Reinforcement Learning

Lab #2: Markov Decision Process

Daniil Arapov

Partially based on msu.ai lecture #15

Test questions for today:

1. What is the difference between v and q functions?
2. Derive Bellman equation for value function and for action-value function
3. Given two-armed bandit with the following arms:

$$R_1 = r_1 \qquad (p = 100\%)$$

$$R_2 = \begin{cases} r_2, & (p = 99.9\%), \\ -\infty, & (p = 0.1\%). \end{cases} \qquad 0 < r_1 < r_2$$

   a. Define action probabilities for eps-greedy policy
   b. Define action probabilities for softmax-based policy
   c. When $p_\epsilon(a_2) = p_{softmax}(a_2)$?
   d. When $p_\epsilon(a_2) > p_{softmax}(a_2)$?

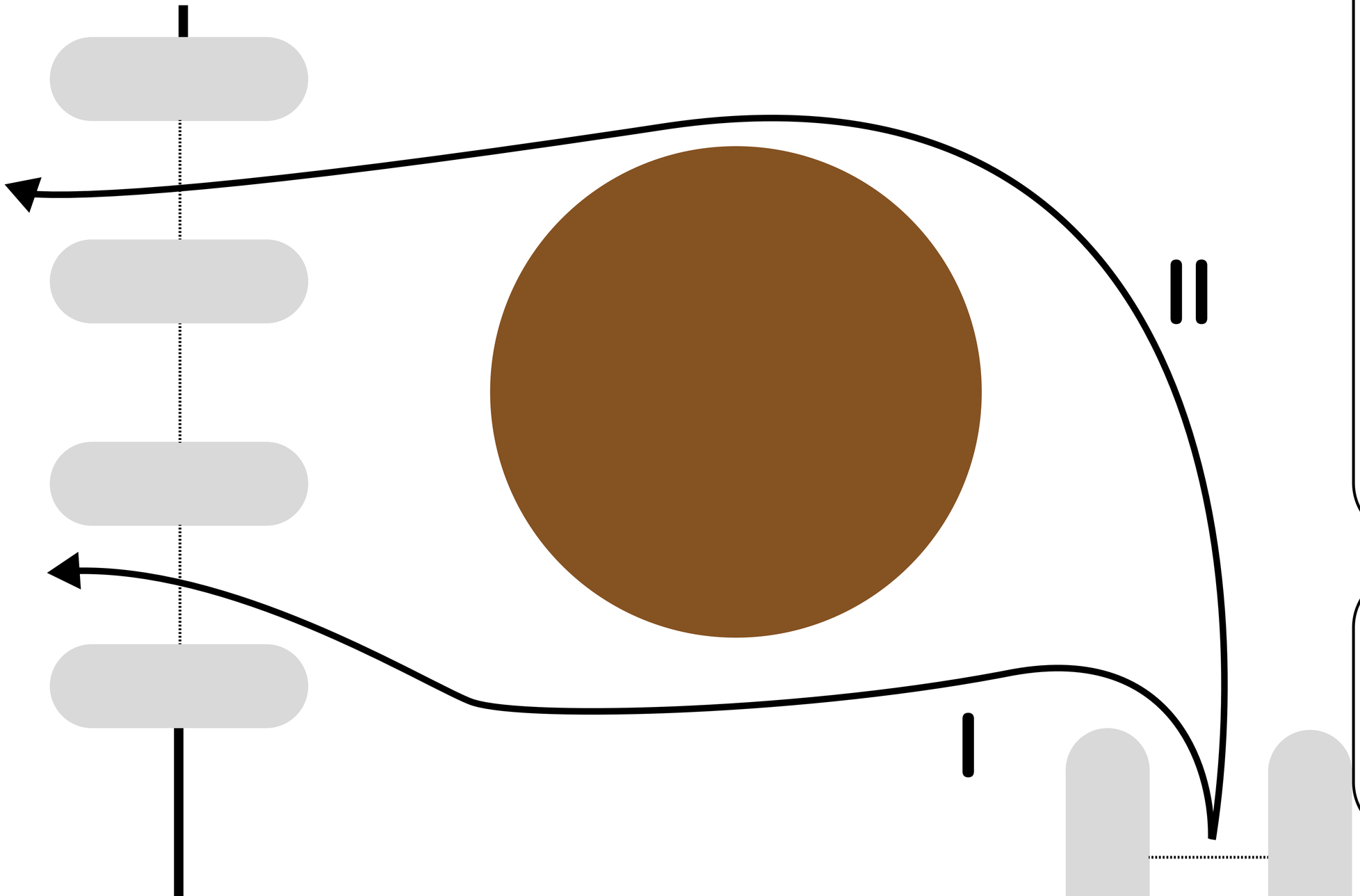
Survivorship bias

Sometimes we do not really need an exploitation, as it was shown.

Survivorship bias - we do not have negative experience although there is a chance to loose too much. *Dead Men Tell No Tales.*

Previously we have seen that exploitation (overfitting) is good, but what was the condition?

Ground floor turnstile problem



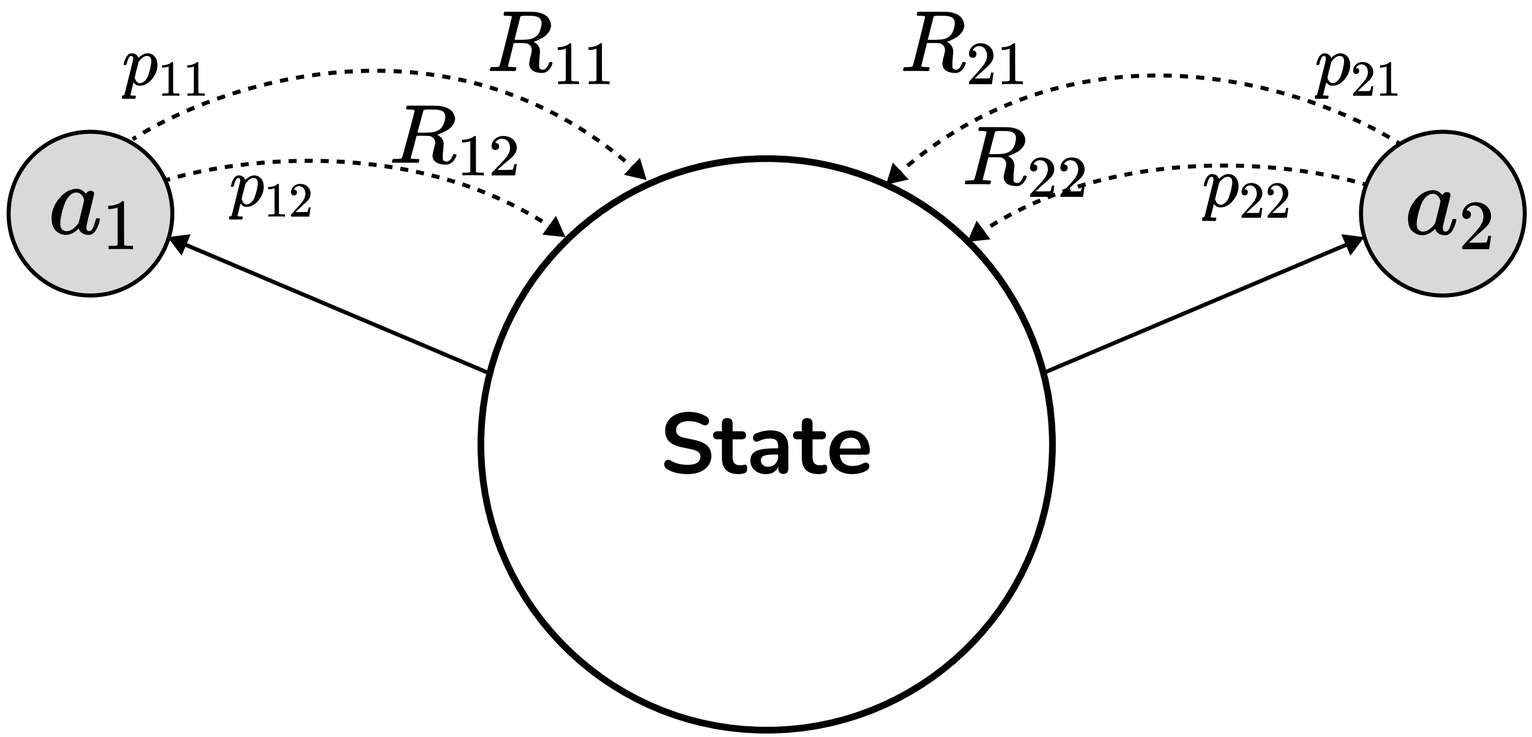
Option **I**:
+ faster reach destination
- wait while db processes exit

Option **II**:
+ no db wait
- takes longer

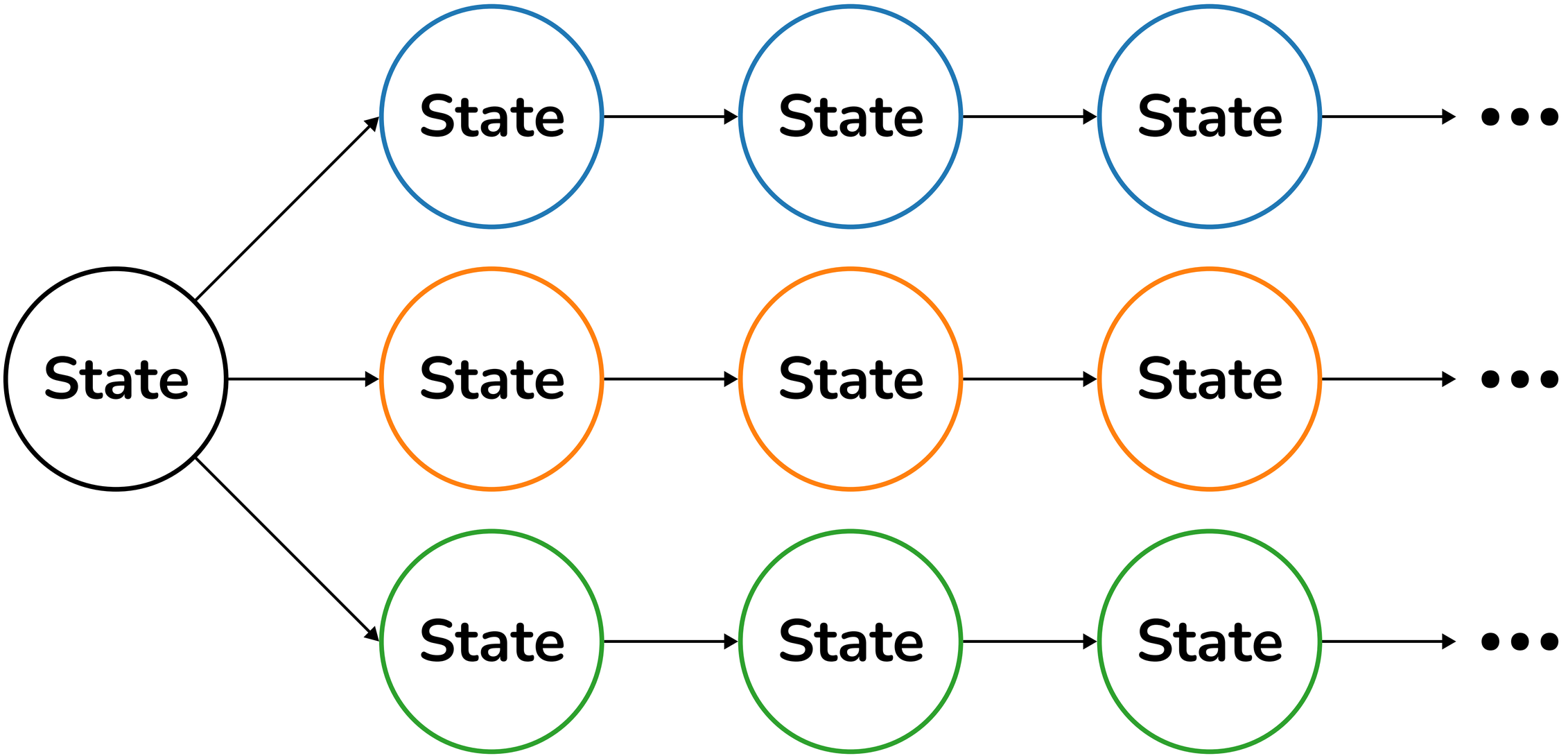
Now **II** is better; when db is fixed, **I** is better.

Markov Decision Process

Stateless as Markov Decision Process

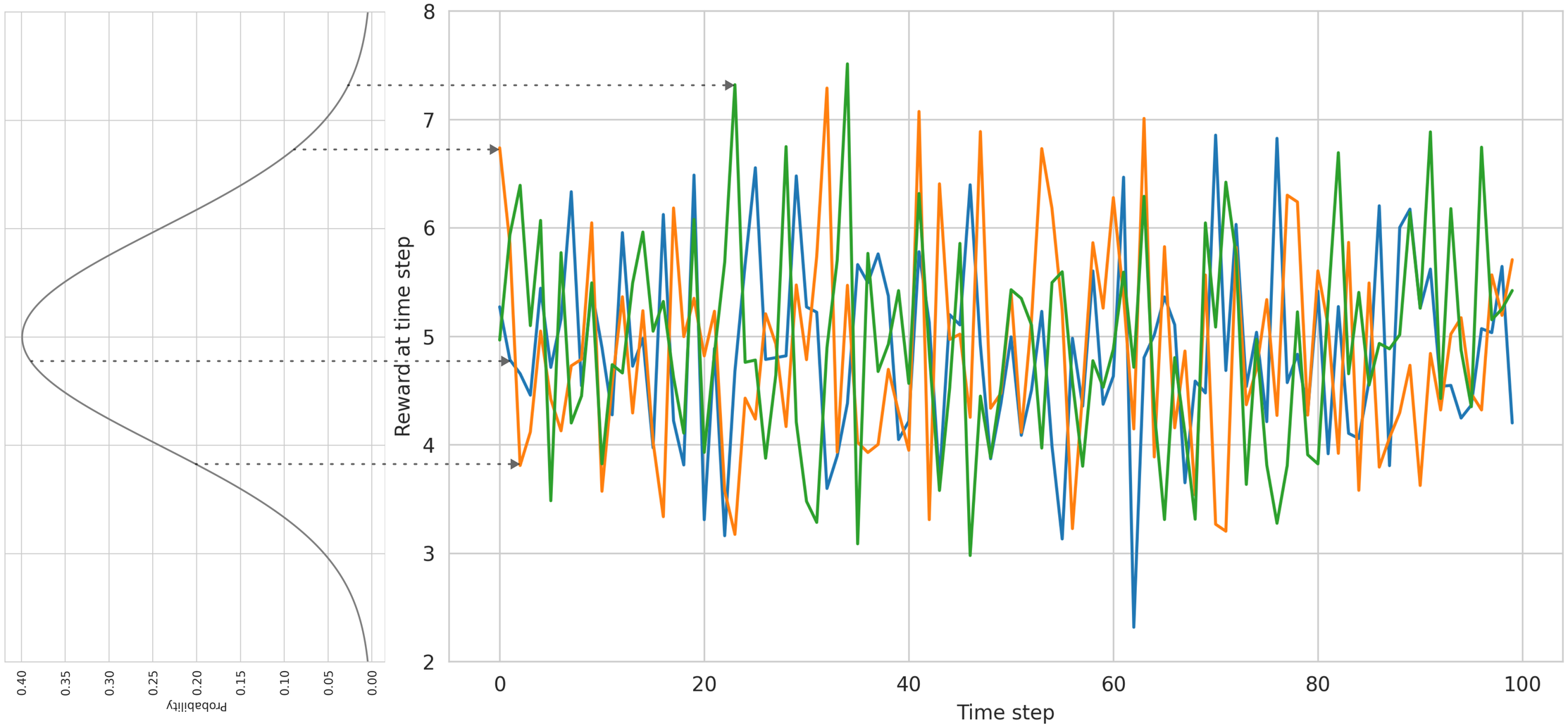


Naive Markov Decision Process



Choosing the path

Naive Markov Decision Process Rewards

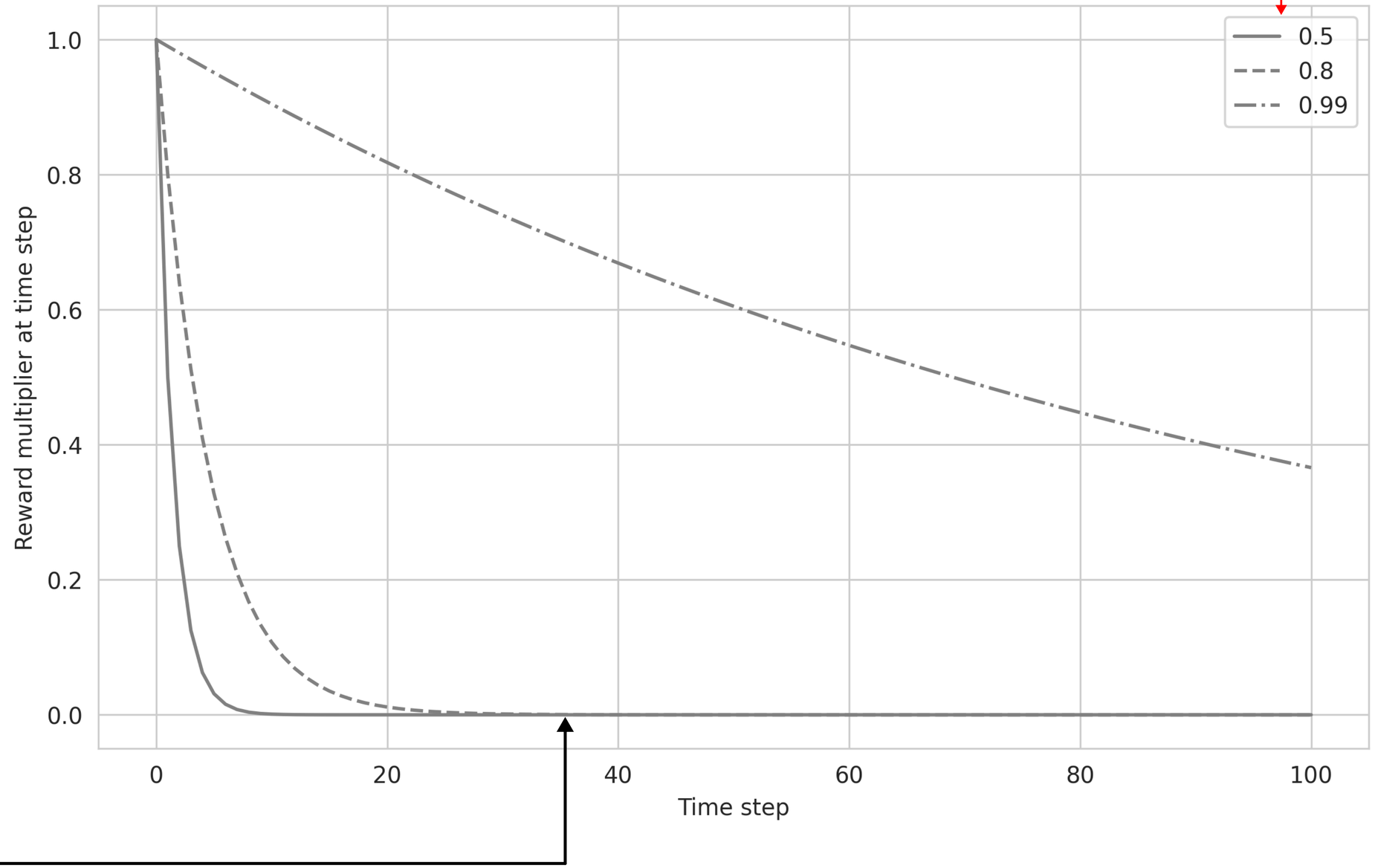


Reward distribution

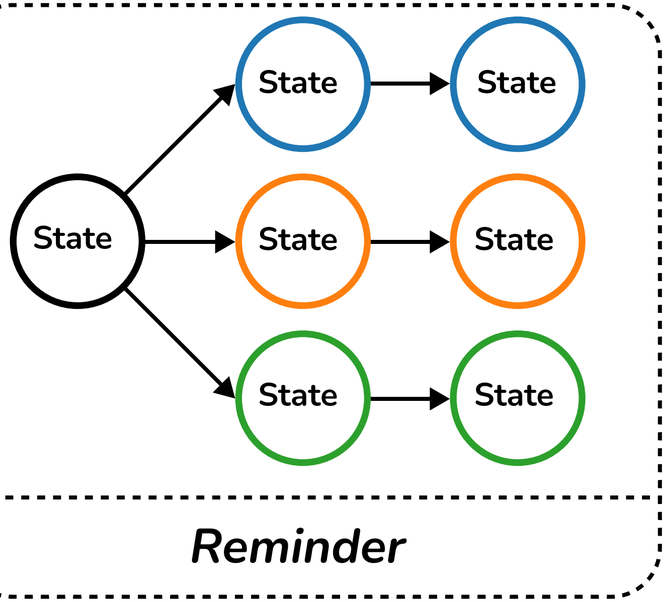
Generated reward values

Discount factor multipliers

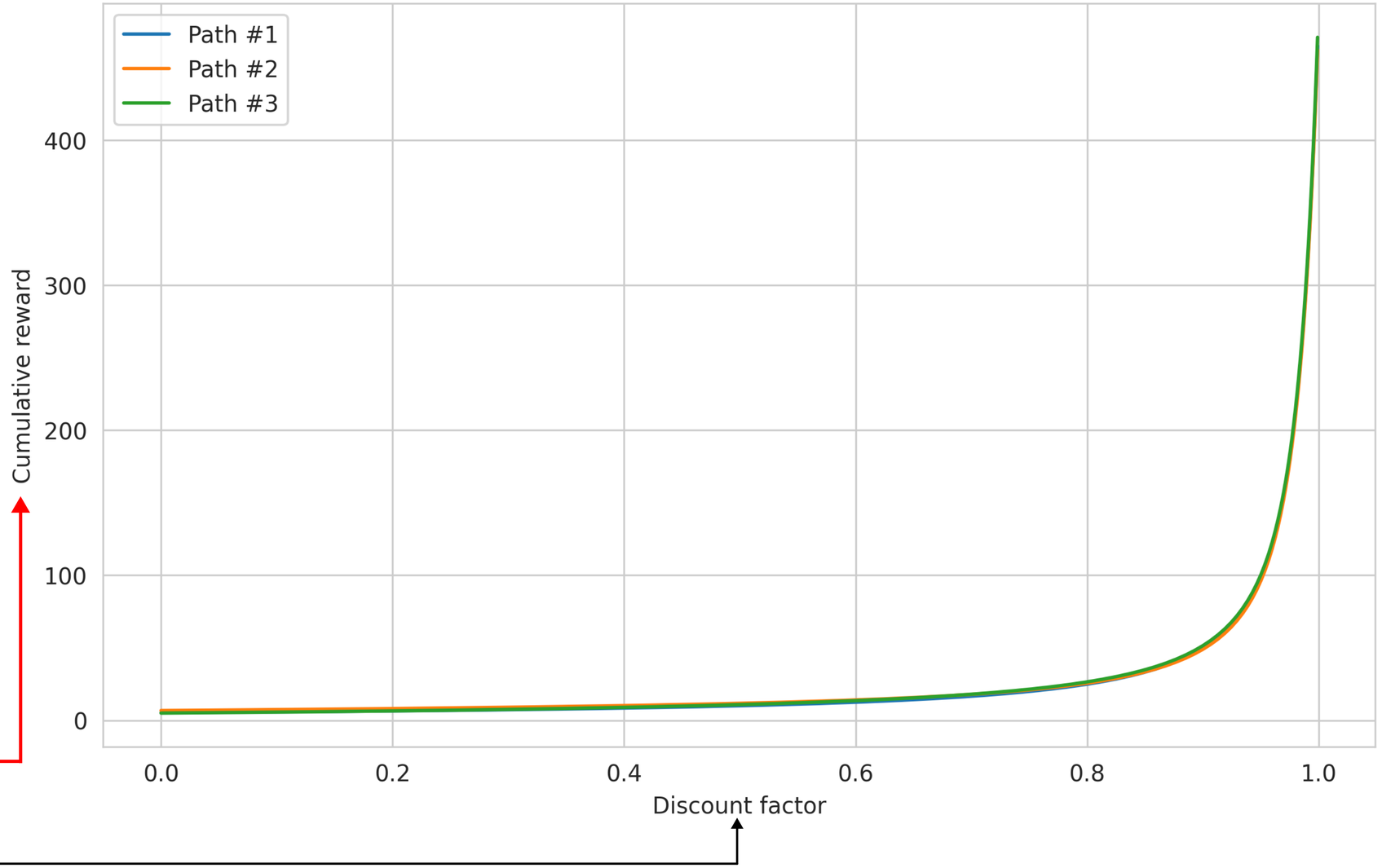
$$\sum_{i=0}^{N} R_i \cdot \gamma^i$$



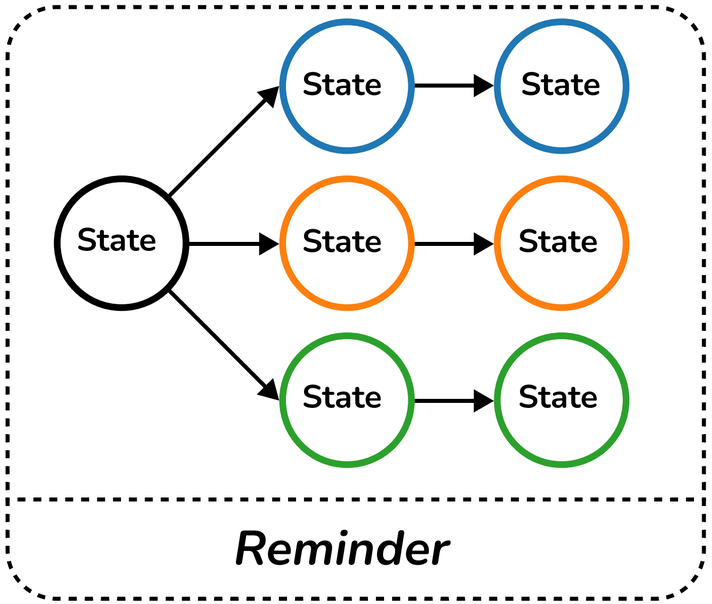
Changing discount rate



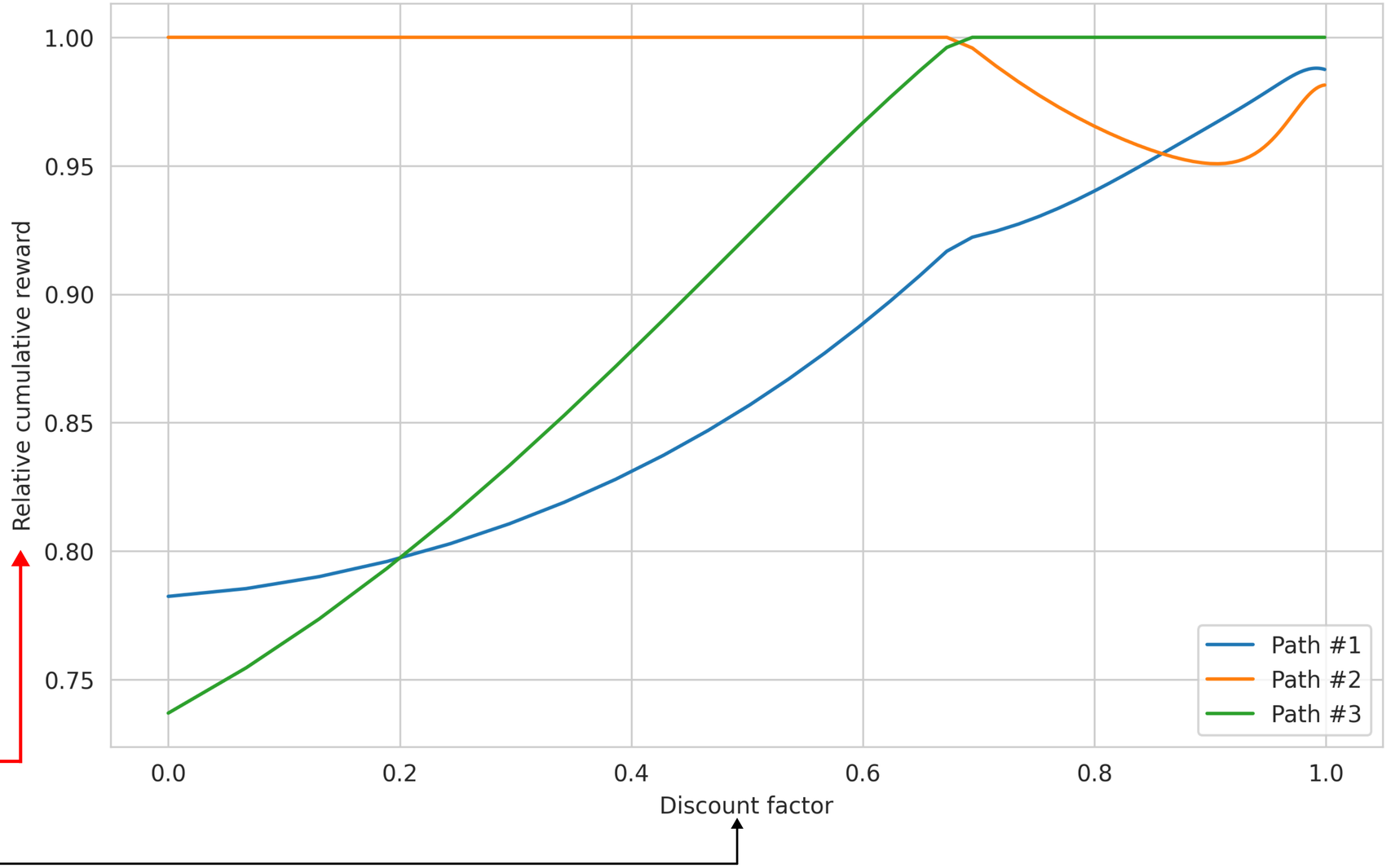
$$\sum_{i=0}^{N} R_i \cdot \gamma^i$$



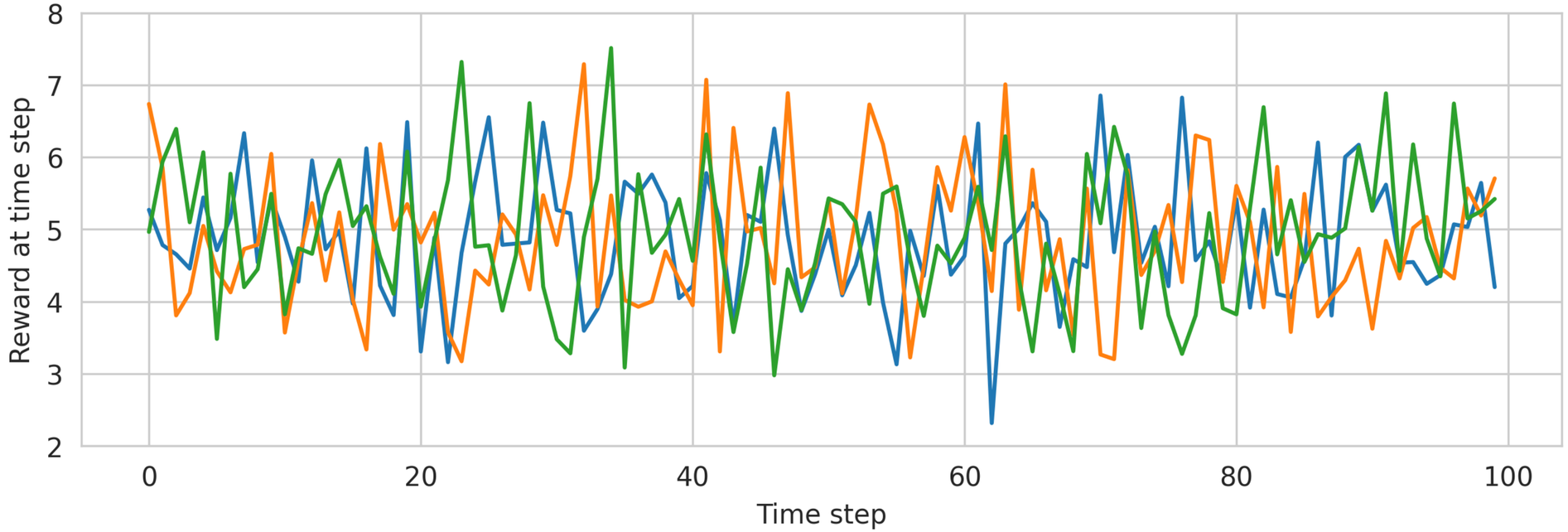
Changing discount rate (normalized)

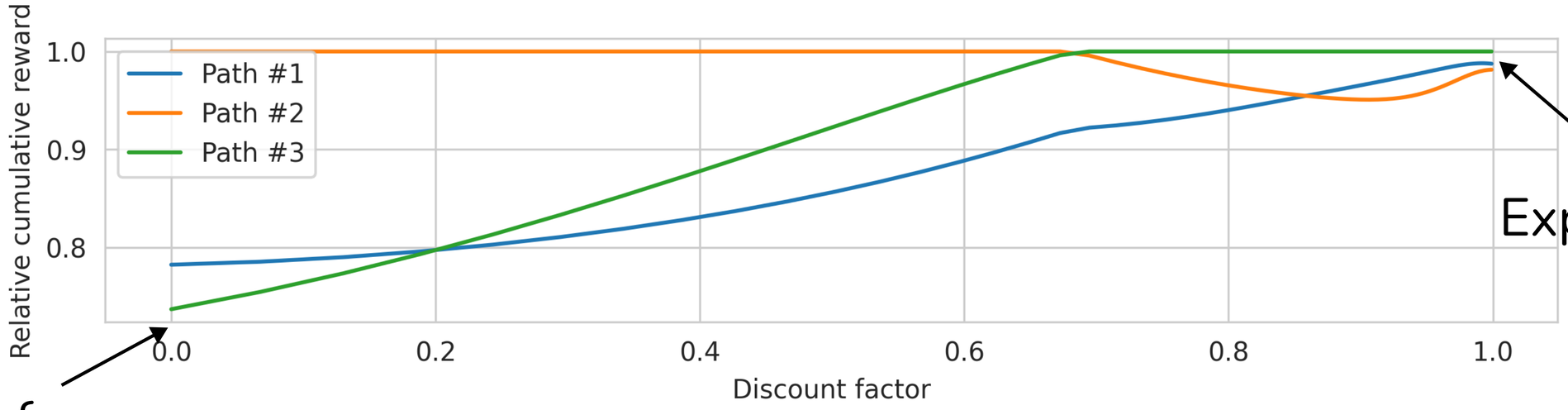


$$\sum_{i=0}^{N} R_i \cdot \gamma^i$$



Analyzing the results





Expectation

Local info

Solving naive MDP (Create an environment)

```
# hyperparams
T = 100
path_cnt = 3
gamma = 0.9

# original value function approx
random_gen = stats.norm(loc = 0, scale = 1)
value_model = random_gen.rvs((path_cnt, T+1))
value_model[:, -1] = 0 # terminate value

# action rewards
base_distr = stats.norm(loc = 5, scale = 1)
action_rewards = np.array([base_distr.rvs(T) for _ in range(path_cnt)])
```

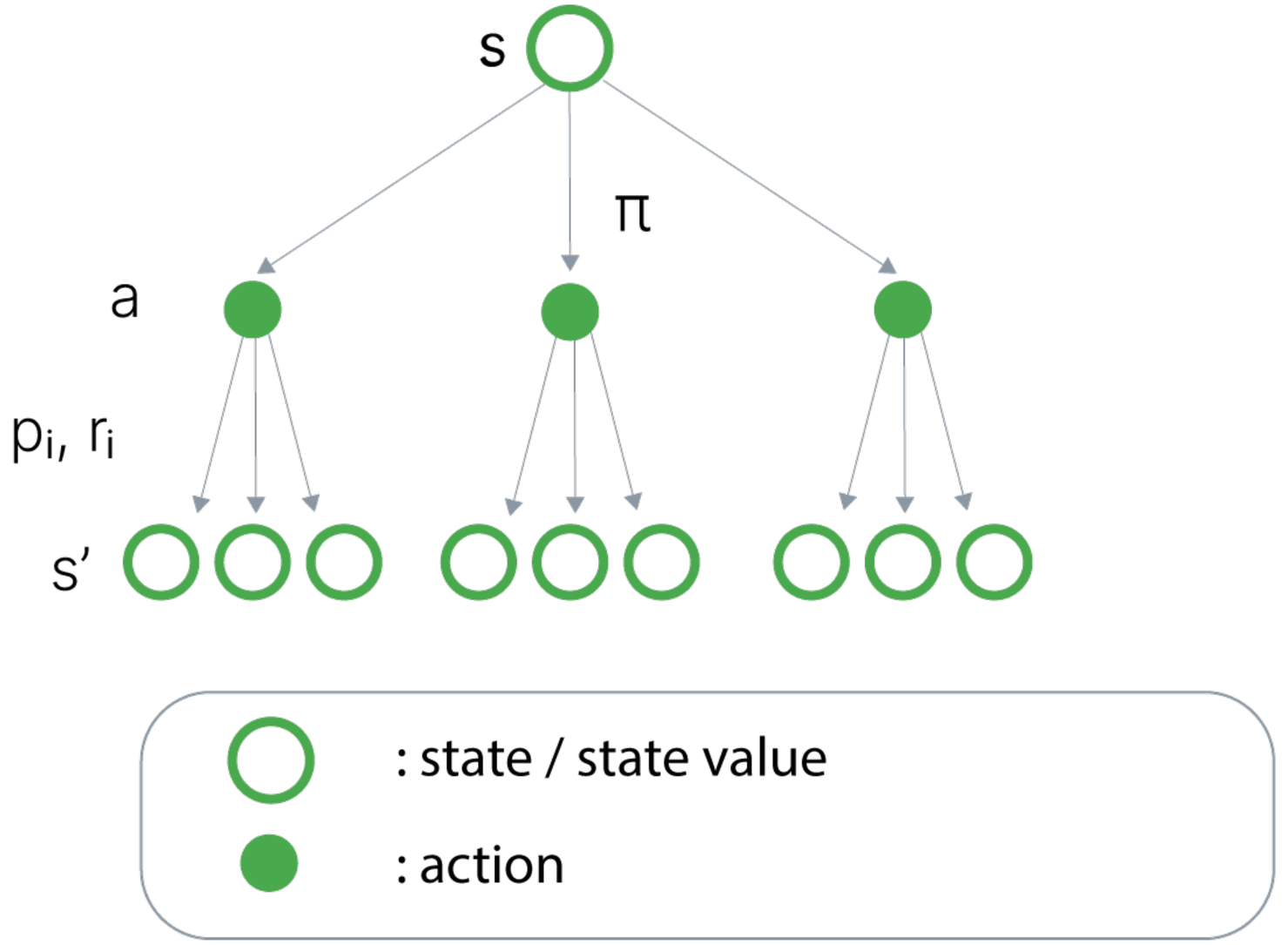
Solving naive MDP (Dynamic Programming)

```
dp_steps = 1000
corrections = []

for iter_num in range(dp_steps):
    value_model_new = value_model.copy()

    # iterating over the states
    for path in range(path_cnt):
        for t in range(T): # except terminate values
            # the only possible action and next state
            value_model_new[path, t] = action_rewards[path, t] + \
                                       value_model[path, t + 1] * gamma

    correction = np.abs(value_model_new - value_model).sum()
    corrections.append(correction)
```

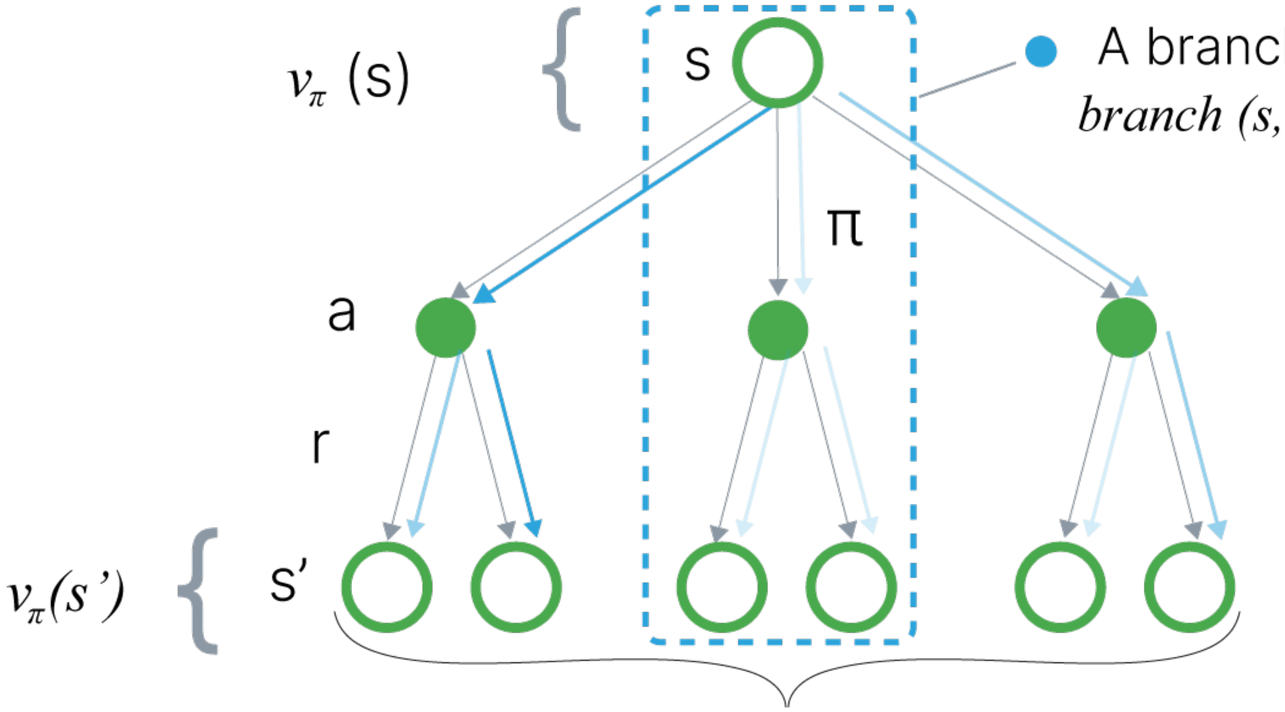
Converged in 100 iters

Simple MDP problem
Material based on msu.ai lec#15

A backup diagram of State Value $v_\pi(s)$



A backup diagram of Bellman-equation-like recurrence relation



A branch Bellman backup diagram

$$branch\ (s,\ a) = \sum_a \pi(a|s) \sum_{s'} P^a_{ss'} (R^a_{ss'} + \gamma\ v_\pi(s'))$$

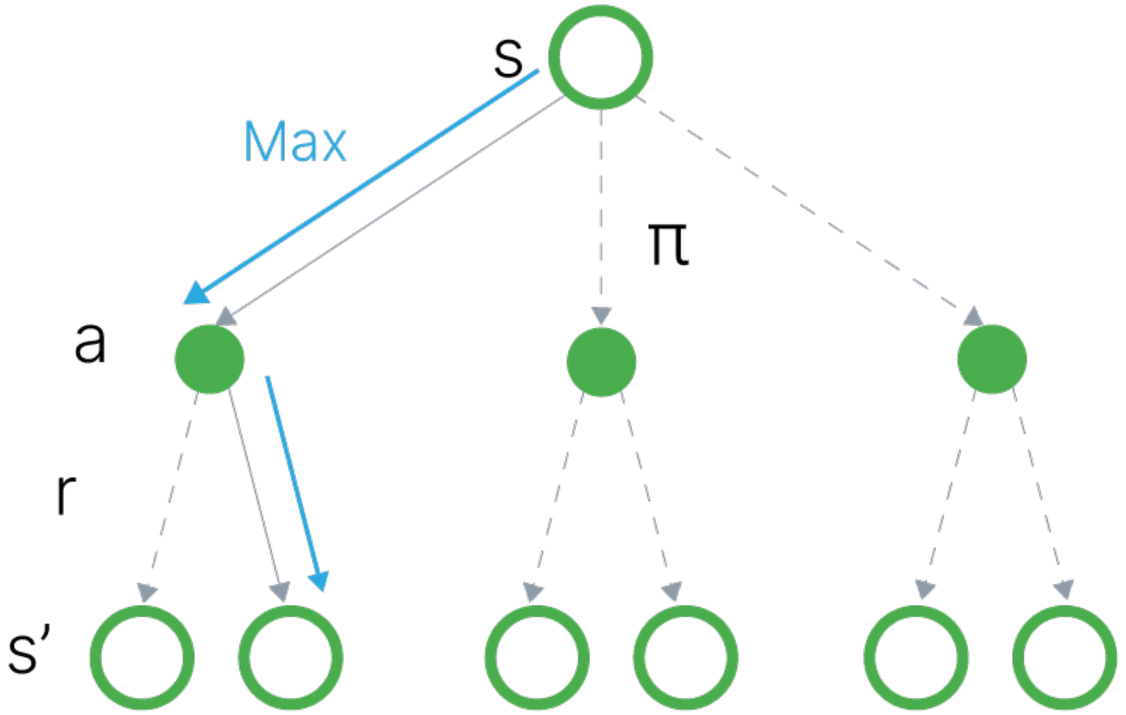
$v_\pi(s)$ is a probabilistic sum of all branches branch $(s,\ a)$ weighted by π(a|s)

: state / state value

: probability of taking the path

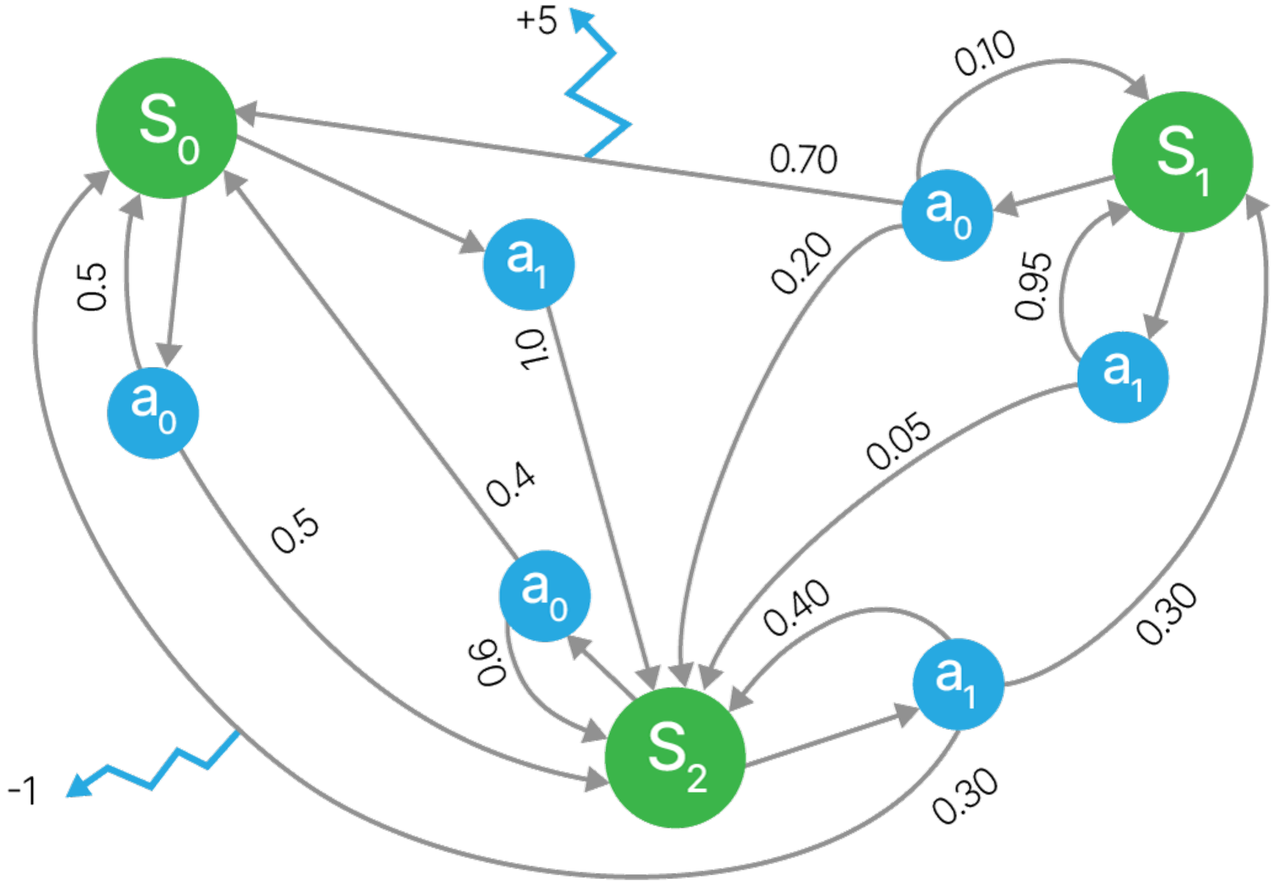
: action

A backup diagram of the Bellman optimality equation



$V_*(s)$ is the maximum value of $\sum_a \pi(a|s) \sum_{s'} P^a_{ss'} \left[R^a_{ss'} + \gamma V_*(s') \right]$

Markov Decision Process



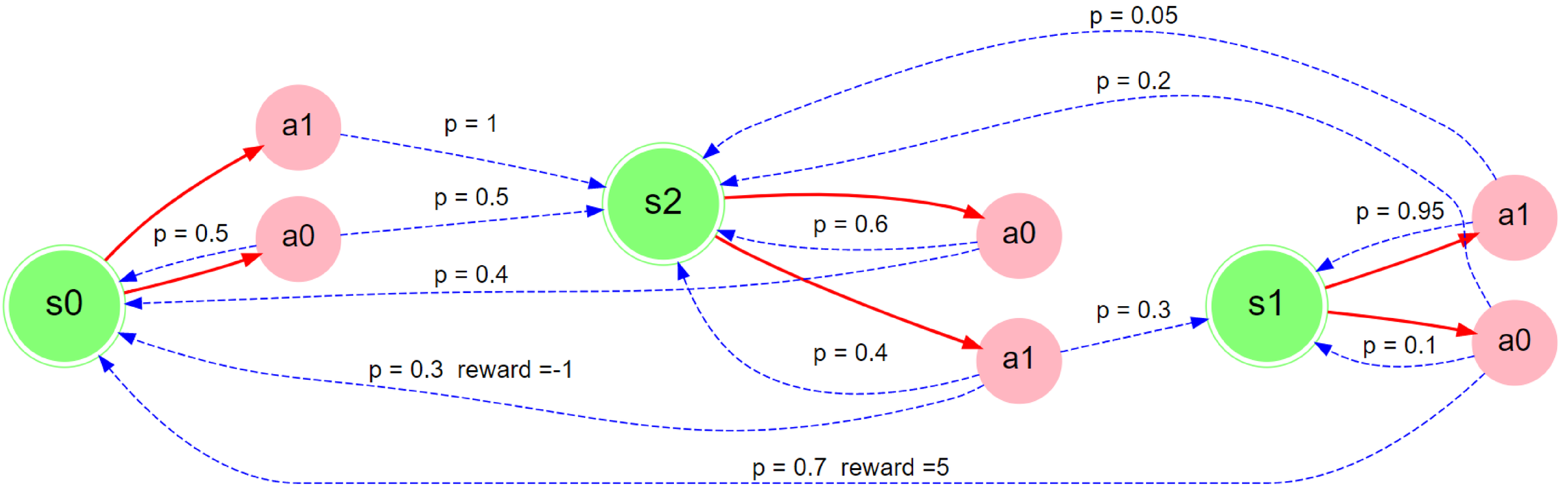

```
transition_probs = {
    's0': {
        'a0': {'s0': 0.5, 's2': 0.5},
        'a1': {'s2': 1}
    },
    's1': {
        'a0': {'s0': 0.7, 's1': 0.1, 's2': 0.2},
        'a1': {'s1': 0.95, 's2': 0.05}
    },
    's2': {
        'a0': {'s0': 0.4, 's2': 0.6},
        'a1': {'s0': 0.3, 's1': 0.3, 's2': 0.4}
    }
}

rewards = {
    's1': {'a0': {'s0': +5}},
    's2': {'a1': {'s0': -1}}
}


mdp = MDP(transition_probs, rewards, initial_state='s0')
```

MDP contents

```
# MDP methods

print("mdp.get_all_states =", mdp.get_all_states())
print("mdp.get_possible_actions('s1') = ", mdp.get_possible_actions('s1'))
print("mdp.get_next_states('s1', 'a0') = ", mdp.get_next_states('s1', 'a0'))

# state, action, next_state
print("mdp.get_reward('s1', 'a0', 's0') = ", mdp.get_reward('s1', 'a0', 's0'))

# get_transition_prob(self, state, action, next_state)
print("mdp.get_transition_prob('s1', 'a0', 's0') = ", mdp.get_transition_prob('s1', 'a0', 's0'))
```

```
mdp.get_all_states = ('s0', 's1', 's2')
mdp.get_possible_actions('s1') =  ('a0', 'a1')
mdp.get_next_states('s1', 'a0') =  {'s0': 0.7, 's1': 0.1, 's2': 0.2}
mdp.get_reward('s1', 'a0', 's0') =  5
mdp.get_transition_prob('s1', 'a0', 's0') =  0.7
```

MDP visualization



Value iteration

1. Initialize $V_{(0)}(s) = 0$, for all s
2. For $i = 0, 1, 2, \ldots$
3. $V_{(i+1)}(s) = \max_a \sum_{s'} P^a_{ss'} \cdot [R^a_{ss'} + \gamma V_i(s')]$, for all s

```python
def get_new_state_value(mdp, state_values, state, gamma):
    """ Computes next V(s) as in formula above. Please do not change state_values in process. """
    if mdp.is_terminal(state):
      return 0 # Game over

    q_max = float('-inf')
    actions = mdp.get_possible_actions(state)
    for a in actions:
      q = get_action_value(mdp, state_values, state, a, gamma)
      q_max = max(q_max, q)
    return q_max
```

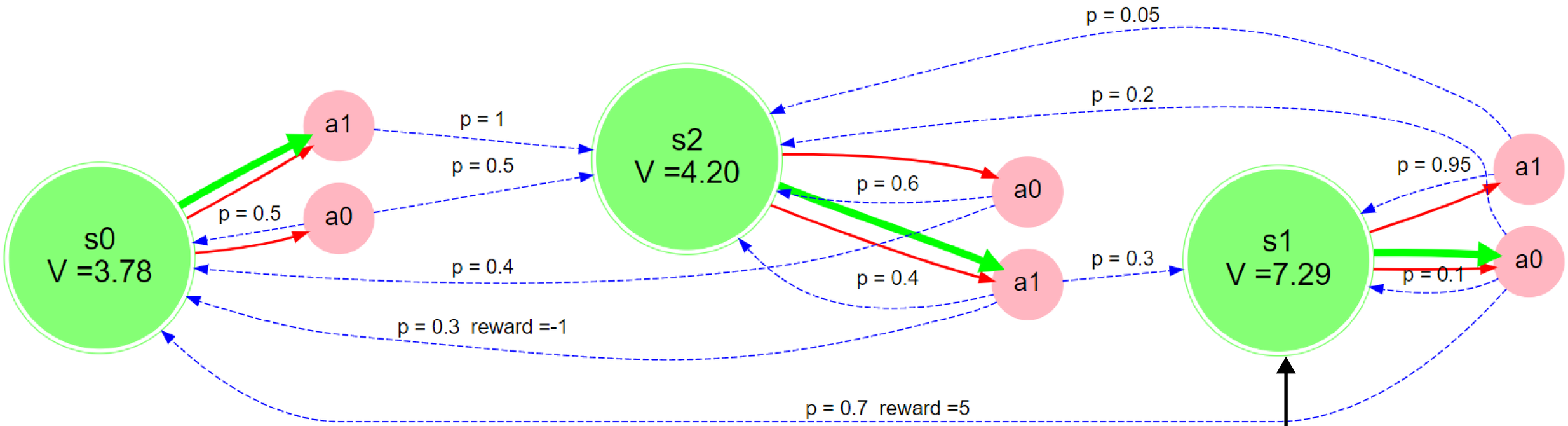
Action value computation

```
def get_action_value(mdp, state_values, state, action, gamma):
  """
  Computes Q(s,a) as in formula above

  mdp : MDP object
  state_values : dictionary of { state_i : V_i }
  state: string id of current state
  gamma: float discount coeff

  """


  next_states = mdp.get_next_states(state, action)

  Q = 0.0

  for next_state in next_states.keys():
    p = next_states[next_state] # alternatively p = mdp.get_transition_prob(state, action, next_state)
    Q += p * (mdp.get_reward(state, action, next_state) + gamma * state_values[next_state])
  return Q
```

Value iteration application

```
# parameters
gamma = 0.9             # discount for MDP
num_iter = 100          # maximum iterations, excluding initialization
# stop VI if new values are this close to old values (or closer)
min_difference = 0.001

# initialize V(s)
state_values = {s: 0 for s in mdp.get_all_states()}

display(MDP.plot_graph_with_state_values(mdp, state_values))

for i in range(num_iter):

    # Compute new state values using the functions you defined above.
    # It must be a dict {state : float V_new(state)}

    new_state_values = {}
    for s in state_values.keys():
      new_state_values[s] = get_new_state_value(mdp, state_values, s, gamma)
```

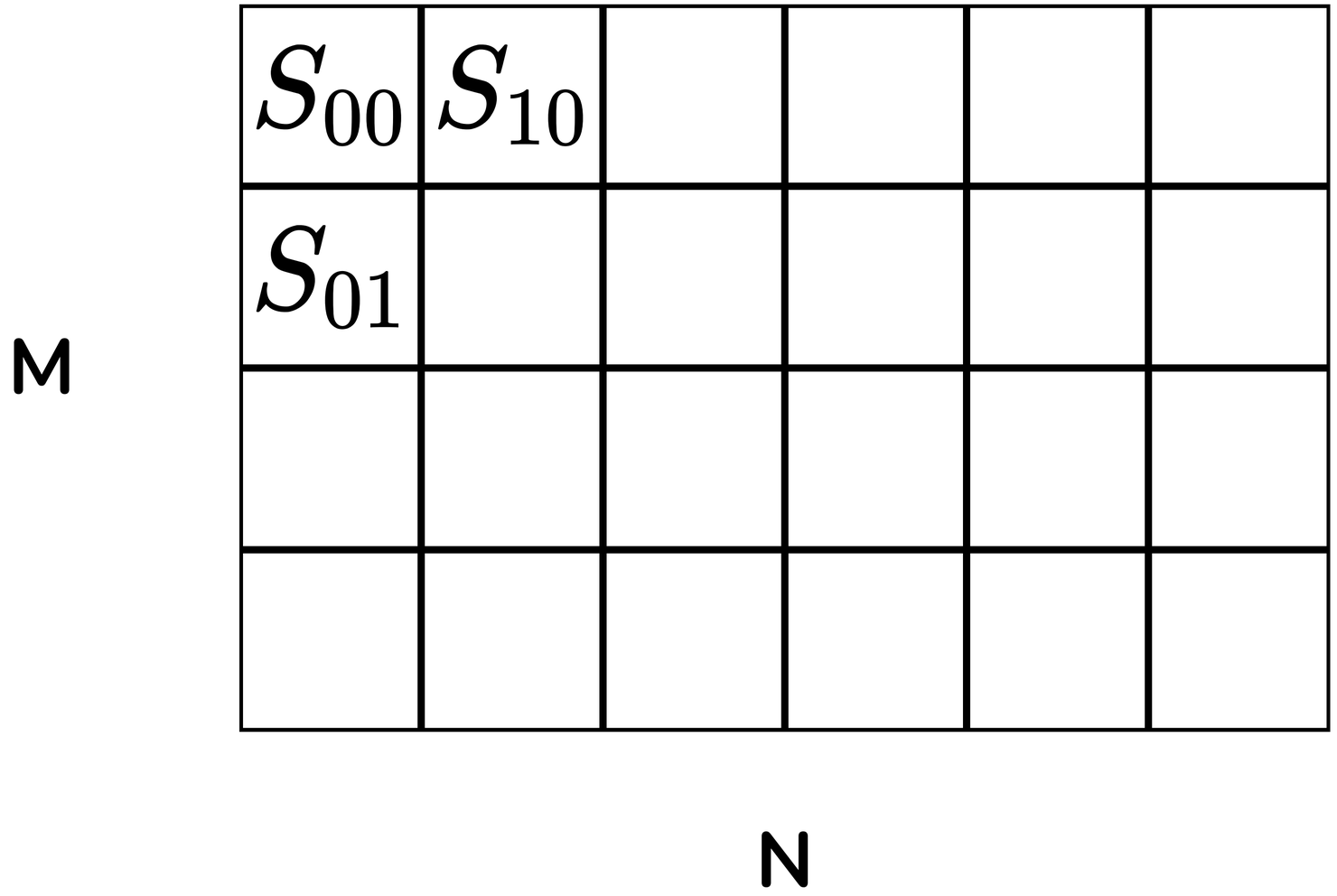
Policy and Value function



$$0.1 \cdot 0.9 \cdot 7.29 + 0.7 \cdot 5 + 0.7 \cdot 0.9 \cdot 3.78 + 0.2 \cdot 0.9 \cdot 4.2 = 7.2935$$

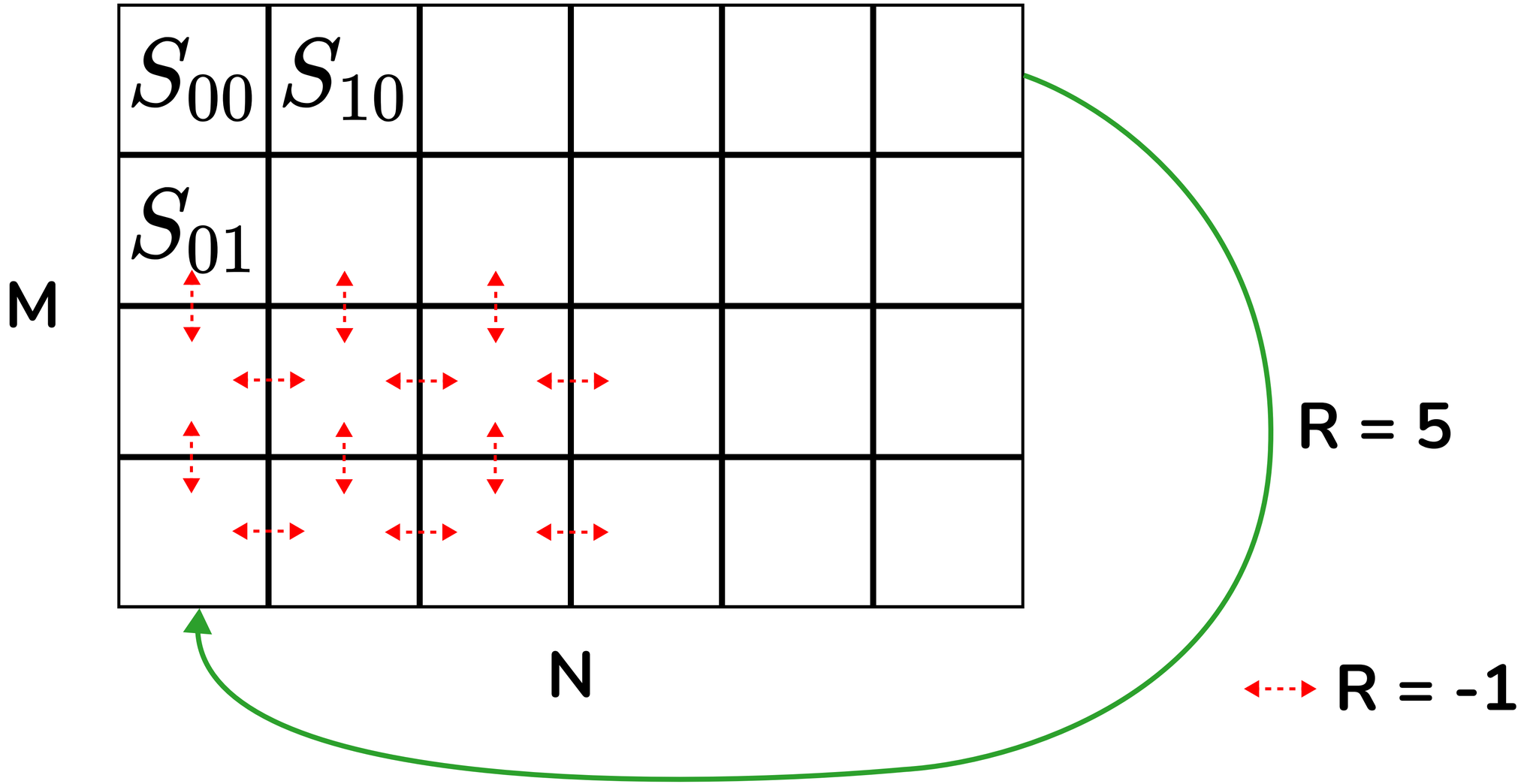
GridWorld Environment

GridWorld Environment



- Each cell is a state
- By default agent can move to adjacent states
- By default rewards are constant

Endless world



Endless world

```
[[-1. -1. -1. -1. -1.  5.]
 [-1. -1. -1. -1. -1. -1.]
 [-1. -1. -1. -1. -1. -1.]
 [-1. -1. -1. -1. -1. -1.]]
```

t = 1

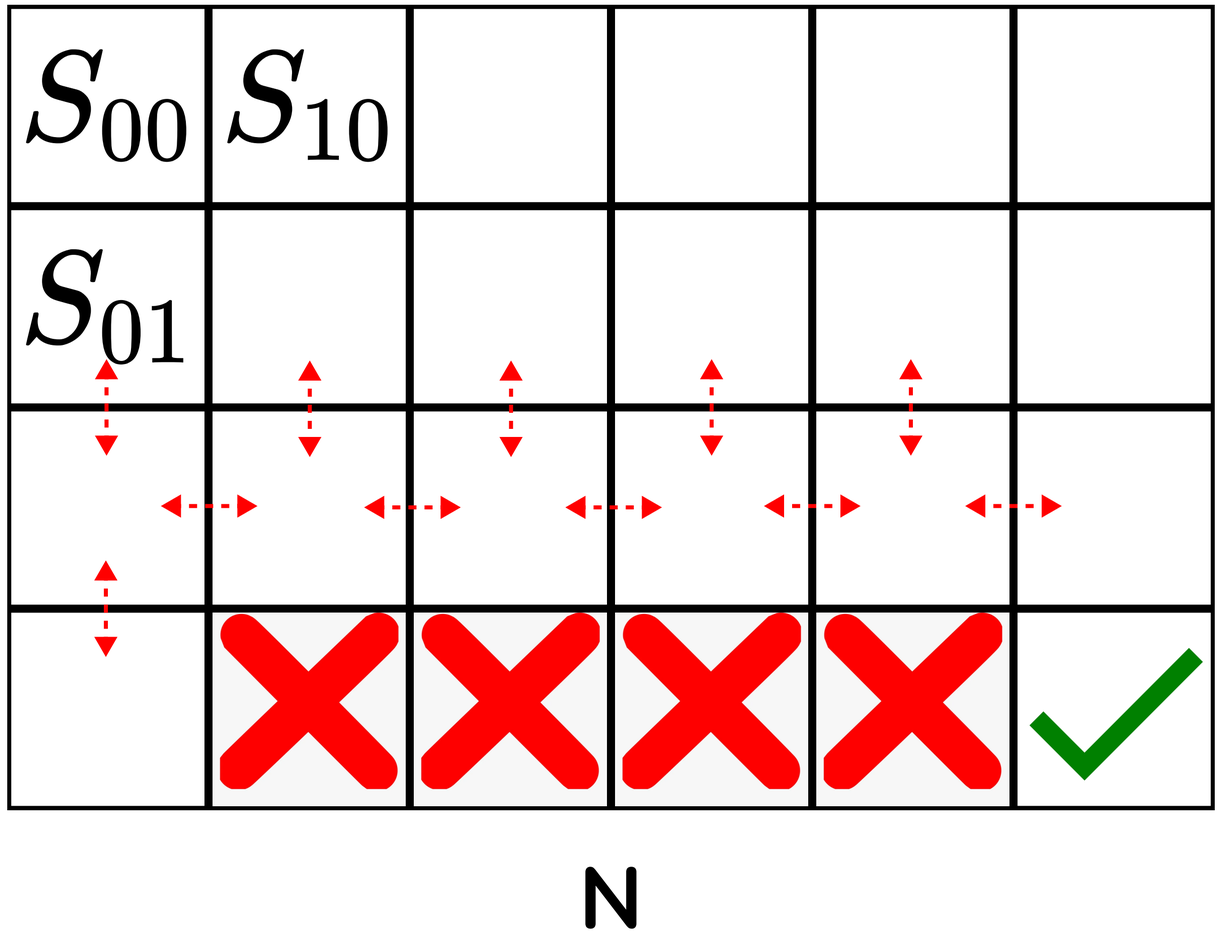
```
[[-1.11 -1.11 -1.11 -1.05 -0.51  4.89]
 [-1.11 -1.11 -1.11 -1.11 -1.05 -0.51]
 [-1.11 -1.11 -1.11 -1.11 -1.11 -1.05]
 [-1.11 -1.11 -1.11 -1.11 -1.11 -1.11]]
```

t = 3

```
[[-1.11105111 -1.11051111 -1.10511111 -1.05111111 -0.51111111  4.88888889]
 [-1.11110511 -1.11105111 -1.11051111 -1.10511111 -1.05111111 -0.51111111]
 [-1.11111051 -1.11110511 -1.11105111 -1.11051111 -1.10511111 -1.05111111]
 [-1.11111105 -1.11111051 -1.11110511 -1.11105111 -1.11051111 -1.10511111]]
```

t = 99

Terminating world



R = -1

Terminating states values:

V = 10

V = -10

Terminating world

```
[[  0.   0.   0.   0.   0.   0.]
 [  0.   0.   0.   0.   0.   0.]
 [  0.   0.   0.   0.   0.   0.]
 [  0. -10. -10. -10. -10.  10.]]
```

t = 0

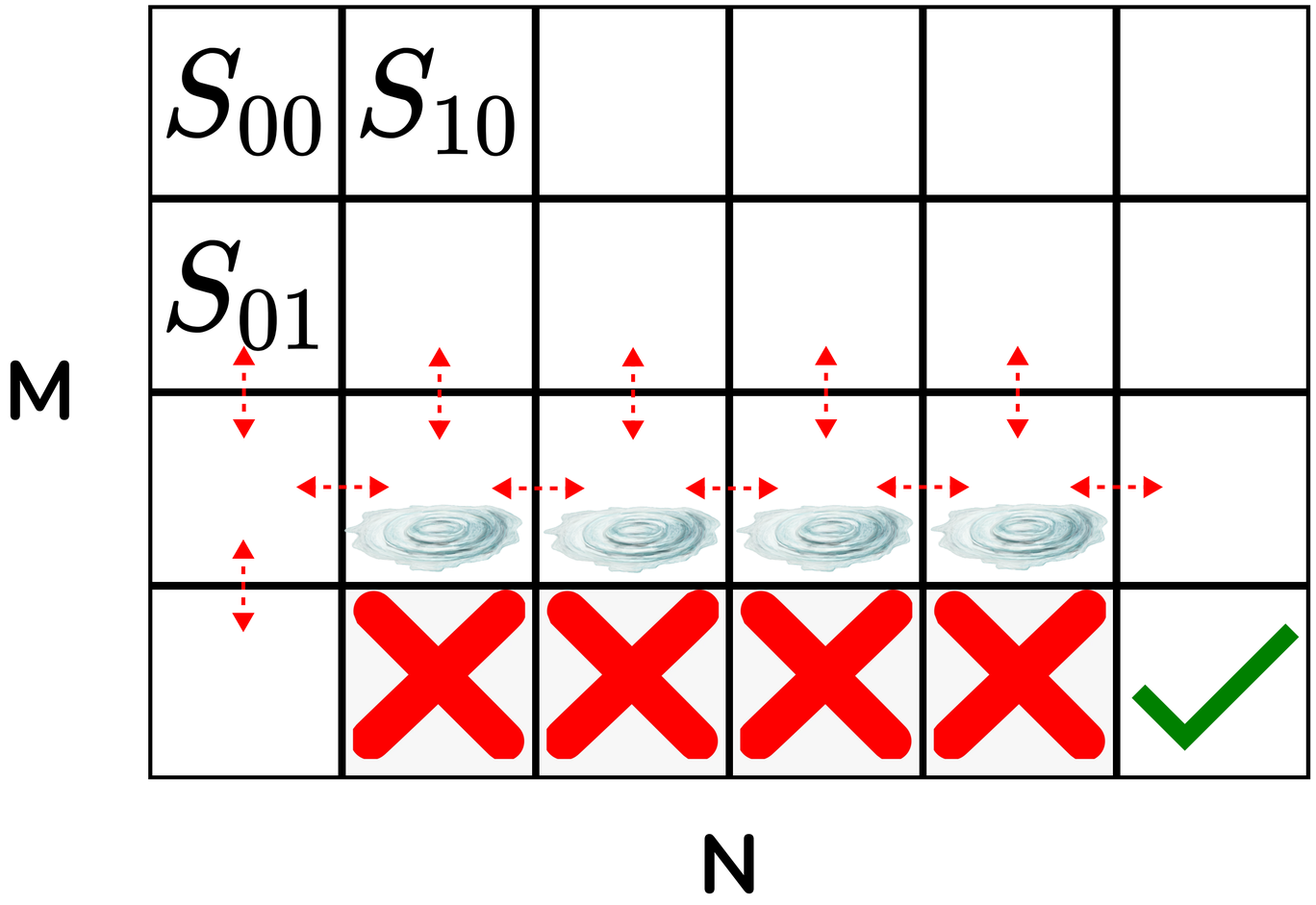
```
[[ -1.11  -1.11  -1.11  -1.11  -1.11  -1.1 ]
 [ -1.11  -1.11  -1.11  -1.11  -1.1   -1.  ]
 [ -1.11  -1.11  -1.11  -1.1   -1.     0.  ]
 [ -1.11 -10.   -10.   -10.   -10.    10.  ]]
```

t = 3

```
[[ -1.111111  -1.11111   -1.1111    -1.111     -1.11      -1.1     ]
 [ -1.11111   -1.1111    -1.111     -1.11      -1.1       -1.      ]
 [ -1.1111    -1.111     -1.11      -1.1       -1.         0.      ]
 [ -1.11111  -10.       -10.       -10.       -10.       10.      ]]
```

t = 99

Task for today



R = -1

Terminating states values:

V = 10

V = -10

Implement “slippery cells”. When agent makes an action leading to “Slippery cell”, 5% chance to apper in the cell below target “slippery” one.

In your experiments use discounting factor = 0.3.