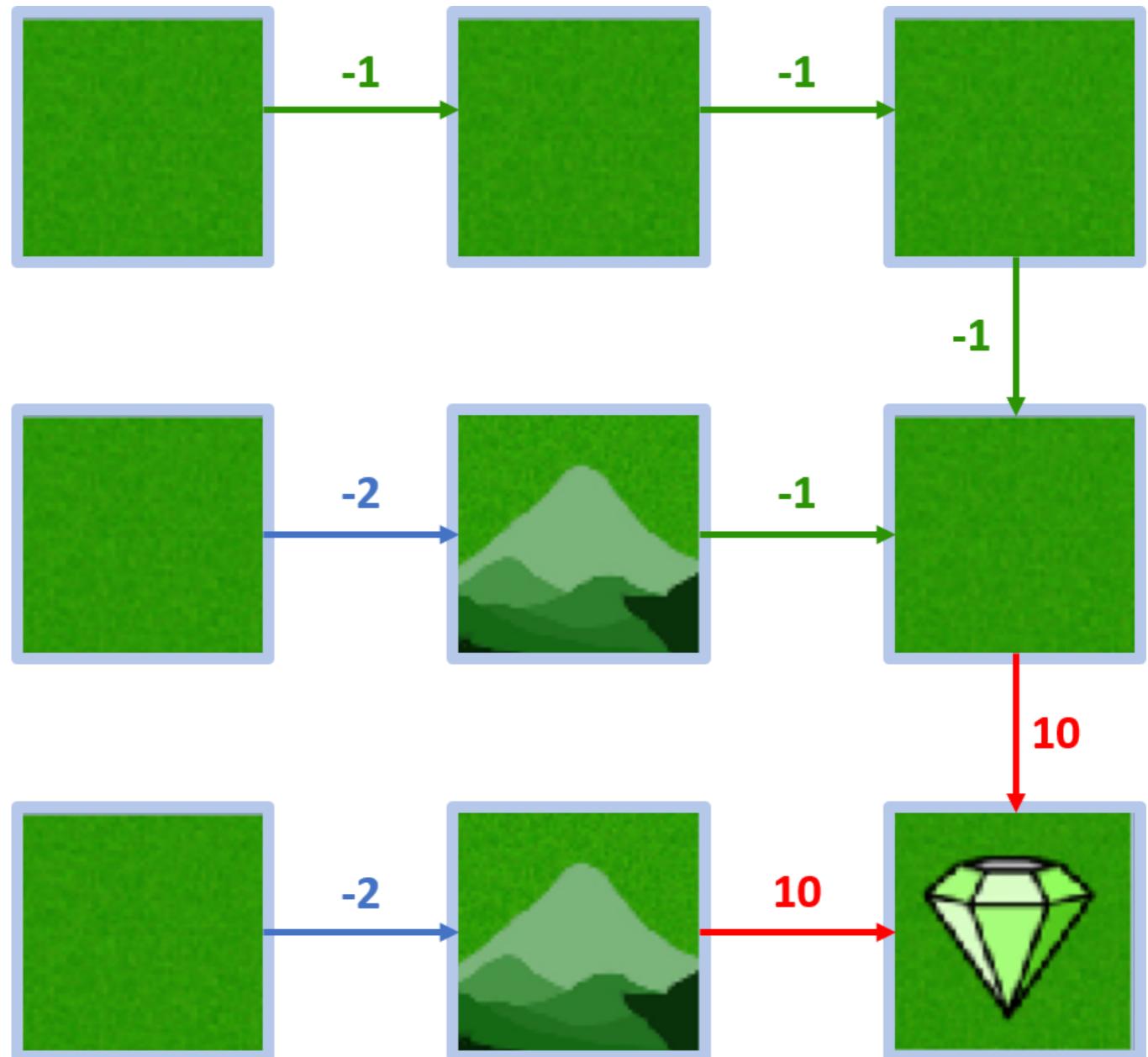
```
terminal_state = 8  
gamma = 1  
  
V = {state: compute_state_value(state)  
      for state in range(num_states)}  
  
print(V)
```

```
{0: 1, 1: 8, 2: 9,  
 3: 2, 4: 7, 5: 10,  
 6: 3, 7: 5, 8: 0}
```



Changing policies

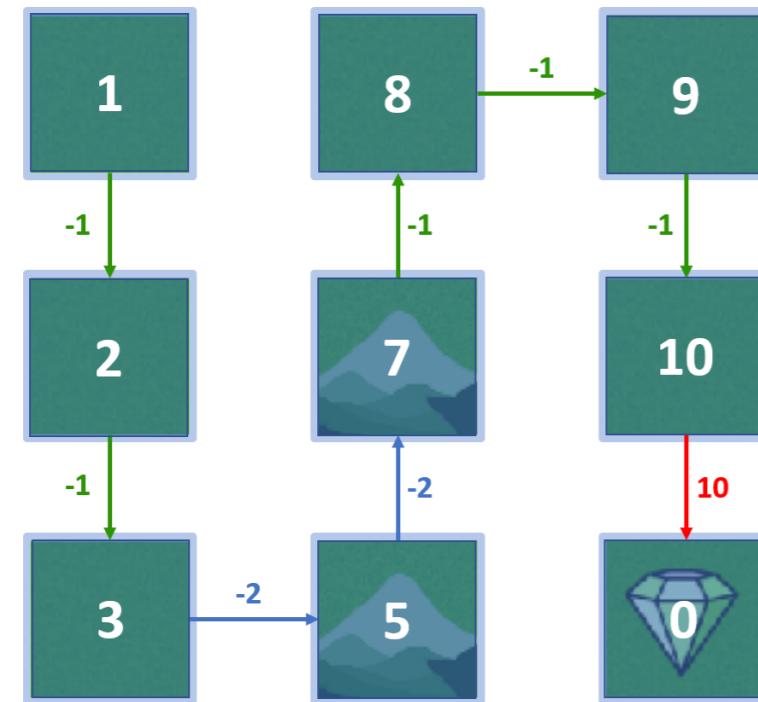
```
# 0: left, 1: down, 2: right, 3: up
policy_two = {
    0:2, 1:2, 2:1,
    3:2, 4:2, 5:1,
    6:2, 7:2
}
V_2 = {state: compute_state_value(state)
        for state in range(num_states)}
print(V_2)
```



Comparing policies

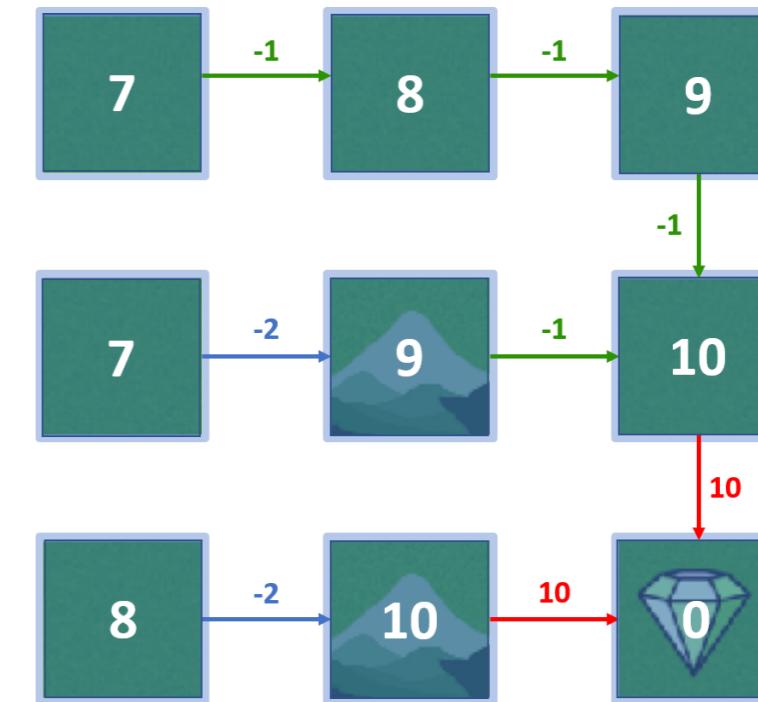
State-values for policy 1

```
{0: 1, 1: 8, 2: 9,  
3: 2, 4: 7, 5: 10,  
6: 3, 7: 5, 8: 0}
```



State-values for policy 2

```
{0: 7, 1: 8, 2: 9,  
3: 7, 4: 9, 5: 10,  
6: 8, 7: 10, 8: 0}
```



Let's practice!

REINFORCEMENT LEARNING WITH GYMNASIUM IN PYTHON

Action-value functions

REINFORCEMENT LEARNING WITH GYMNASIUM IN PYTHON



Fouad Trad
Machine Learning Engineer

Action-value functions (Q-values)

- Expected return of:
 - Starting at a state s
 - Taking action a
 - Following the policy
- Estimates desirability of actions within states

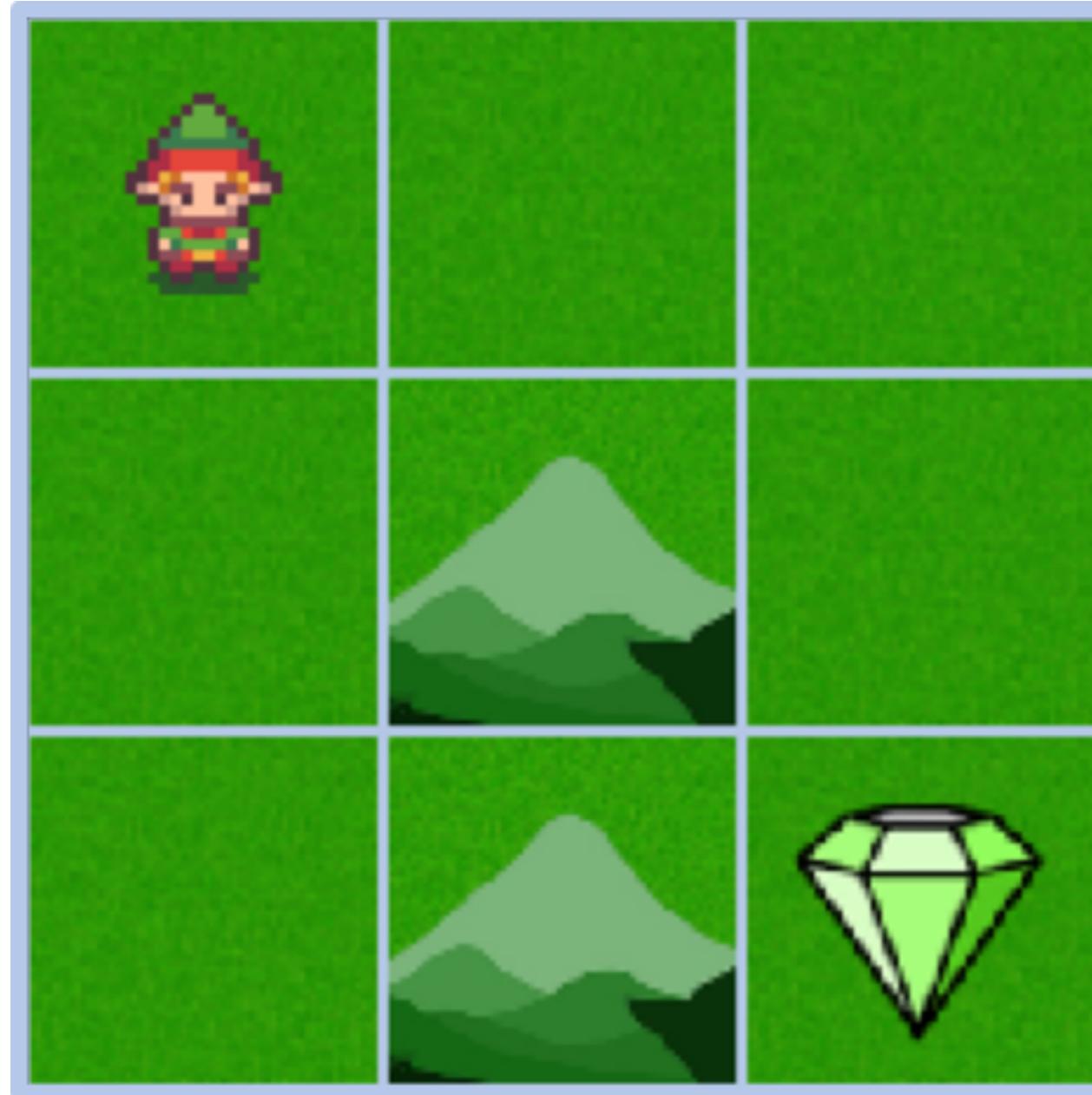
$$\underline{Q(s, a)} = \underline{r_a + \gamma V(s + 1)}$$

*Action – value
of state s , action a*

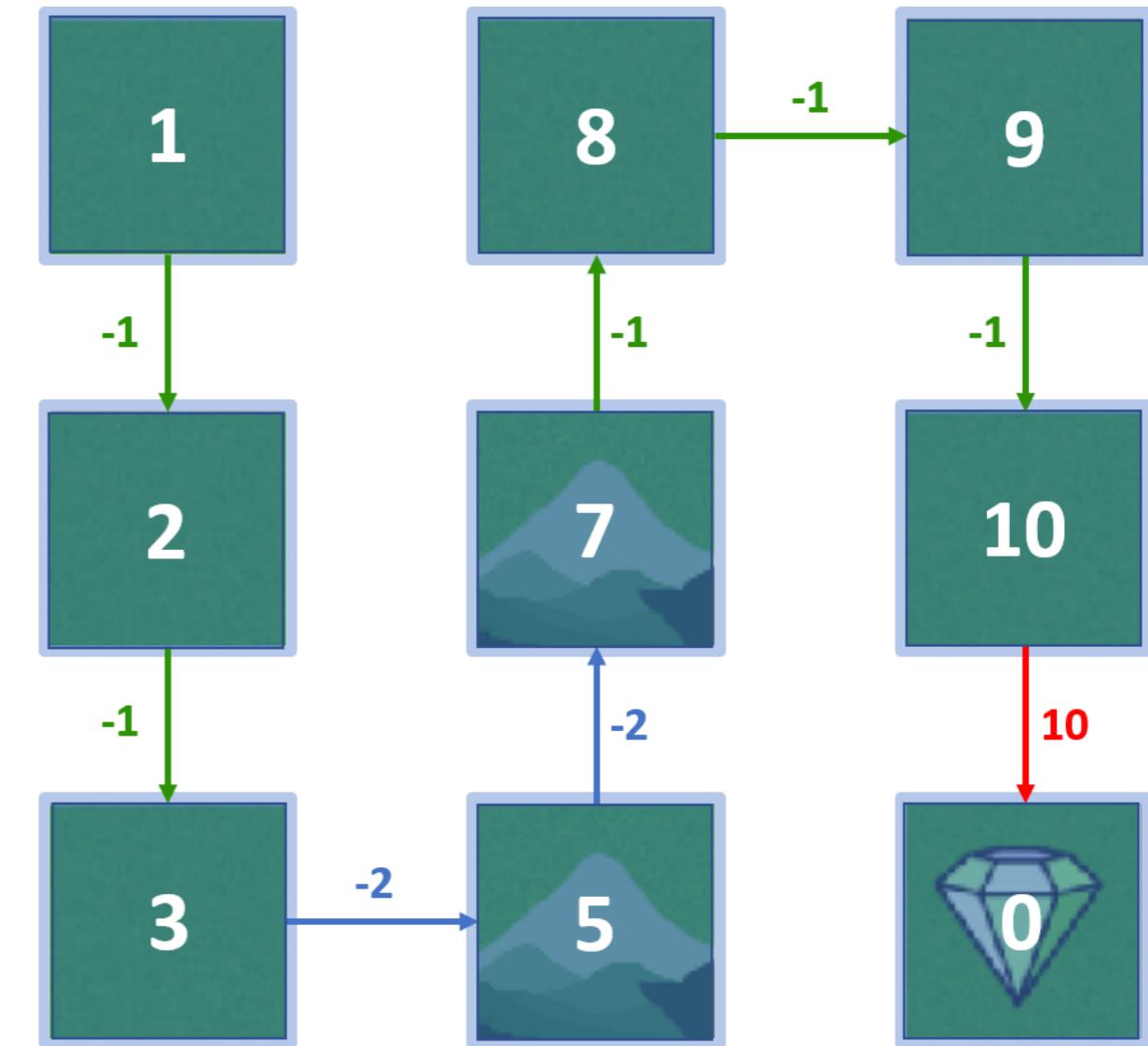
Sum of:

- reward received after performing action a in state s
- discounted value of the next state resulting from action a

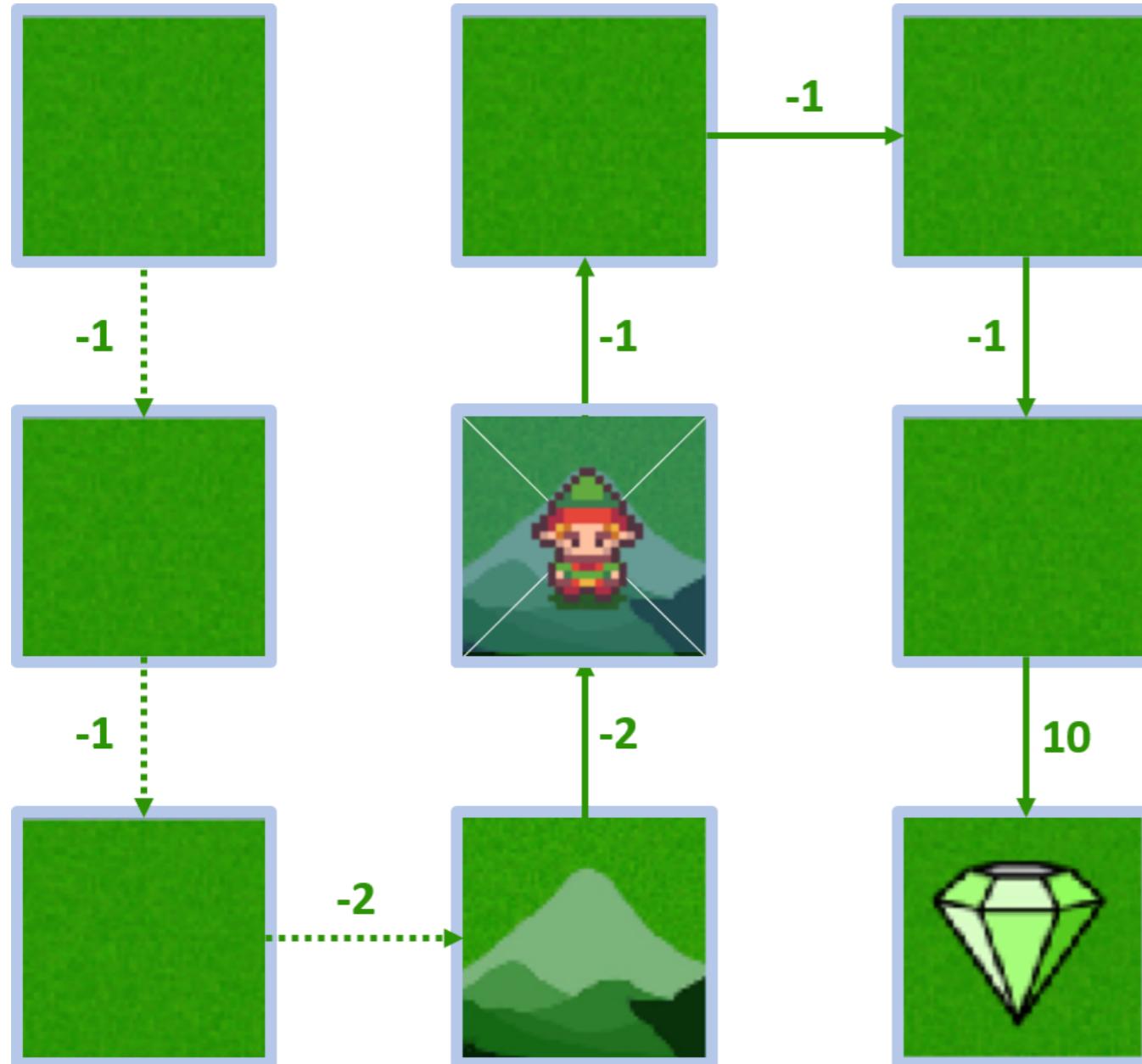
Grid world



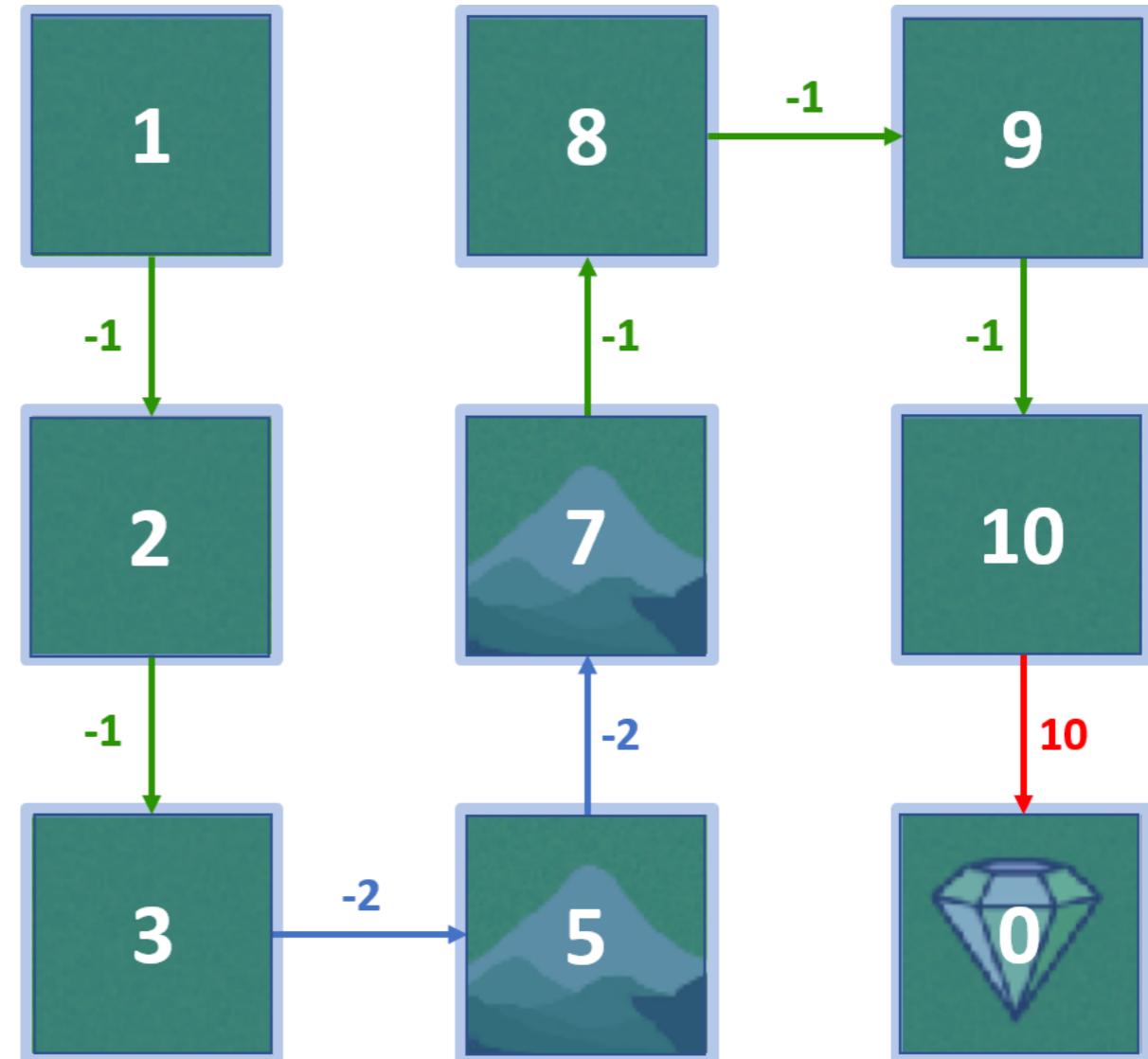
- State-values



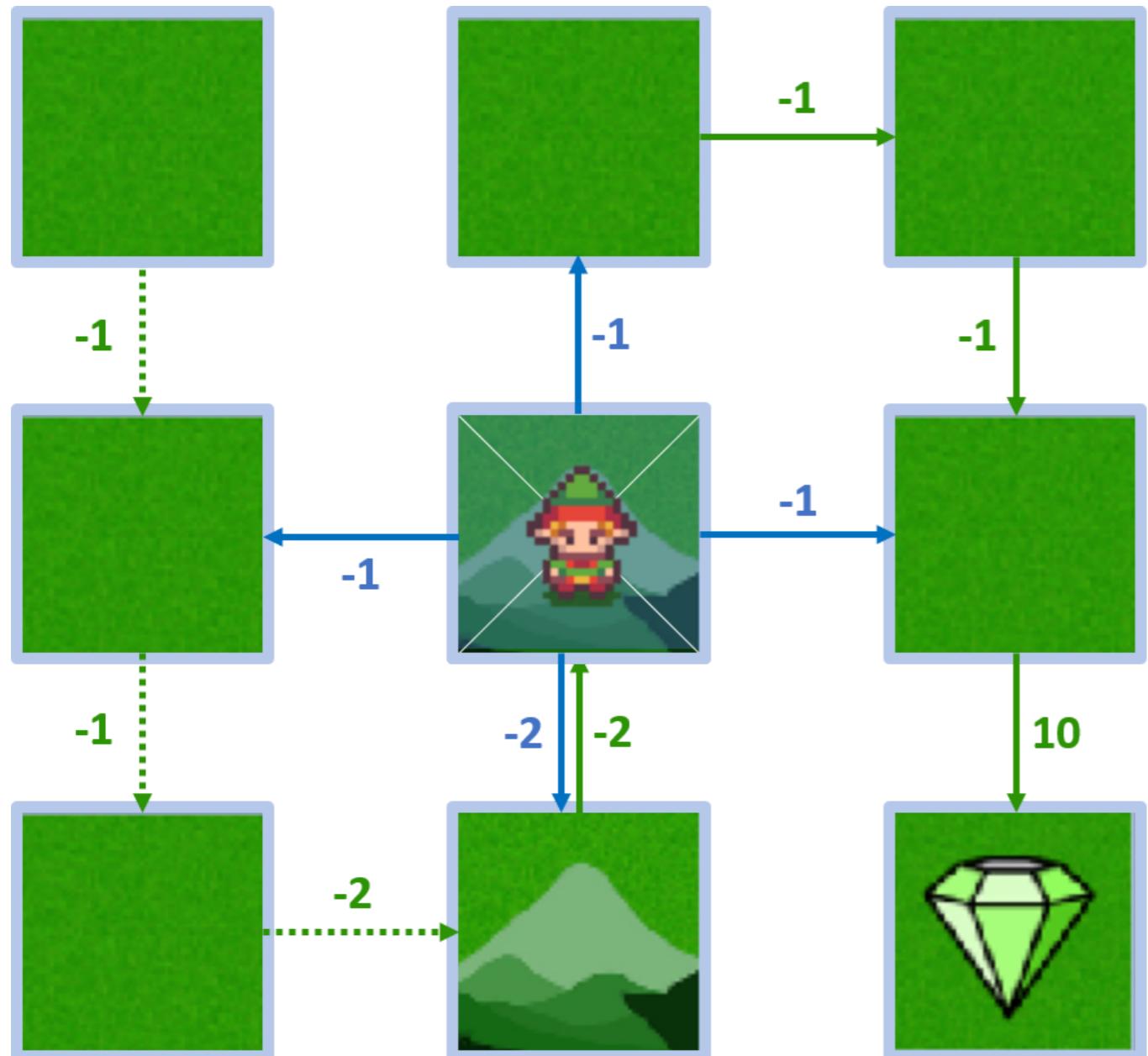
Q-values - state 4



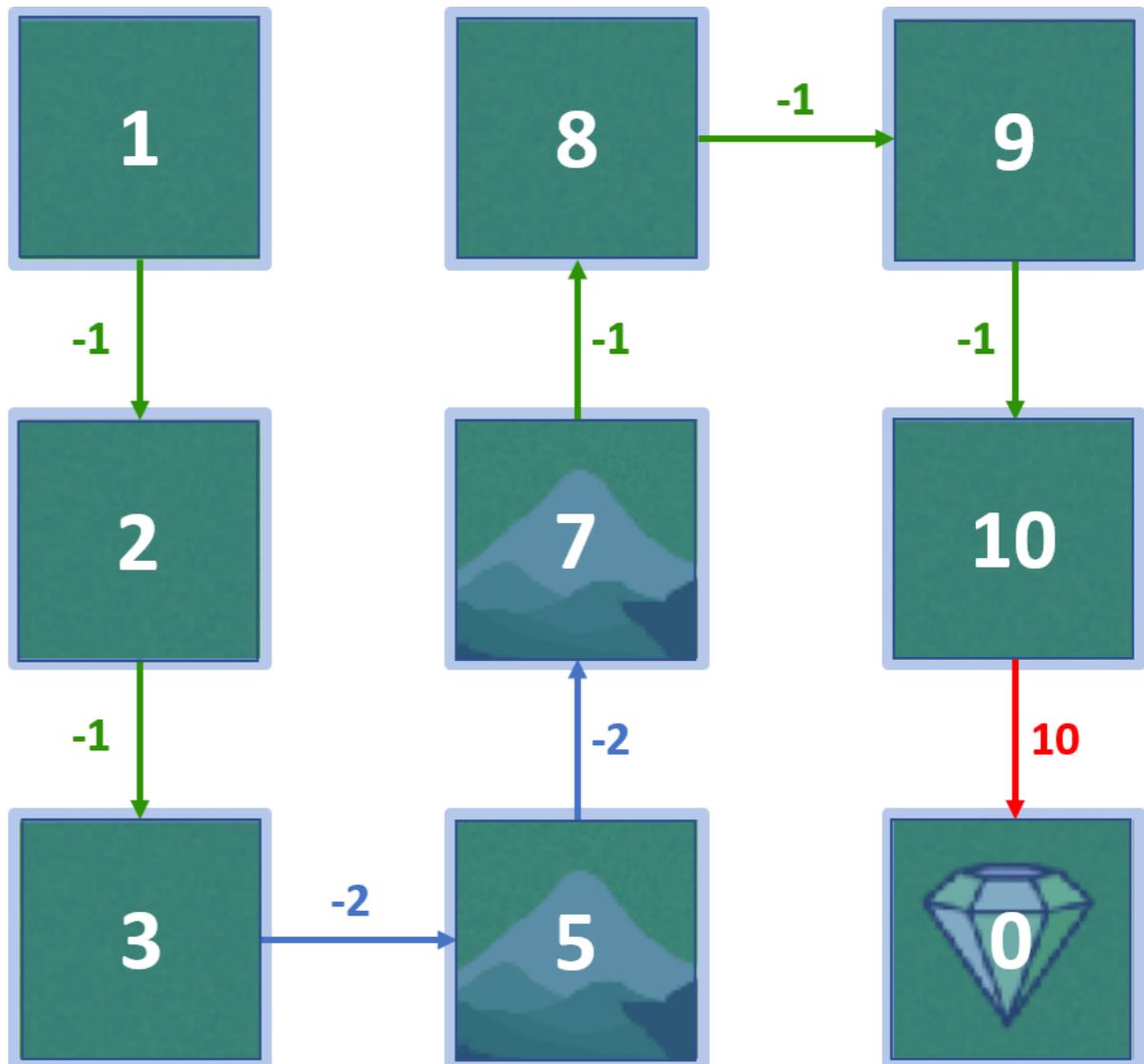
- Agent born in state 4



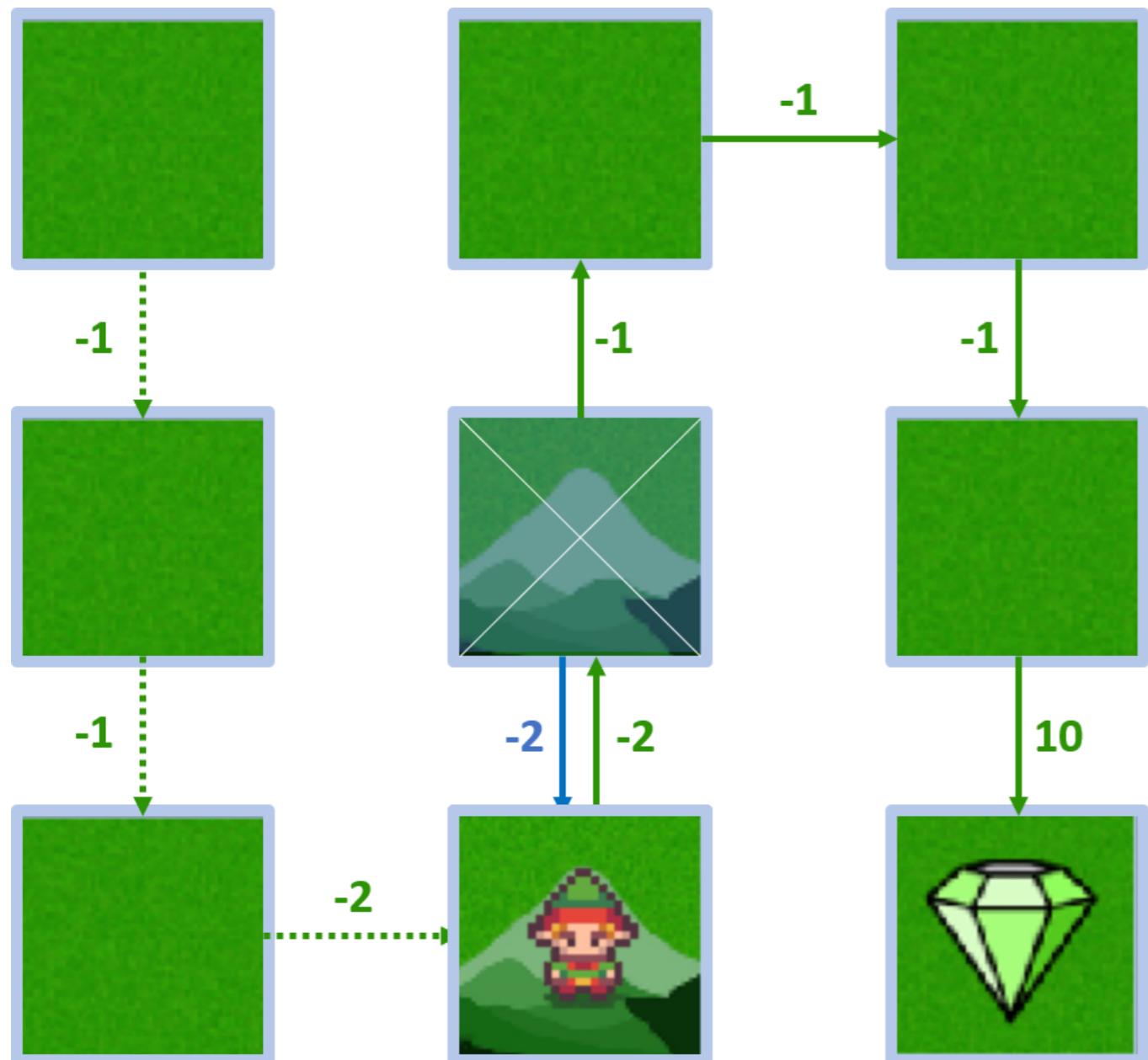
Q-values - state 4



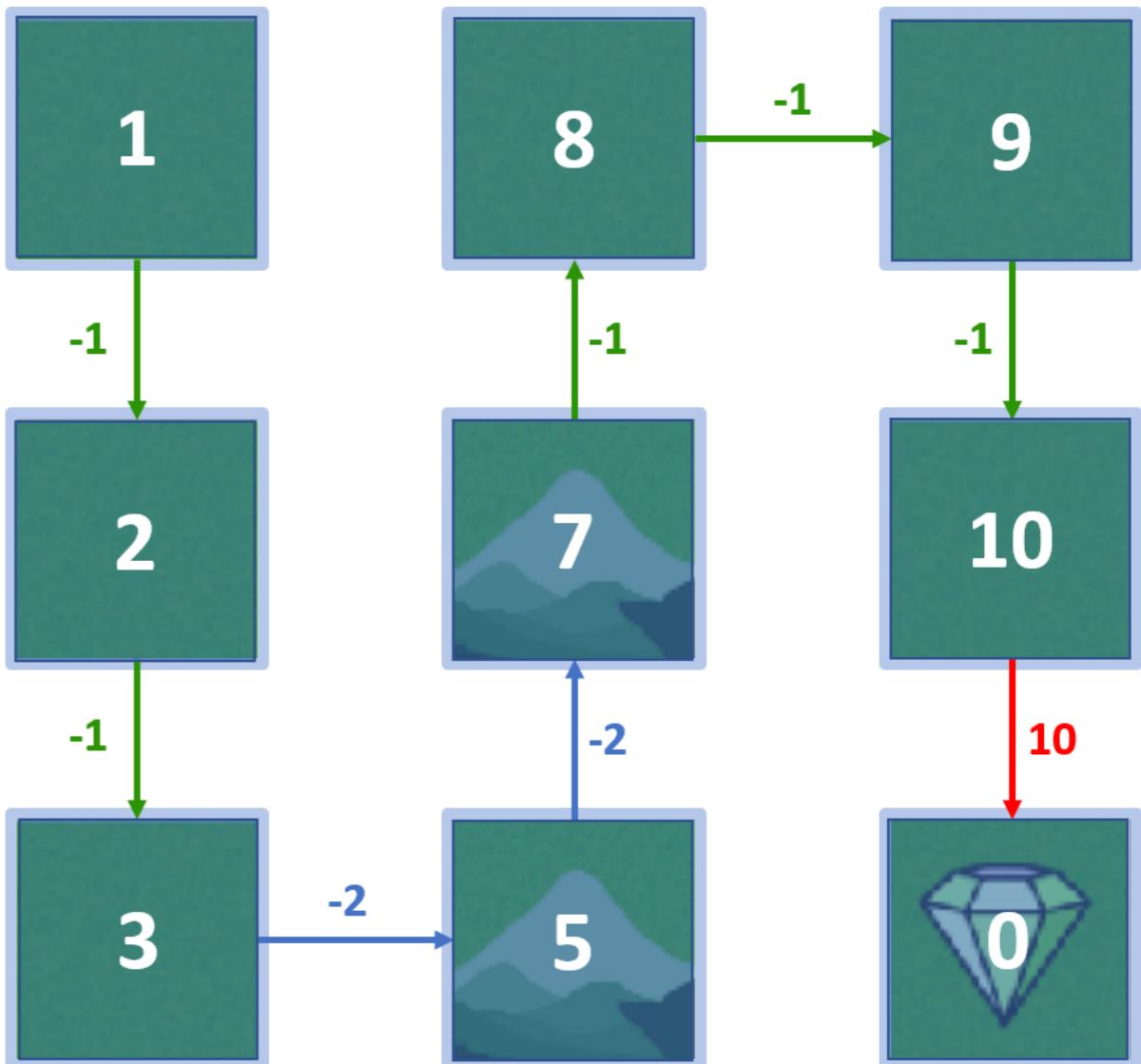
- Agent can move up, down, left, right



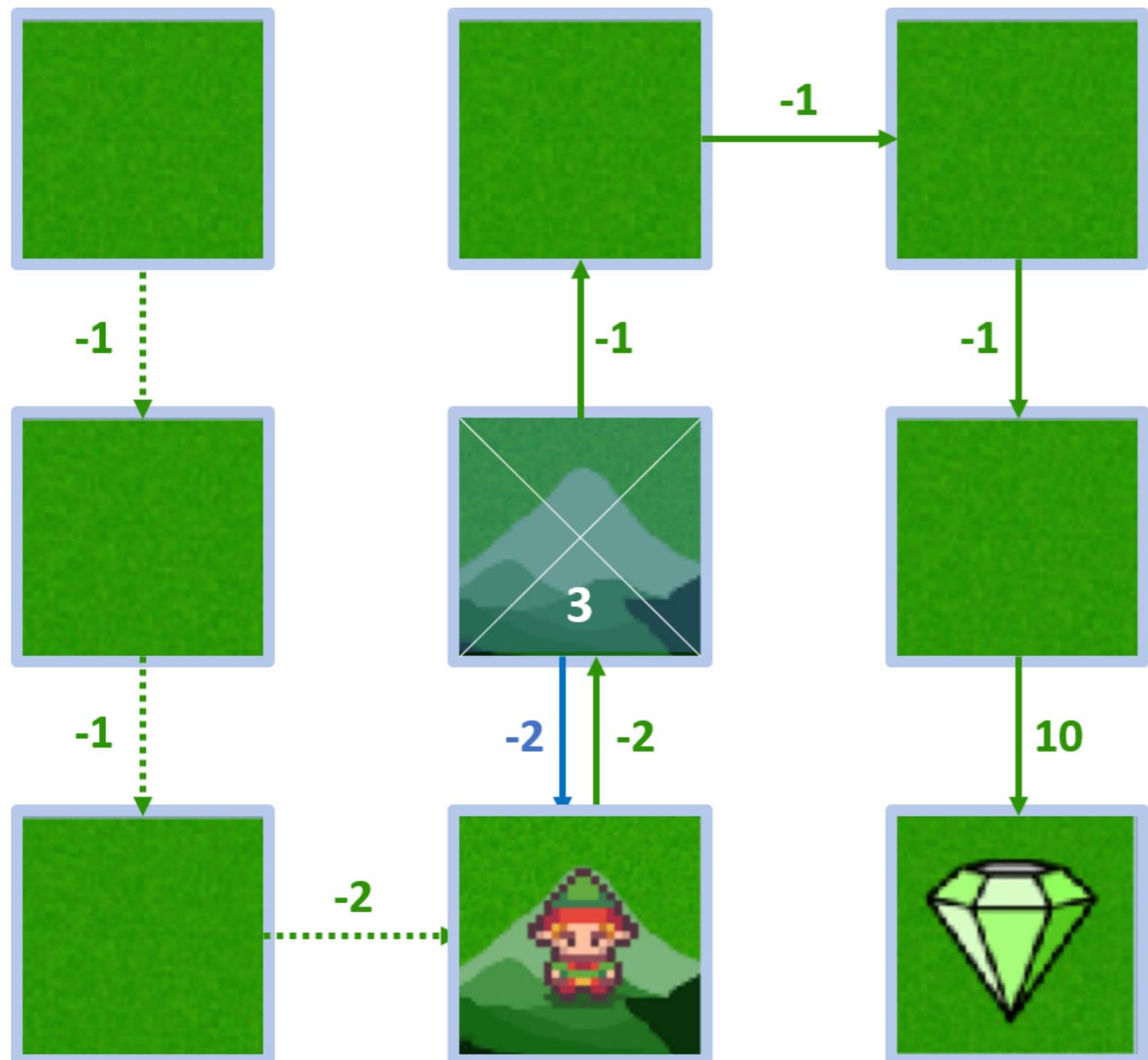
State 4 - action down



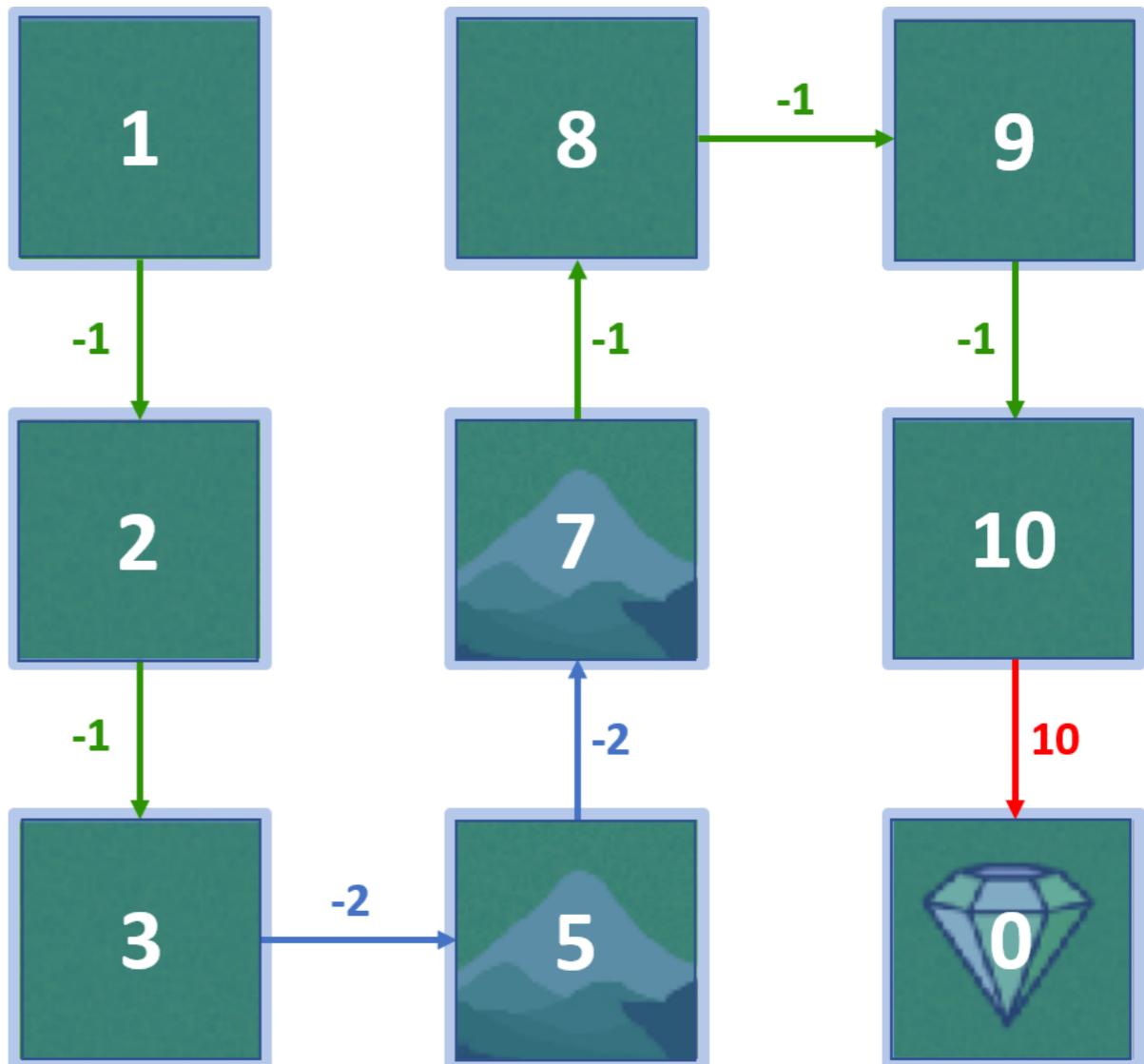
- Reward: -2, state-value: 5



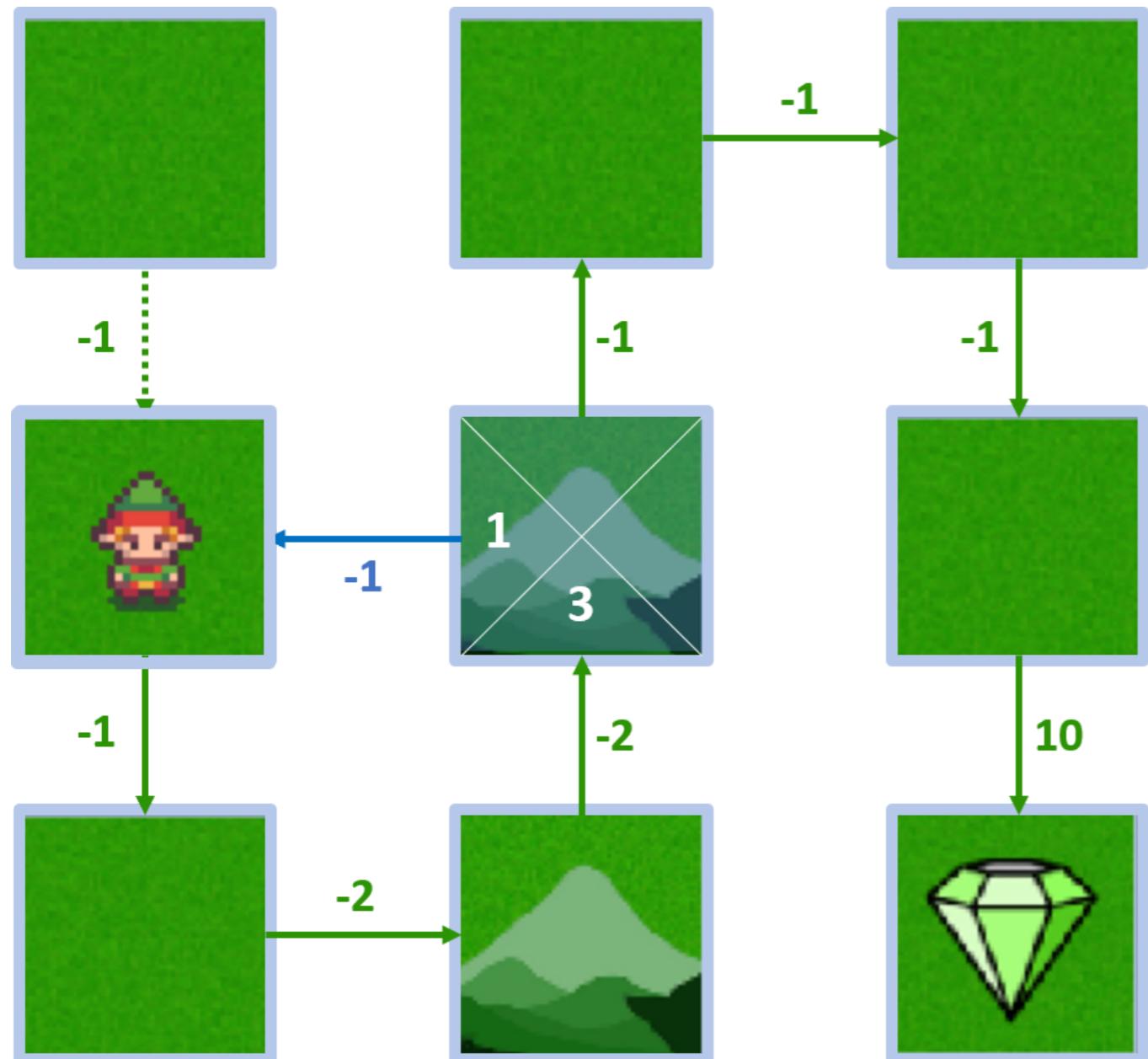
State 4 - action down



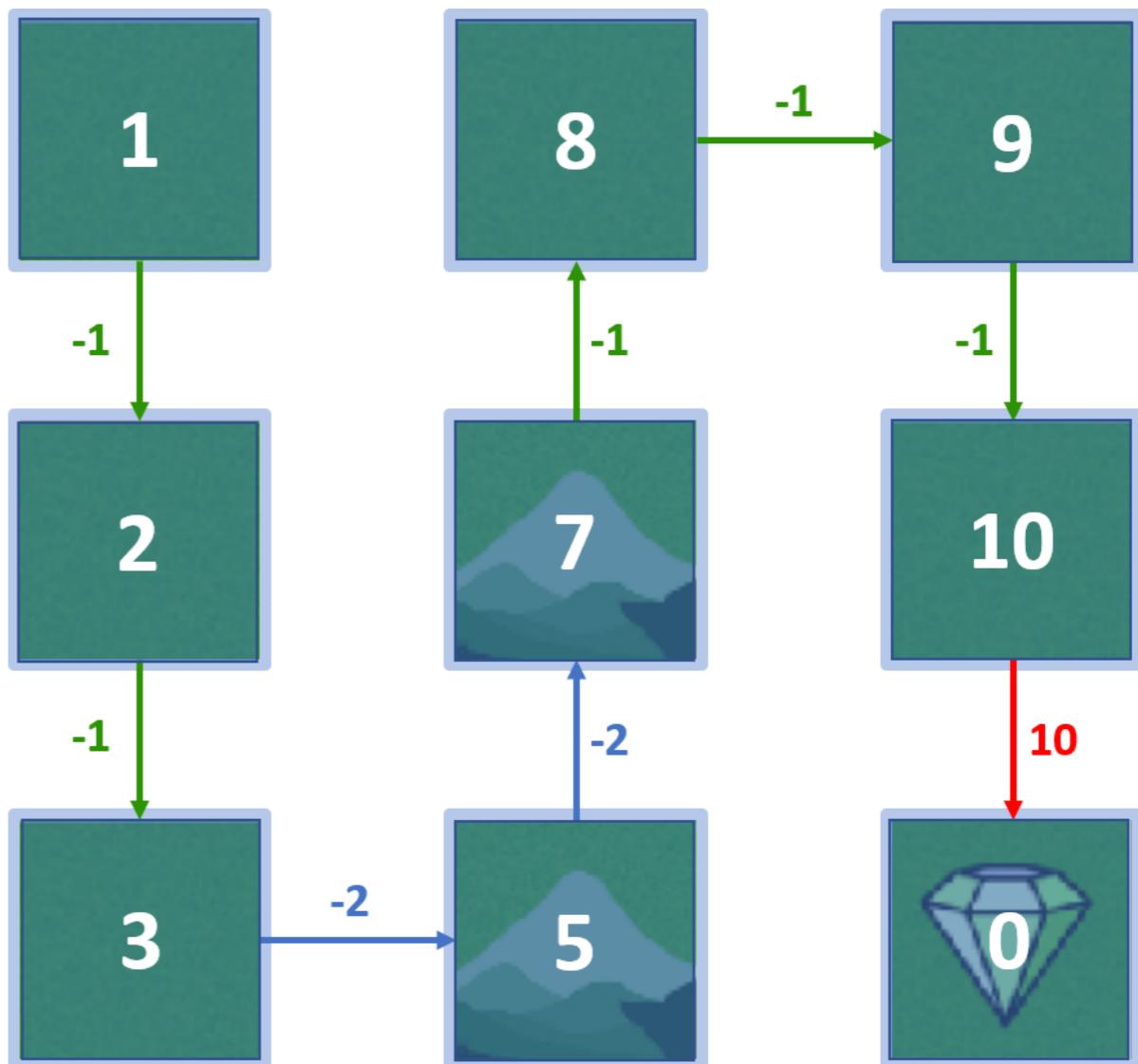
- $Q(4, \text{down}) = -2 + 1 \times 5 = 3$



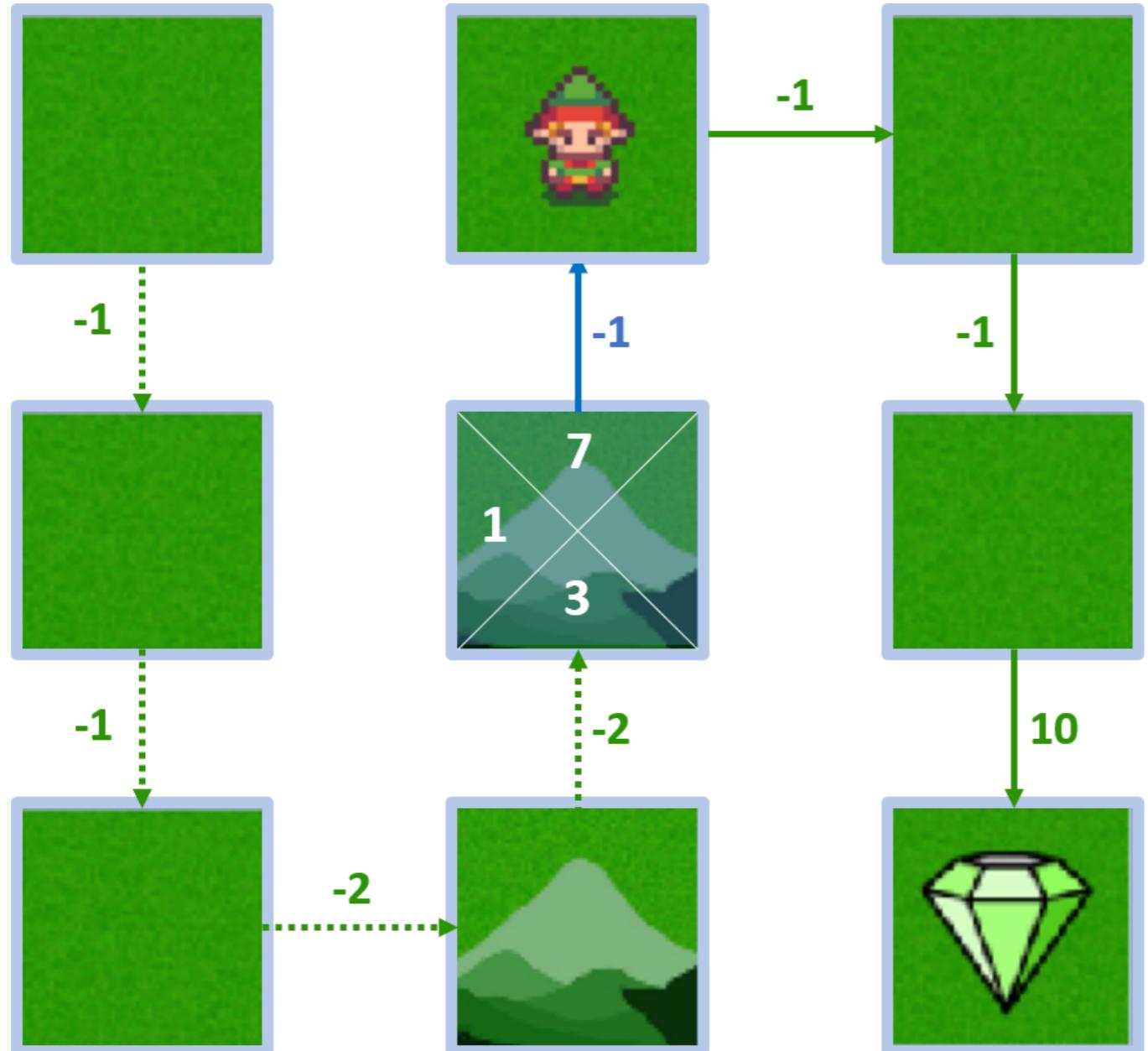
State 4 - action left



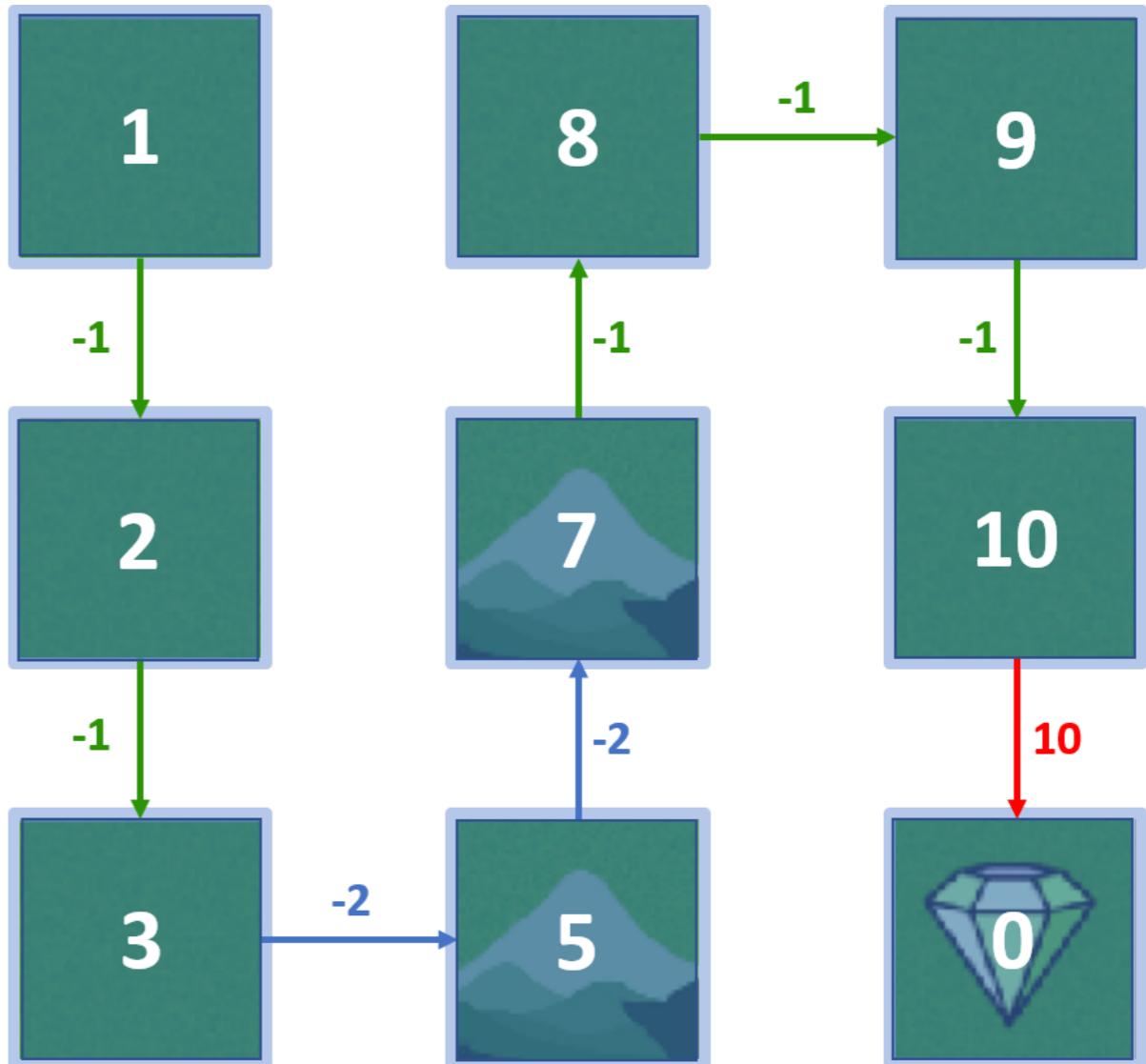
- $Q(4, \text{left}) = -1 + 1 \times 2 = 1$



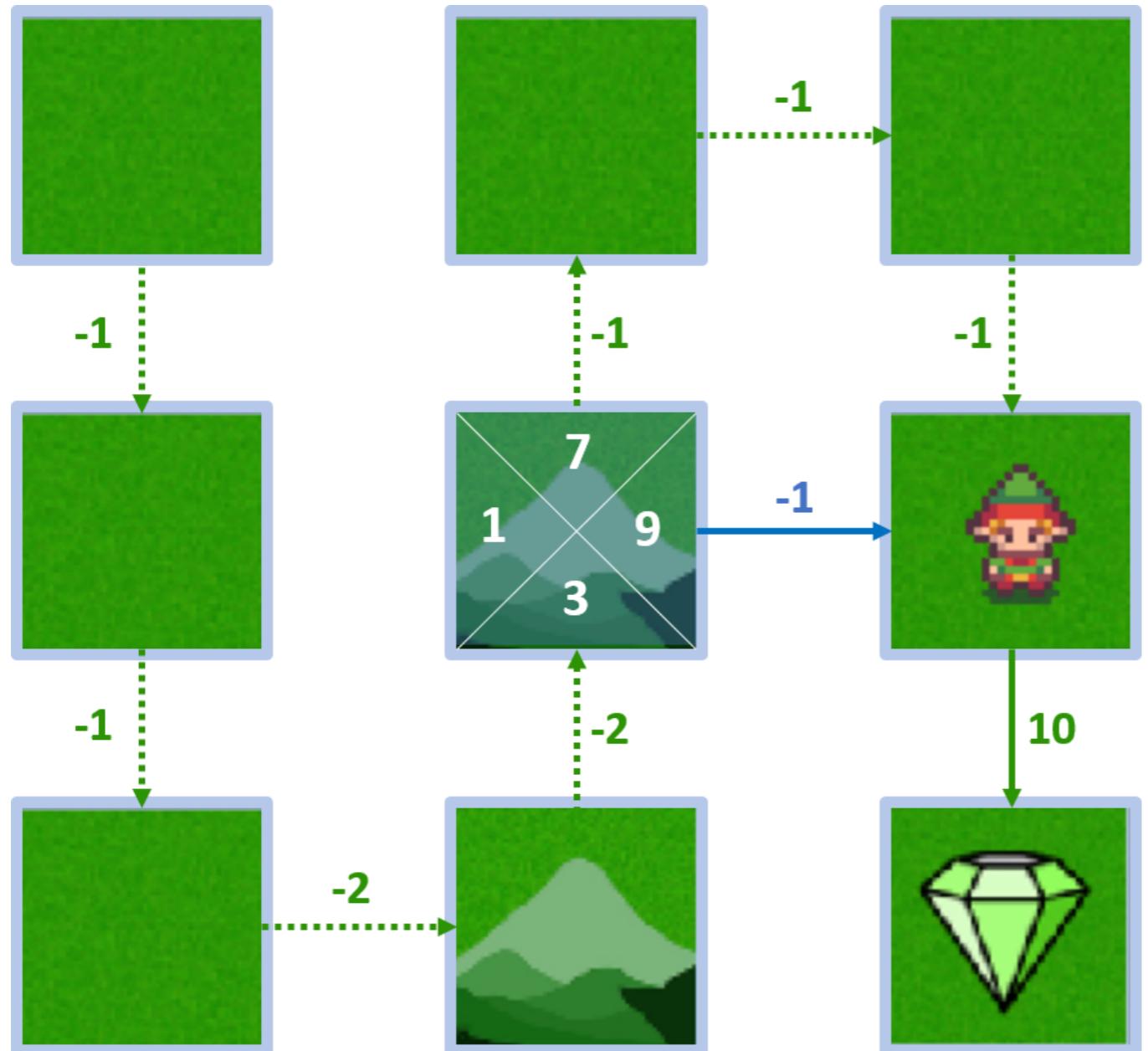
State 4 - action up



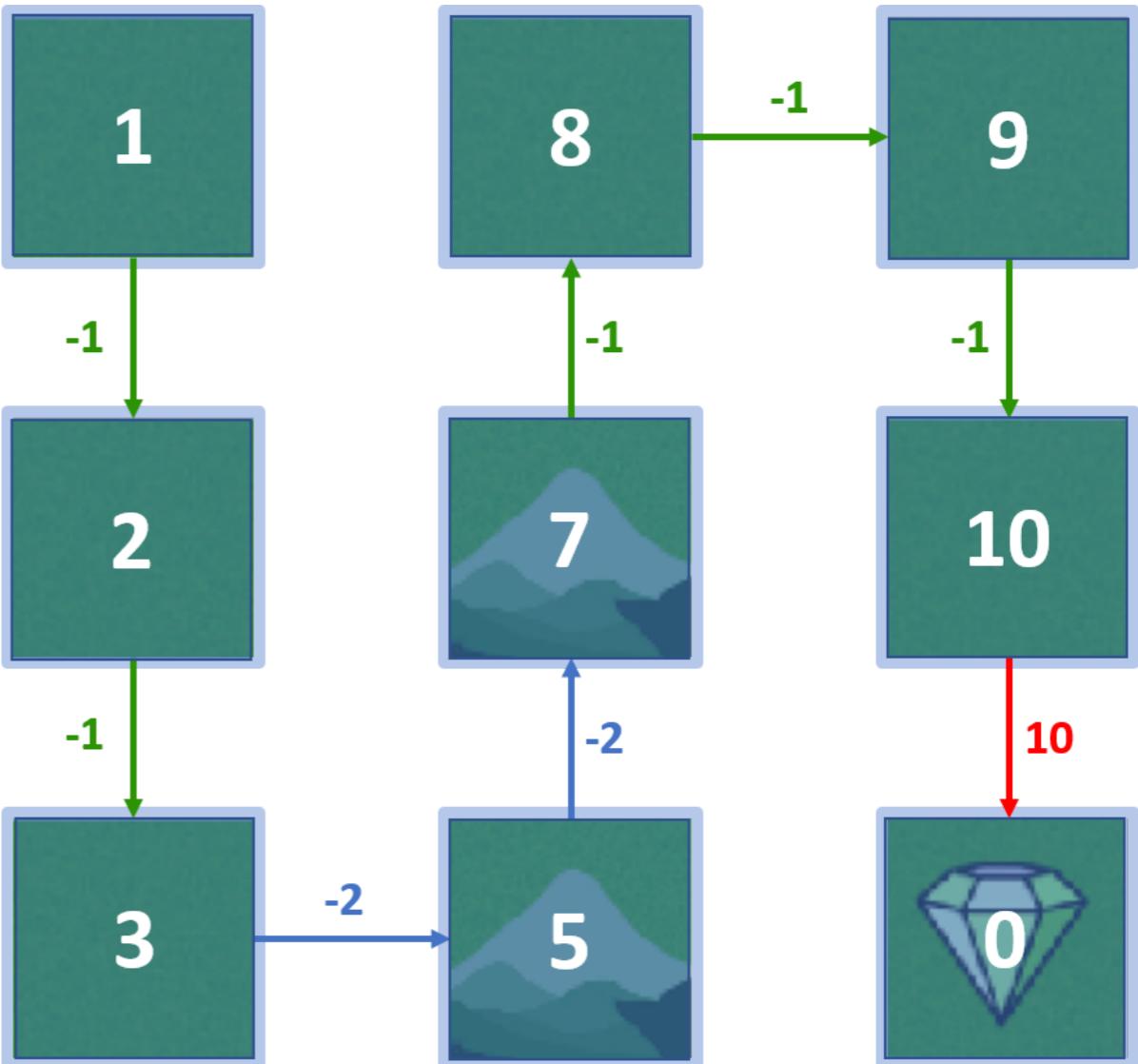
- $Q(4, \text{up}) = -1 + 1 \times 8 = 7$



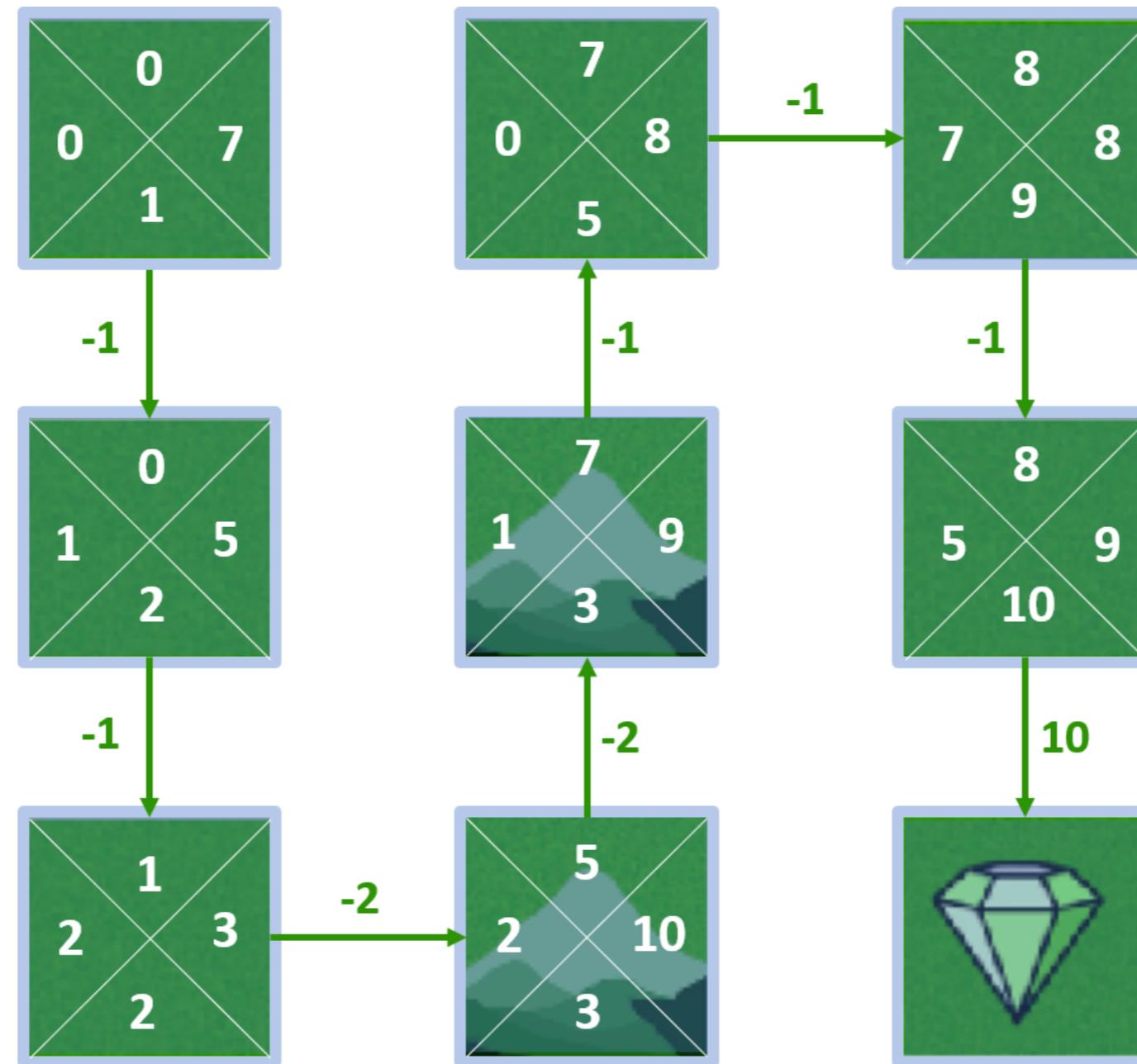
State 4 - action right



- $Q(4, \text{right}) = -1 + 1 \times 10 = 9$



All Q-values



Computing Q-values

```
def compute_q_value(state, action):  
    if state == terminal_state:  
        return None  
    _, next_state, reward, _ = env.unwrapped.P[state][action][0]  
    return reward + gamma * compute_state_value(next_state)
```

$$\underline{Q(s, a) = r_a + \gamma V(s + 1)}$$

*Action – value
of state s, action a*

Sum of:

- reward received after performing action a in state s
- discounted value of the next state resulting from action a

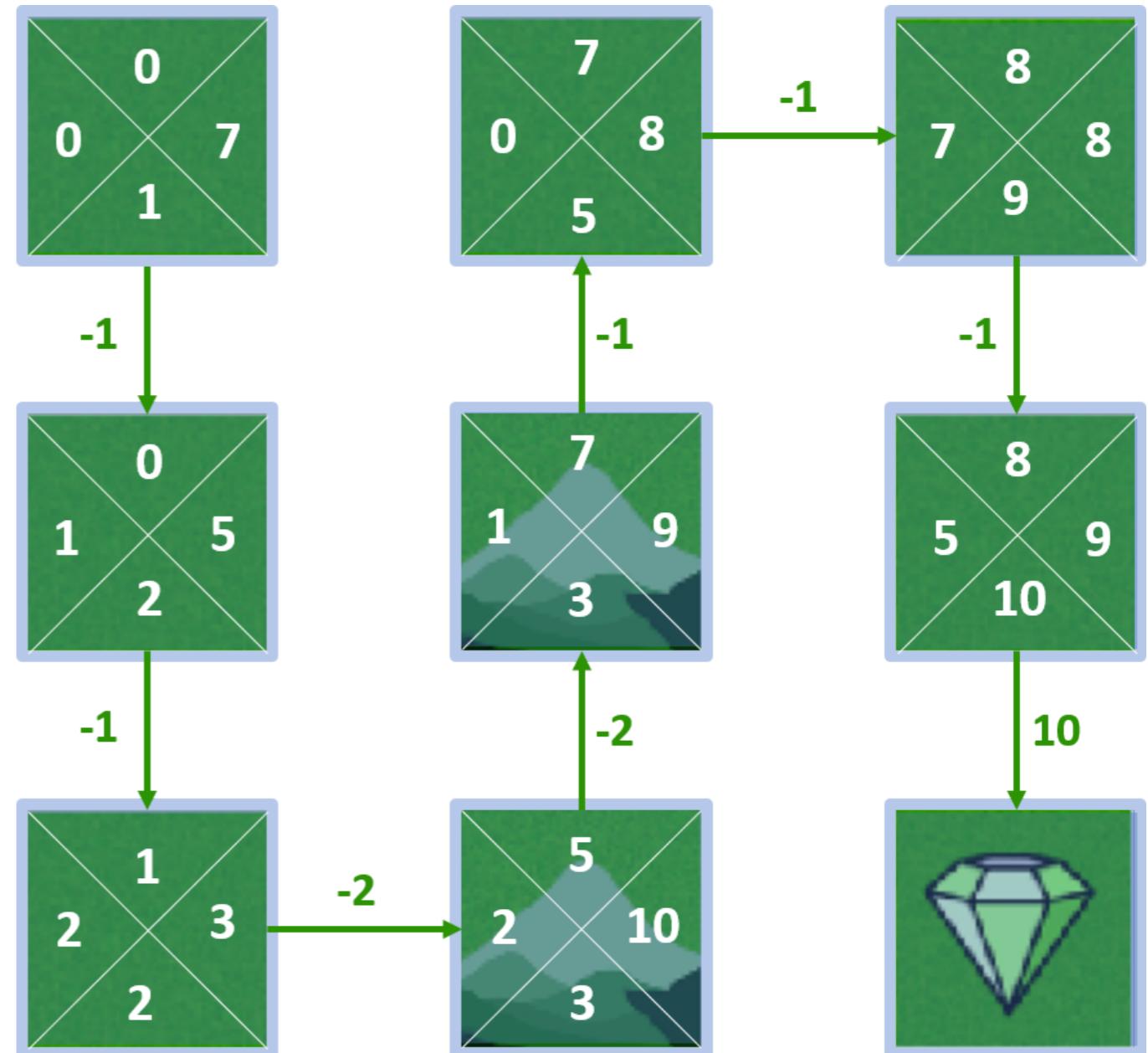
Computing Q-values

```
Q = {(state, action): compute_q_value(state, action)
      for state in range(num_states)
      for action in range(num_actions)}

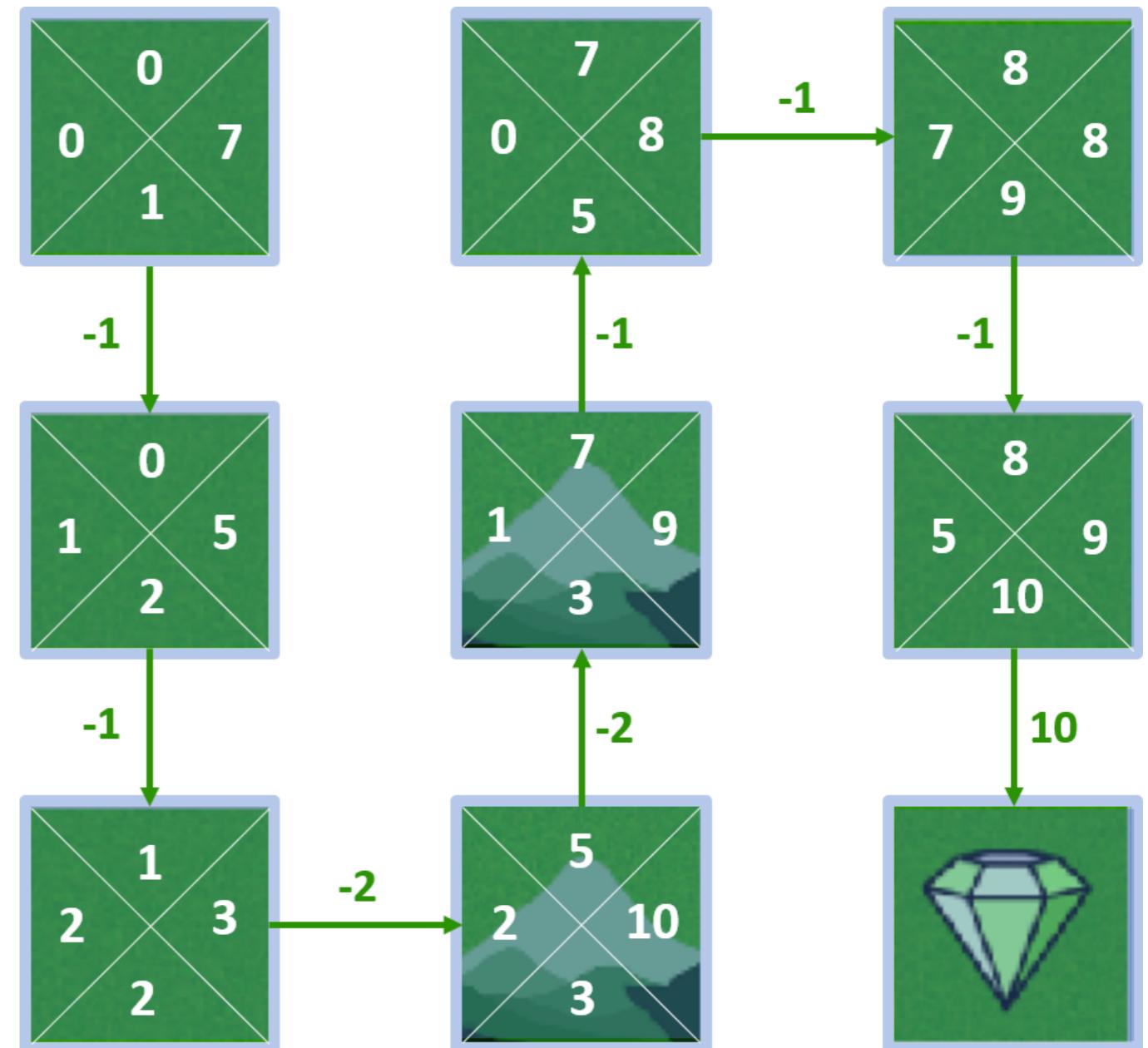
print(Q)
```

Computing Q-values

```
{(0, 0): 0, (0, 1): 1, (0, 2): 7, (0, 3): 0,  
 (1, 0): 0, (1, 1): 5, (1, 2): 8, (1, 3): 7,  
 (2, 0): 7, (2, 1): 9, (2, 2): 8, (2, 3): 8,  
 (3, 0): 1, (3, 1): 2, (3, 2): 5, (3, 3): 0,  
 (4, 0): 1, (4, 1): 3, (4, 2): 9, (4, 3): 7,  
 (5, 0): 5, (5, 1): 10, (5, 2): 9, (5, 3): 8,  
 (6, 0): 2, (6, 1): 2, (6, 2): 3, (6, 3): 1,  
 (7, 0): 2, (7, 1): 3, (7, 2): 10, (7, 3): 5,  
 (8, 0): None, (8, 1): None, (8, 2): None, (8, 3): None}
```

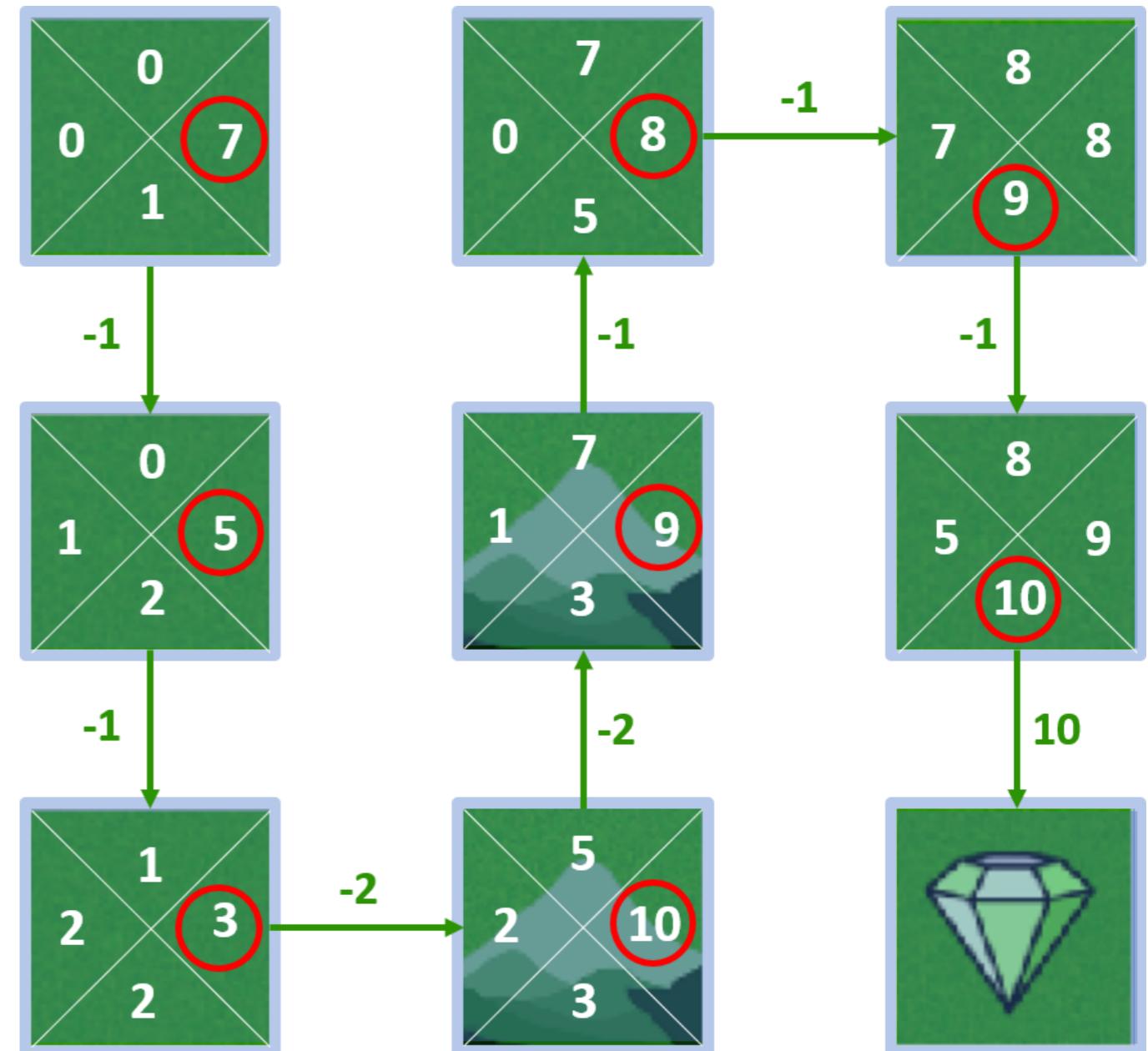


Improving the policy

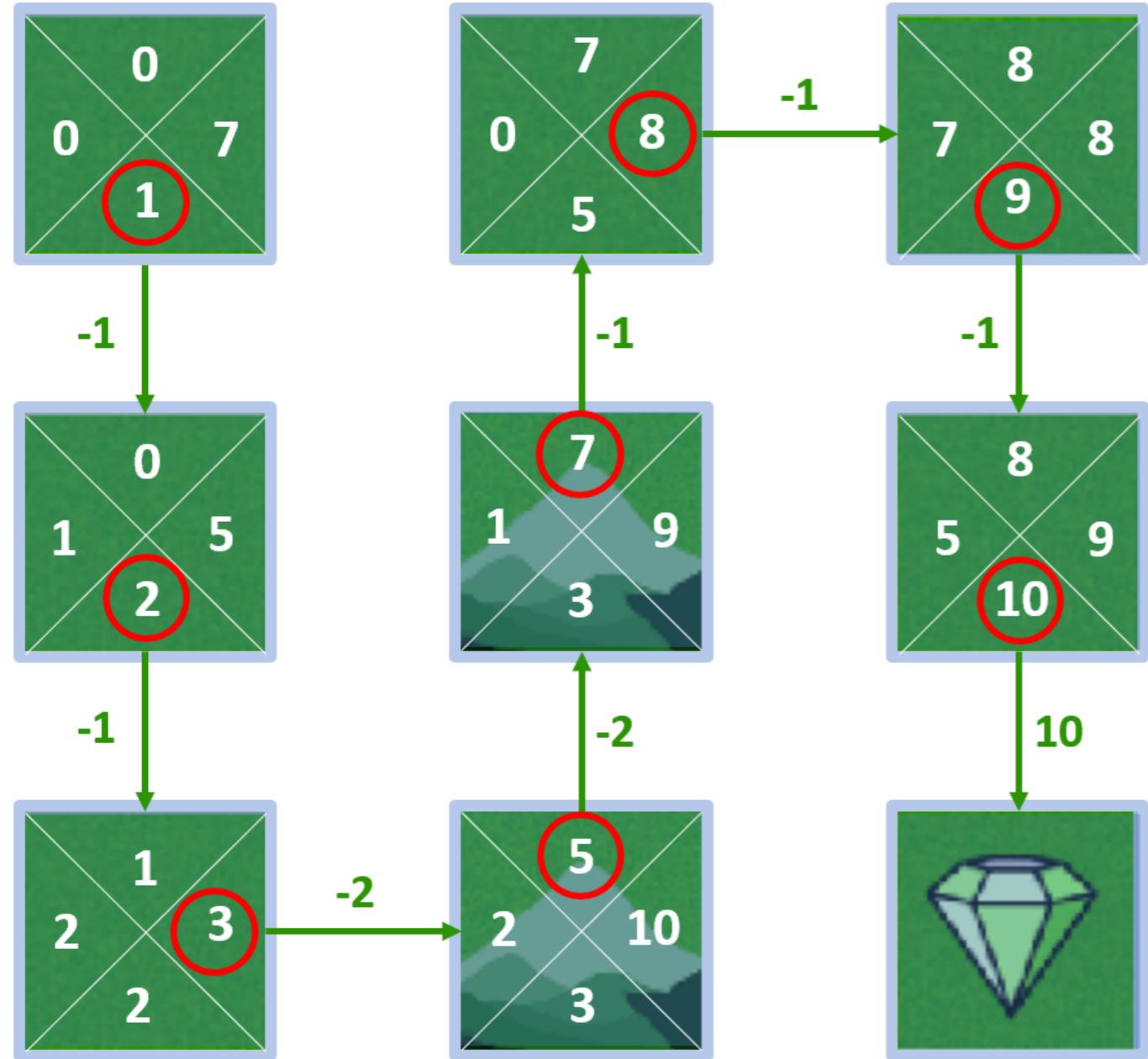


Improving the policy

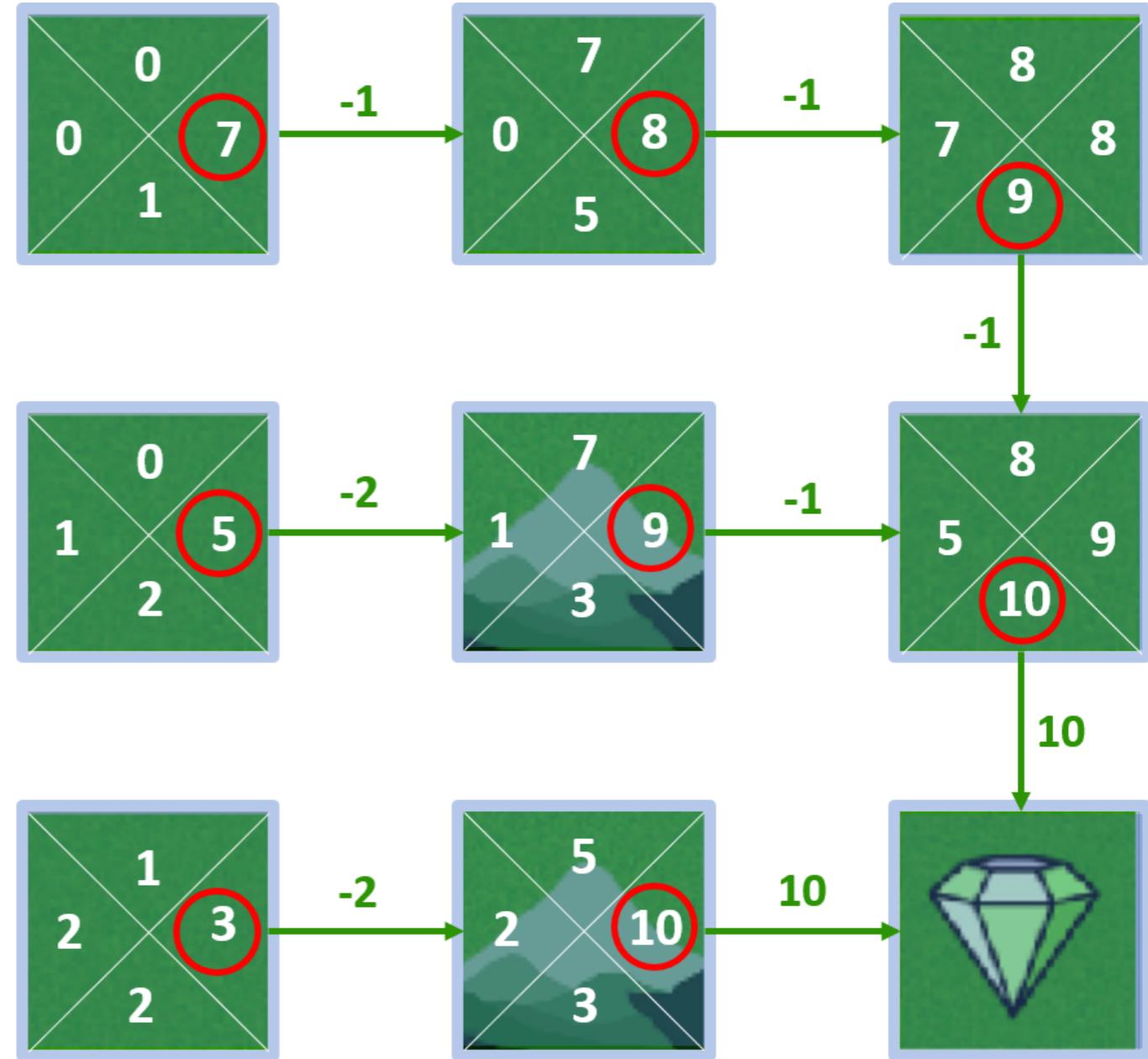
- Selecting for each state the action with highest Q-value



Improving the policy



Old policy



Improving the policy

```
improved_policy = {}

for state in range(num_states-1):
    max_action = max(range(num_actions), key=lambda action: Q[(state, action)])
    improved_policy[state] = max_action

print(improved_policy)
```

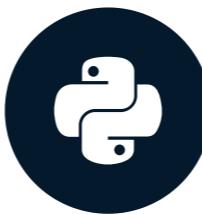
```
{0: 2, 1: 2, 2: 1,
3: 2, 4: 2, 5: 1,
6: 2, 7: 2}
```

Let's practice!

REINFORCEMENT LEARNING WITH GYMNASIUM IN PYTHON

Policy iteration and value iteration

REINFORCEMENT LEARNING WITH GYMNASIUM IN PYTHON



Fouad Trad
Machine Learning Engineer

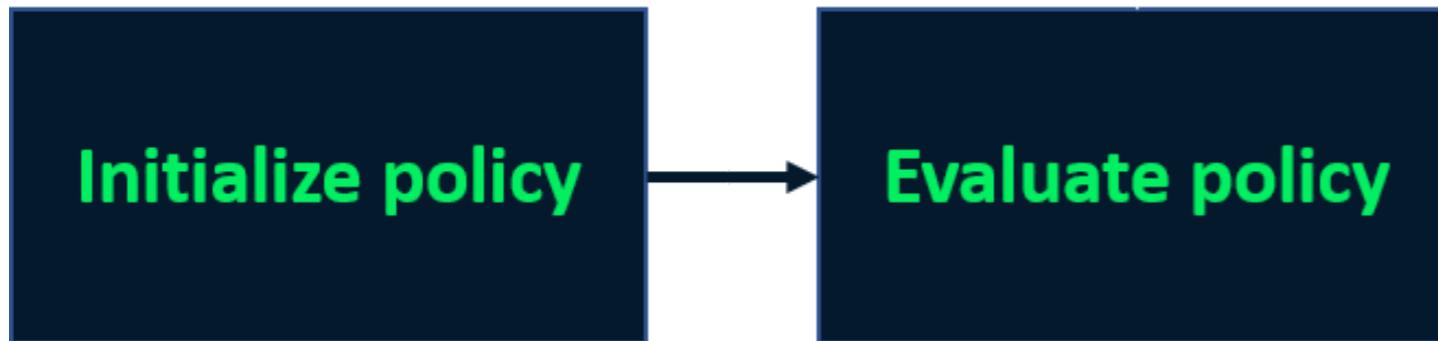
Policy iteration

- Iterative process to find optimal policy

Initialize policy

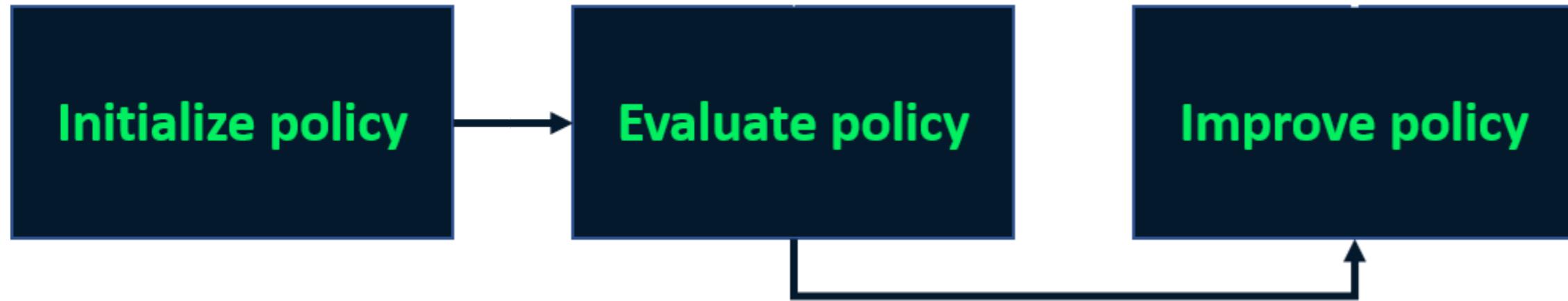
Policy iteration

- Iterative process to find optimal policy



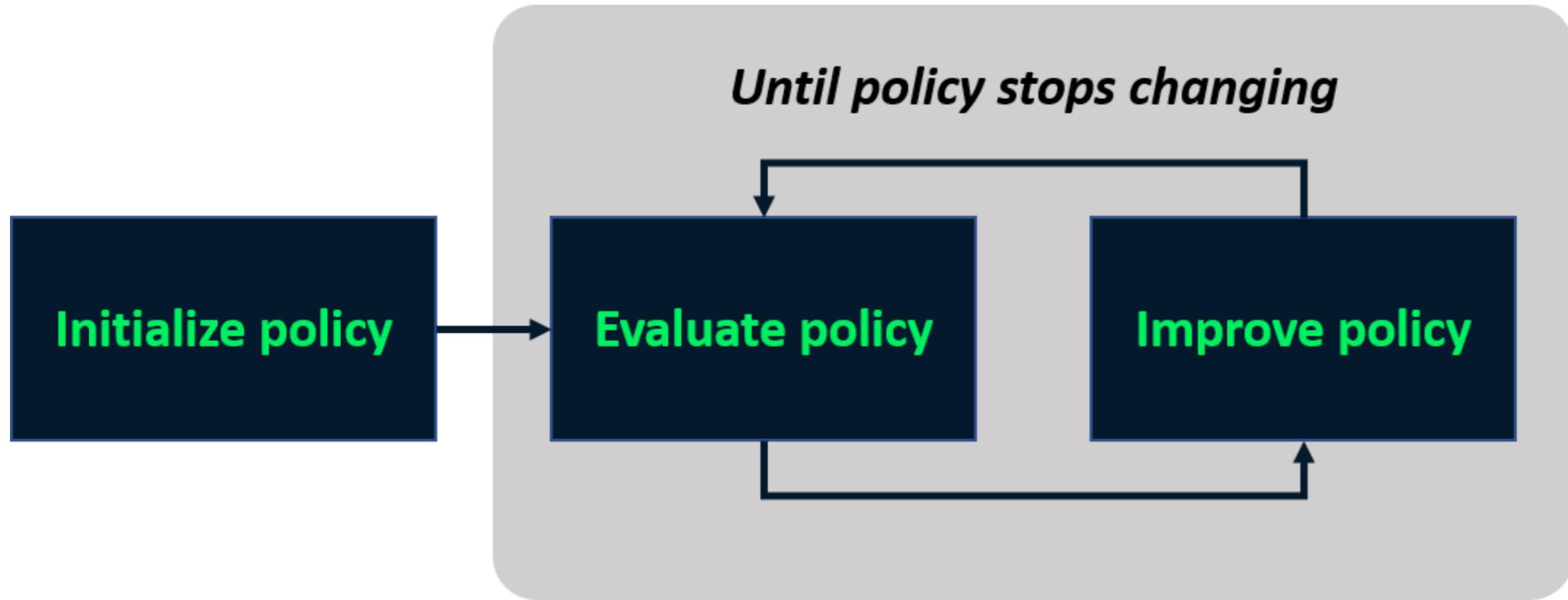
Policy iteration

- Iterative process to find optimal policy



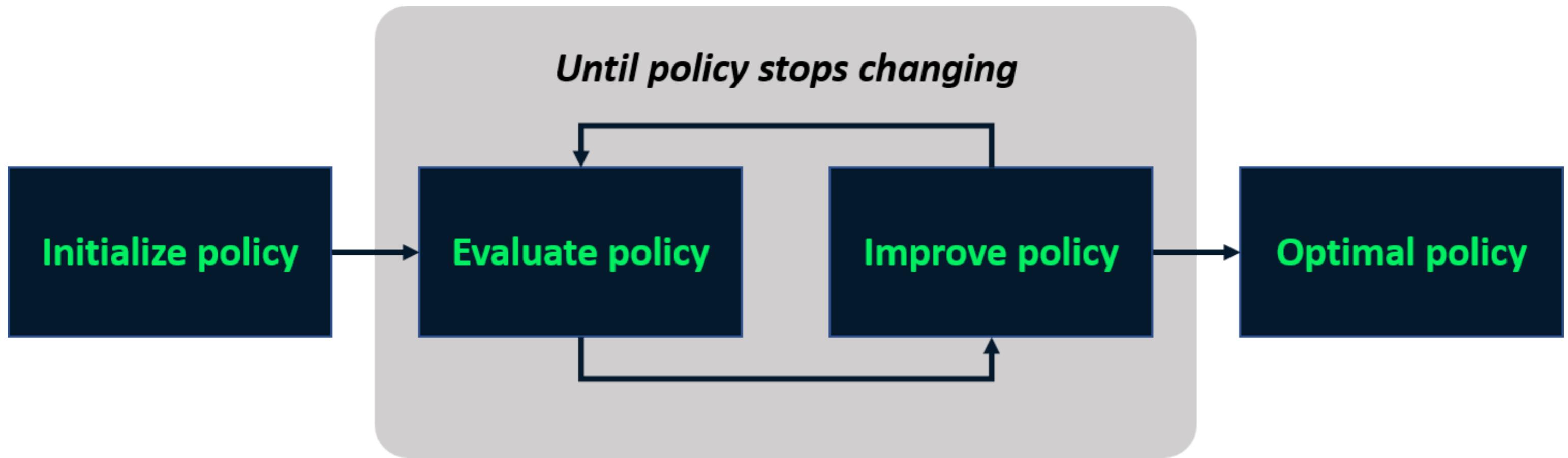
Policy iteration

- Iterative process to find optimal policy



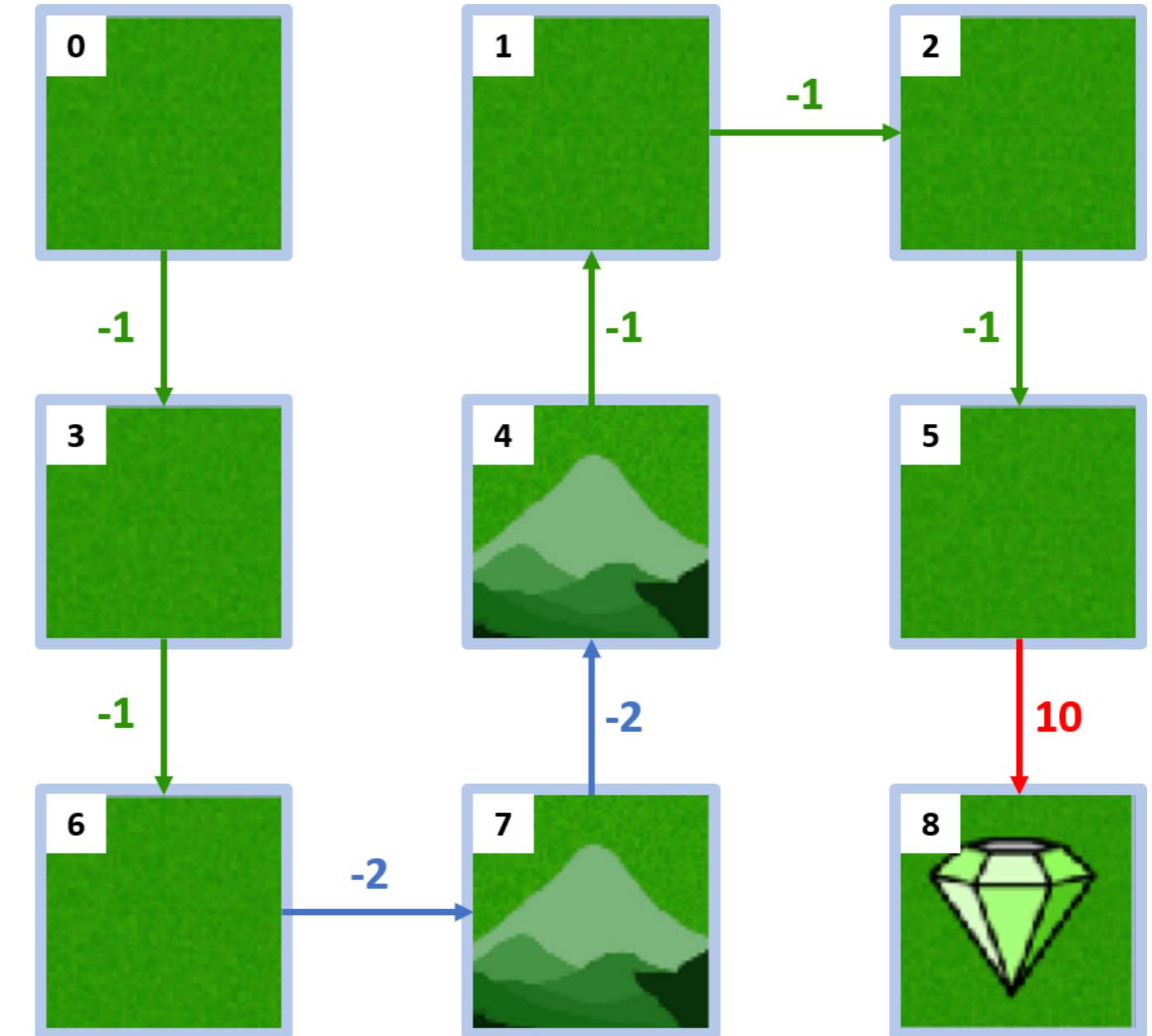
Policy iteration

- Iterative process to find optimal policy



Grid world

```
policy = {  
    0:1, 1:2, 2:1,  
    3:1, 4:3, 5:1,  
    6:2, 7:3  
}
```



Policy evaluation

```
def policy_evaluation(policy):  
    V = {state: compute_state_value(state, policy) for state in range(num_states)}  
    return V
```

Policy improvement

```
def policy_improvement(policy):
    improved_policy = {s: 0 for s in range(num_states-1)}
    Q = {(state, action): compute_q_value(state, action, policy)
          for state in range(num_states) for action in range(num_actions)}

    for state in range(num_states-1):
        max_action = max(range(num_actions), key=lambda action: Q[(state, action)])
        improved_policy[state] = max_action

    return improved_policy
```

Policy iteration

```
def policy_iteration():
    policy = {0:1, 1:2, 2:1, 3:1, 4:3, 5:1, 6:2, 7:3}
    while True:
        V = policy_evaluation(policy)
        improved_policy = policy_improvement(policy)

        if improved_policy == policy:
            break

        policy = improved_policy

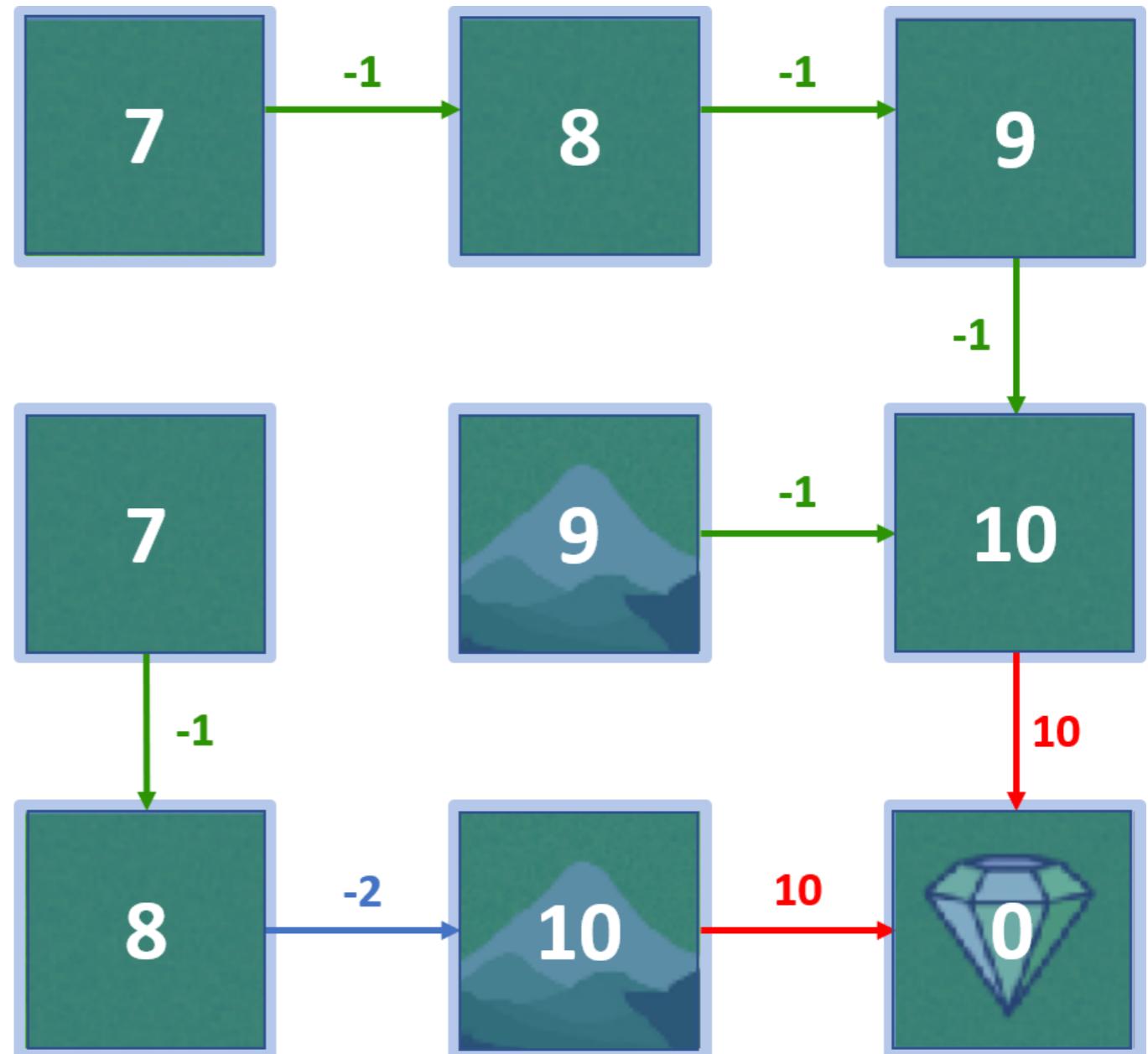
    return policy, V
```

Optimal policy

```
policy, V = policy_iteration()  
print(policy, V)
```

```
{0: 2, 1: 2, 2: 1,  
3: 1, 4: 2, 5: 1,  
6: 2, 7: 2}
```

```
{0: 7, 1: 8, 2: 9,  
3: 7, 4: 9, 5: 10,  
6: 8, 7: 10, 8: 0}
```



Value iteration

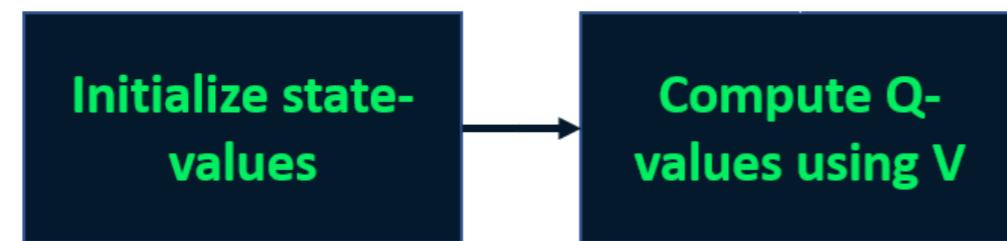
- Combines policy evaluation and improvement in one step
 - Computes optimal state-value function
 - Derives policy from it

Initialize state-values

$$v \begin{array}{|c|c|c|} \hline 0 & 0 & 0 \\ \hline 0 & 0 & 0 \\ \hline 0 & 0 & 0 \\ \hline \end{array}$$

Value iteration

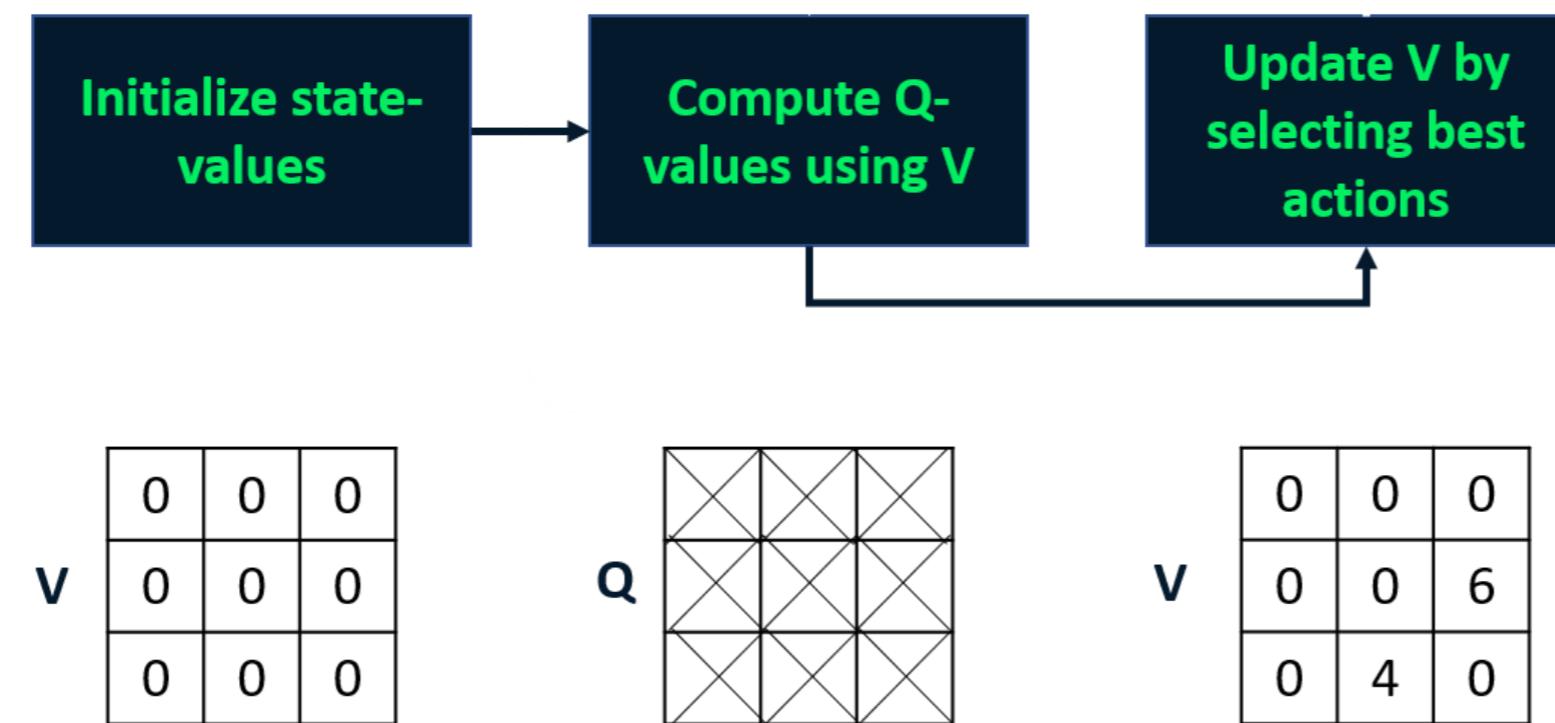
- Combines policy evaluation and improvement in one step.
 - Computes optimal state-value function
 - Derives policy from it



$$V = \begin{matrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{matrix}$$
$$Q = \begin{matrix} \diagup & \diagdown & & \\ \diagdown & \diagup & & \\ & & \diagup & \diagdown \\ \diagup & \diagdown & & \\ \diagdown & \diagup & & \\ & & \diagup & \diagdown \\ & & \diagdown & \diagup \\ \diagup & \diagdown & & \\ \diagdown & \diagup & & \end{matrix}$$

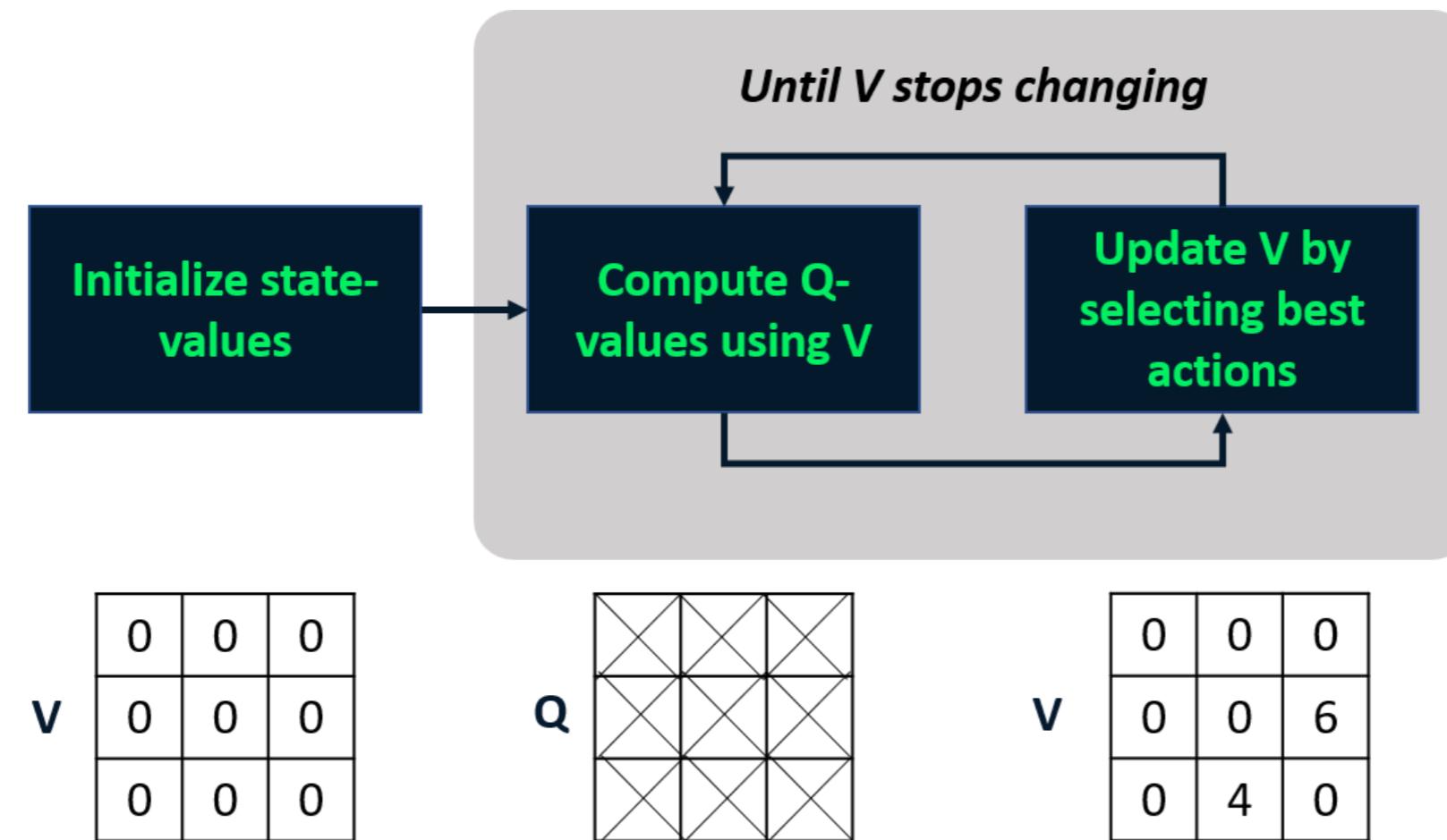
Value iteration

- Combines policy evaluation and improvement in one step.
 - Computes optimal state-value function
 - Derives policy from it



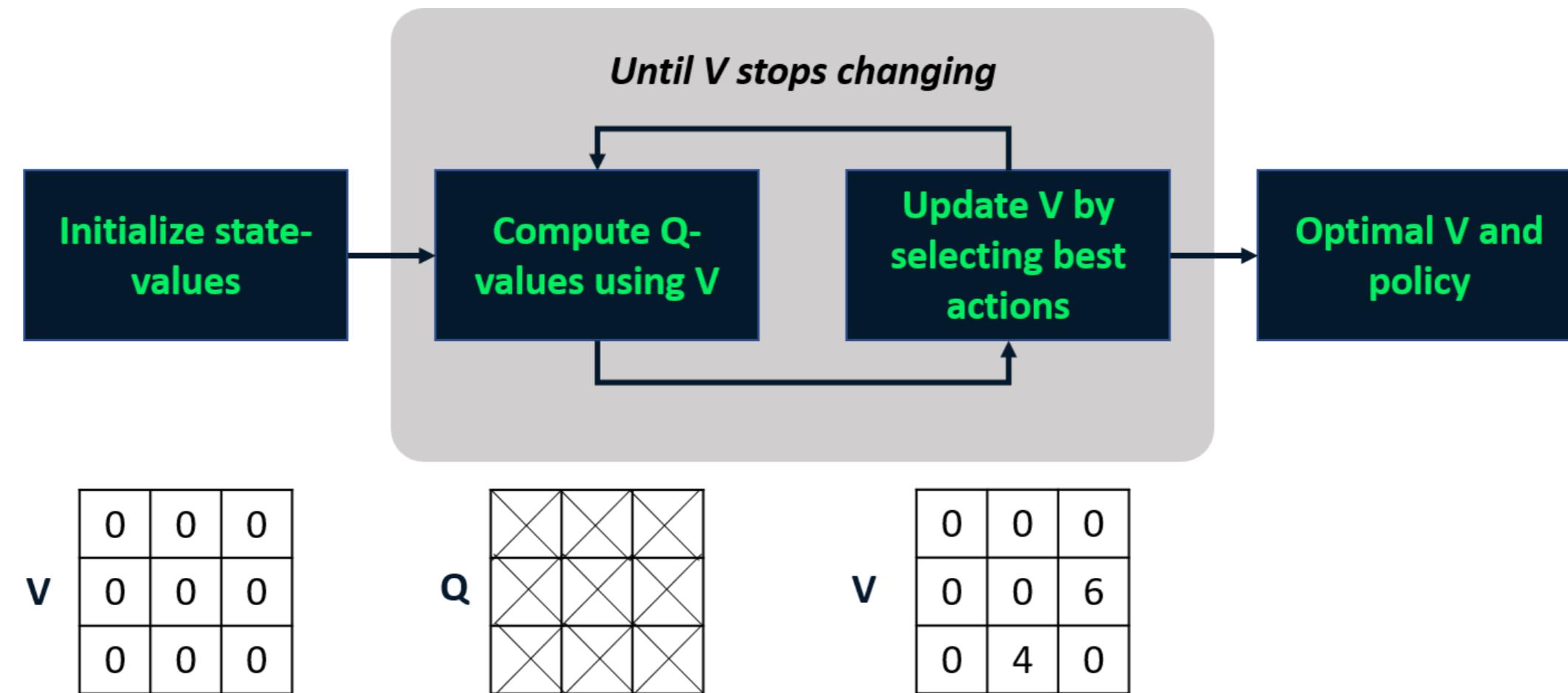
Value iteration

- Combines policy evaluation and improvement in one step.
 - Computes optimal state-value function
 - Derives policy from it



Value iteration

- Combines policy evaluation and improvement in one step.
 - Computes optimal state-value function
 - Derives policy from it



Implementing value-iteration

```
V = {state: 0 for state in range(num_states)}
policy = {state:0 for state in range(num_states-1)}
threshold = 0.001

while True:
    new_V = {state: 0 for state in range(num_states)}
    for state in range(num_states-1):
        max_action, max_q_value = get_max_action_and_value(state, V)
        new_V[state] = max_q_value
        policy[state] = max_action

    if all(abs(new_V[state] - V[state]) < thresh for state in V):
        break
    V = new_V
```

Getting optimal actions and values

```
def get_max_action_and_value(state, V):
    Q_values = [compute_q_value(state, action, V) for action in range(num_actions)]
    max_action = max(range(num_actions), key=lambda a: Q_values[a])
    max_q_value = Q_values[max_action]
    return max_action, max_q_value
```

Computing Q-values

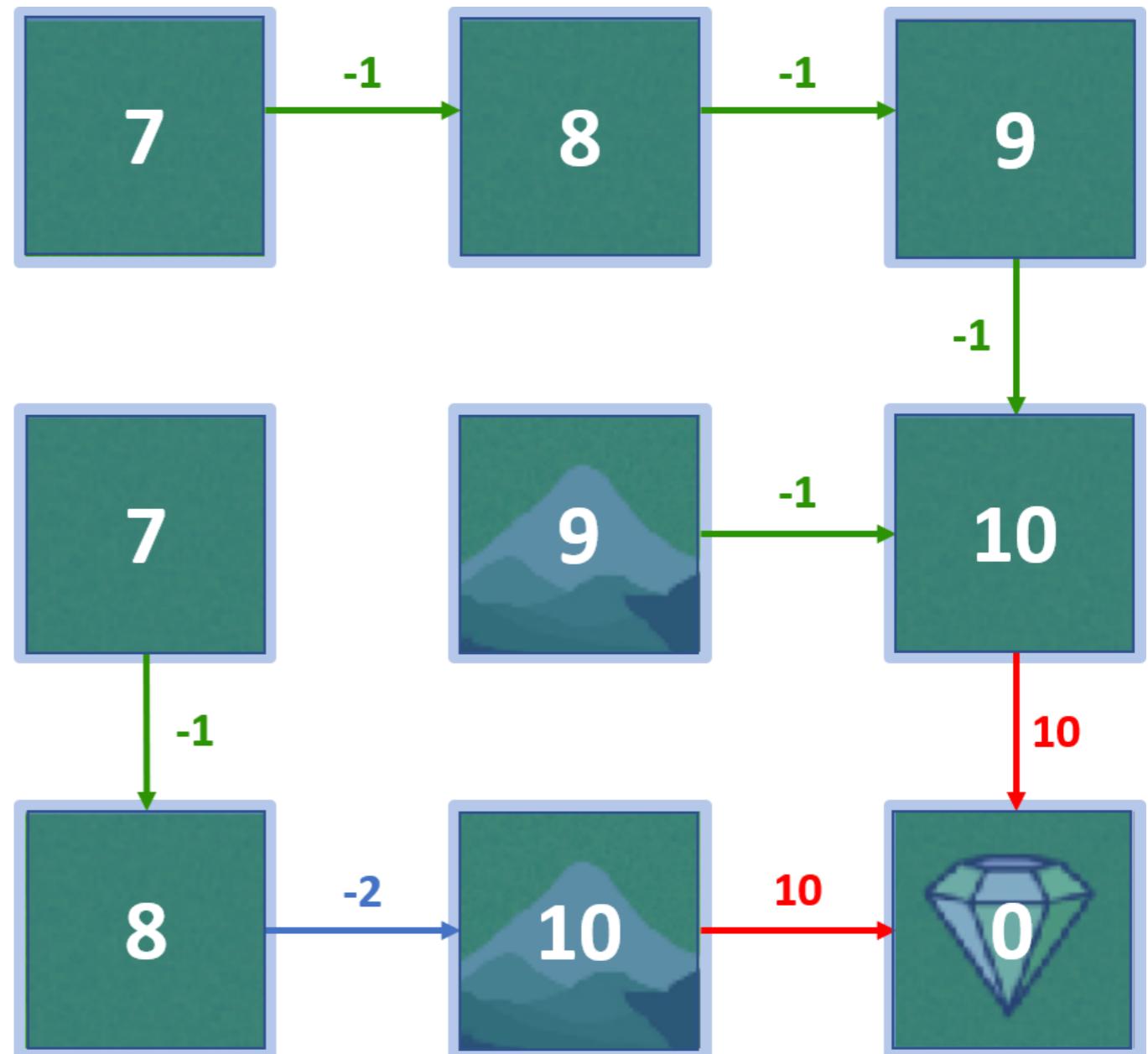
```
def compute_q_value(state, action, V):
    if state == terminal_state:
        return None
    _, next_state, reward, _ = env.P[state][action][0]
    return reward + gamma * V[next_state]
```

Optimal policy

```
print(policy, V)
```

```
{0: 2, 1: 2, 2: 1,  
3: 1, 4: 2, 5: 1,  
6: 2, 7: 2}
```

```
{0: 7, 1: 8, 2: 9,  
3: 7, 4: 9, 5: 10,  
6: 8, 7: 10, 8: 0}
```



Let's practice!

REINFORCEMENT LEARNING WITH GYMNASIUM IN PYTHON