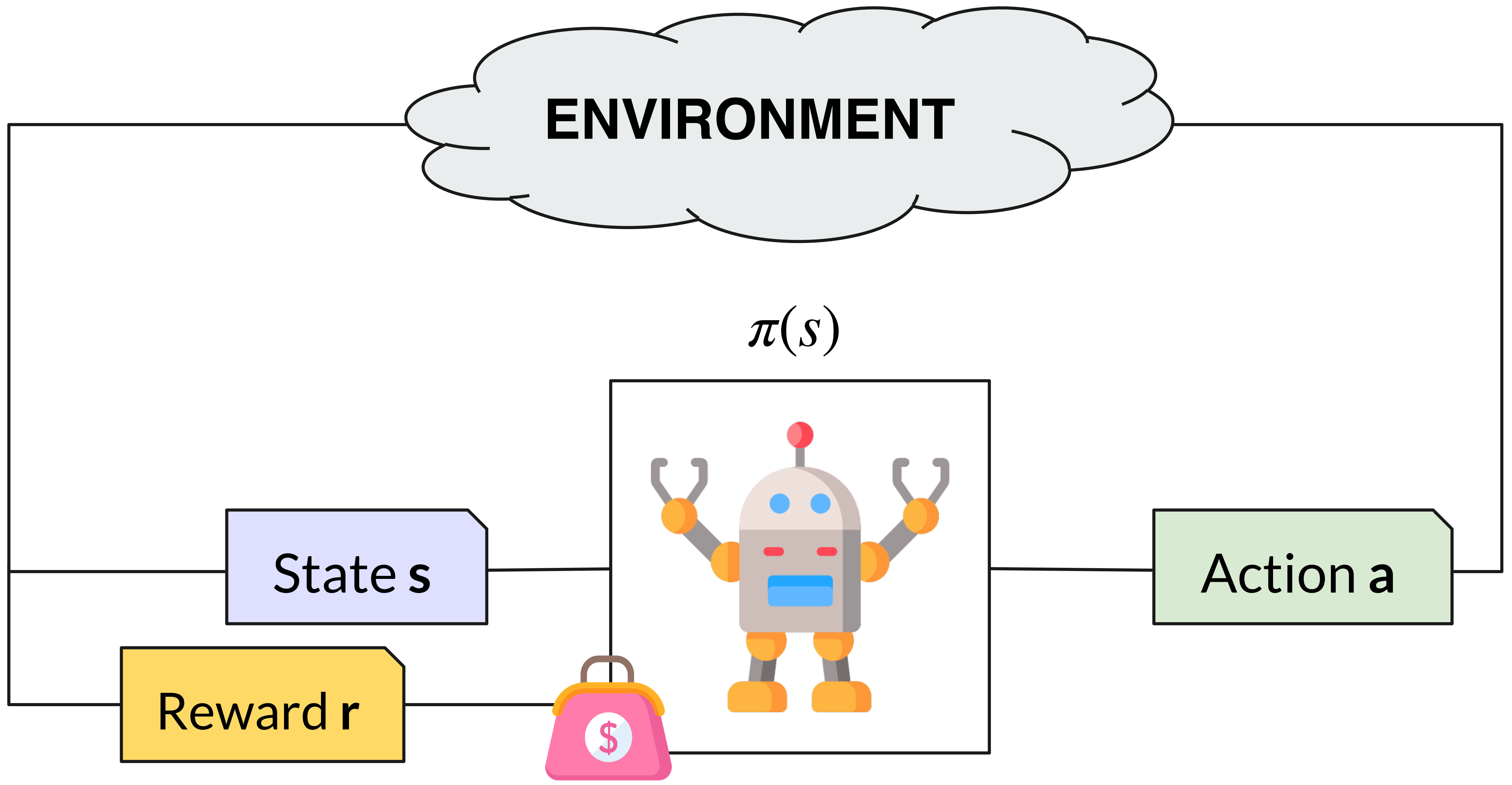
SAPIENZA
UNIVERSITÀ DI ROMA

MARKOV DECISION PROCESSES

COMPUTER NETWORK PERFORMANCE 14-12-2020

Reinforcement Learning

- Branch of automatic learning (just like Supervised Learning or Unsupervised Learning)
- An agent wants to learn an *optimal* policy (how to act) in an environment, with the goal of maximising cumulative reward signals it gets from the environment



Sense
Think
Act

Reinforcement Learning an Example

- A self-driving car (the agent) drives autonomously without going off-road:
  - **States**: raw pixel inputs cameras mounted on the car (radars…)
  - **Actions**: steer right, steer left, do nothing
  - **Rewards**: a function of the distance of the car from the border lines

Reward Maximisation

- Three ways to optimise the sum of future rewards:

Infinite horizon $\sum_{i=0}^{\infty} r_i$

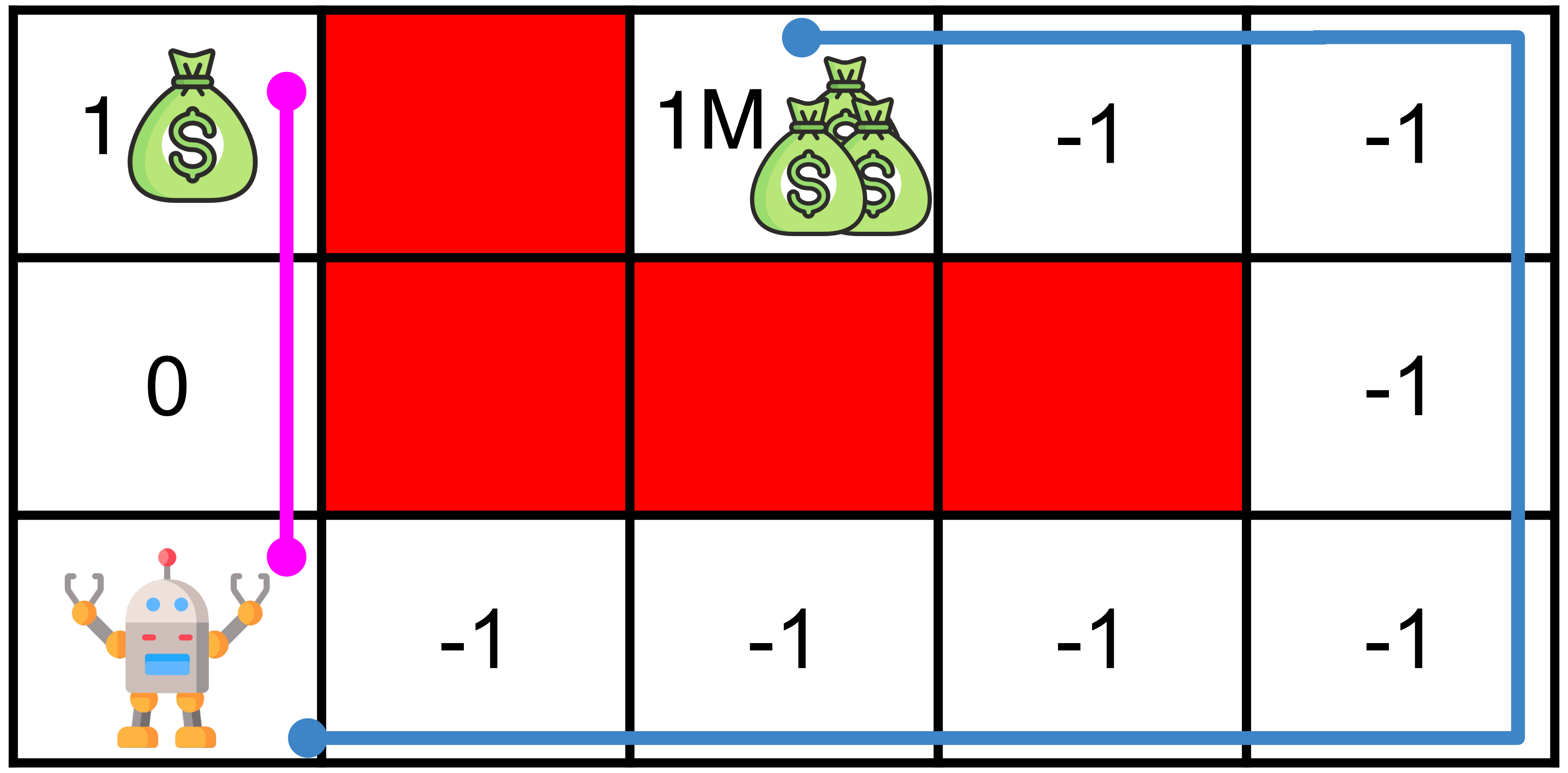
P1: 0, 1, 0, 1 … -> ∞

P2: -1, -1, -1, -1, -1, -1, -1, -1, 1M, -1, 1M … -> ∞

Finite horizon $\sum_{i=0}^{n} r_i$

P1 (n = 3): 0, 1, 0 -> 1$

P2 (n = 3): -1, -1, -1 -> -3$



- **Consider:** if it takes 4 years to get 1$, that dollar is less valuable than if it takes 1 year (**inflation**)!
- **Discounted rewards** $0 < \gamma \leq 1$with parameter γ = 0.95 means that each reward in the future values 5% less than the immediate reward

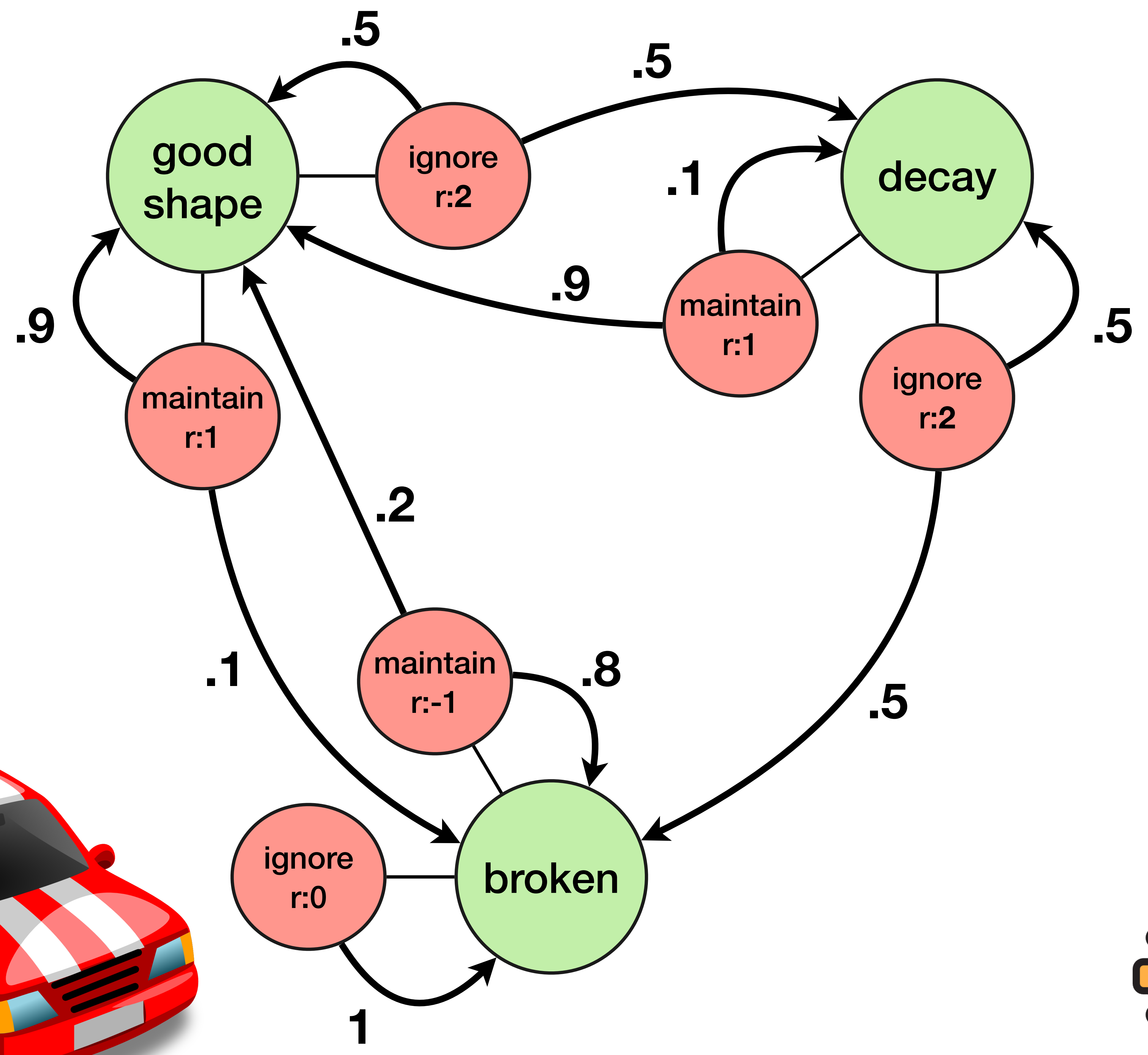
Discounted reward $$\sum_{i=0}^{\infty} \gamma^i \, r_i = r_0 + \gamma r_1 + \gamma^2 r_2 + \gamma^3 r_3 \ldots$$

Markov Decision Processes

- A Markov Decision Process is a 4 tuple **MDP** = ***(S, A, T, R)***
  - S : set of **states**
  - A : set of **actions**
  - $T : \{S \times A \times S\} \to [0,1]$: **transition probability** function s.t. $\sum_{s' \in S} T[s, a, s'] = 1 \; \forall s \in S, a \in A$
  - $R : \{S \times A\} \to \mathbb{R}$: **reward function**
- The **policy** is a function $\pi : S \to A$ specifying what action to take in each state
- The goal is to find π^*, a policy that maximises cumulative discounted reward $\sum_{i=0}^{\infty} \gamma^i \, r_i$

Markov Decision Processes

- A Markov Decision Process is a 4 tuple **MDP** = ***(S, A, T, R)***

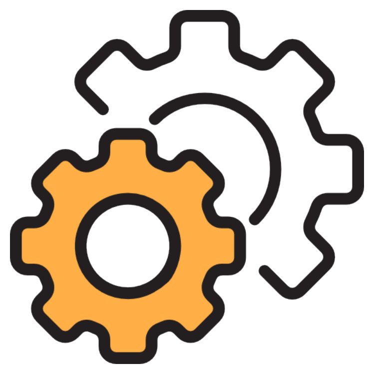


$$S = \{\text{ good shape, decay, broken }\}$$

$$A = \{\text{ maintain, ignore }\}$$

$$R = \begin{array}{cl} \begin{bmatrix} +1 & +2 \\ +1 & +2 \\ -1 & 0 \end{bmatrix} & \begin{array}{l} \text{good shape} \\ \text{decay} \\ \text{broken} \end{array} \end{array}$$

$$T = \begin{array}{ccl} \text{maintain} & \text{ignore} & \\ \begin{bmatrix} 0.9 & 0 & 0.1 \\ 0.9 & 0.1 & 0 \\ 0.2 & 0 & 0.8 \end{bmatrix} & \begin{bmatrix} 0.5 & 0.5 & 0 \\ 0 & 0.5 & 0.5 \\ 0 & 0 & 1 \end{bmatrix} & \begin{array}{l} \text{good shape} \\ \text{decay} \\ \text{broken} \end{array} \end{array}$$

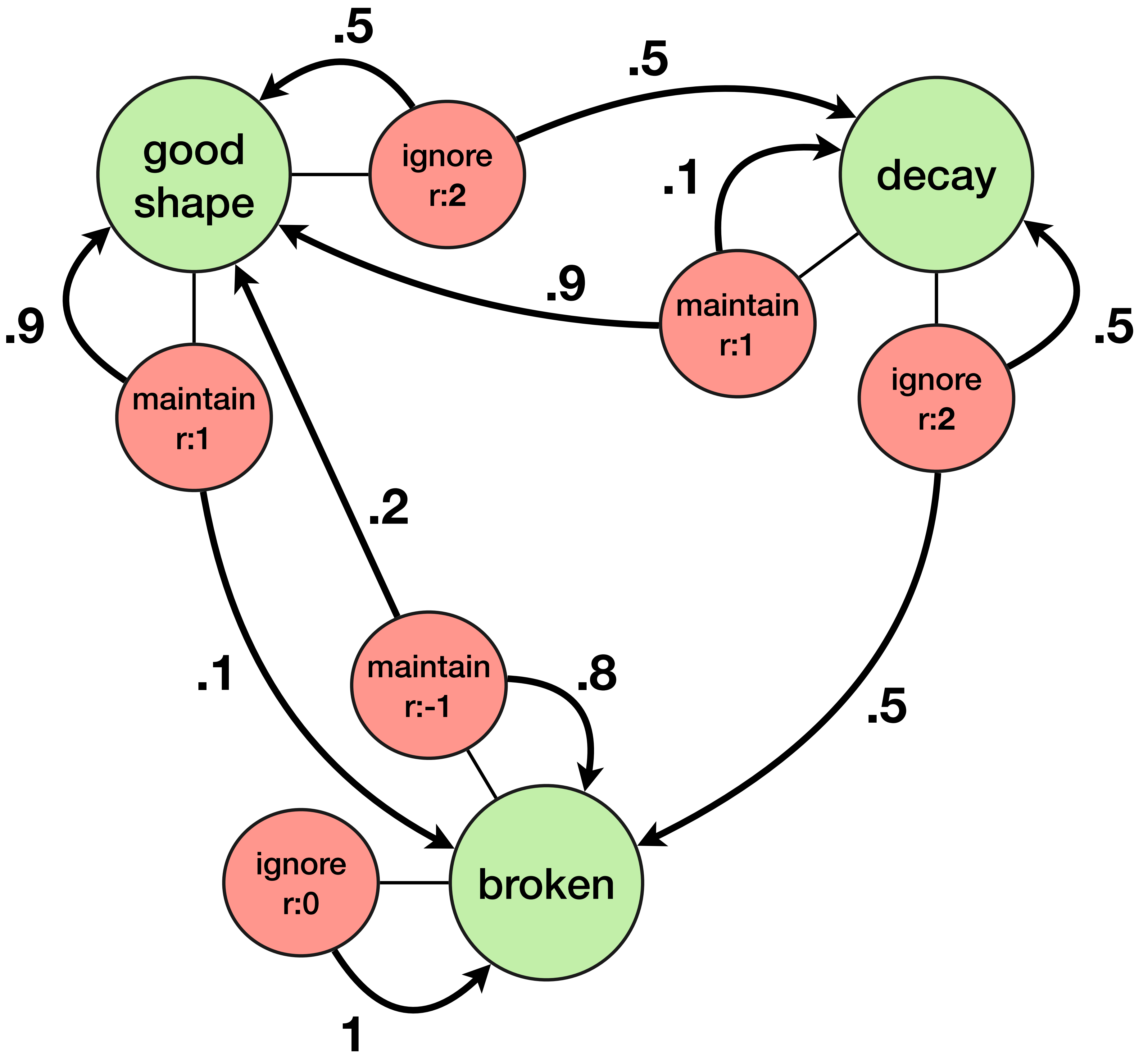


$$\pi = [\text{ ignore, ignore, maintain }]$$

Markov Decision Processes

Process evolution

- At time step $t = 0$, given policy π and initial state s_0
- For $t = 0$ until done:
  - Agent select $a_t \leftarrow \pi[s_t]$
  - Get target state $s'_t \sim T[s_t, a_t, *]$
  - Get reward $r_t \leftarrow R[s_t, a_t]$
  - Agent collects **experience tuple** (s_t, a_t, s'_t, r_t)



$\pi =$ [**ignore, ignore, maintain**] $s_0 =$ **good shape**

$$T = \begin{bmatrix} 0.9 & 0 & 0.1 \\ 0.9 & 0.1 & 0 \\ 0.2 & 0 & 0.8 \end{bmatrix} \begin{bmatrix} 0.5 & 0.5 & 0 \\ 0 & 0.5 & 0.5 \\ 0 & 0 & 1 \end{bmatrix} \quad R = \begin{bmatrix} +1 & +2 \\ +1 & +2 \\ -1 & 0 \end{bmatrix}$$

good shape
decay
broken

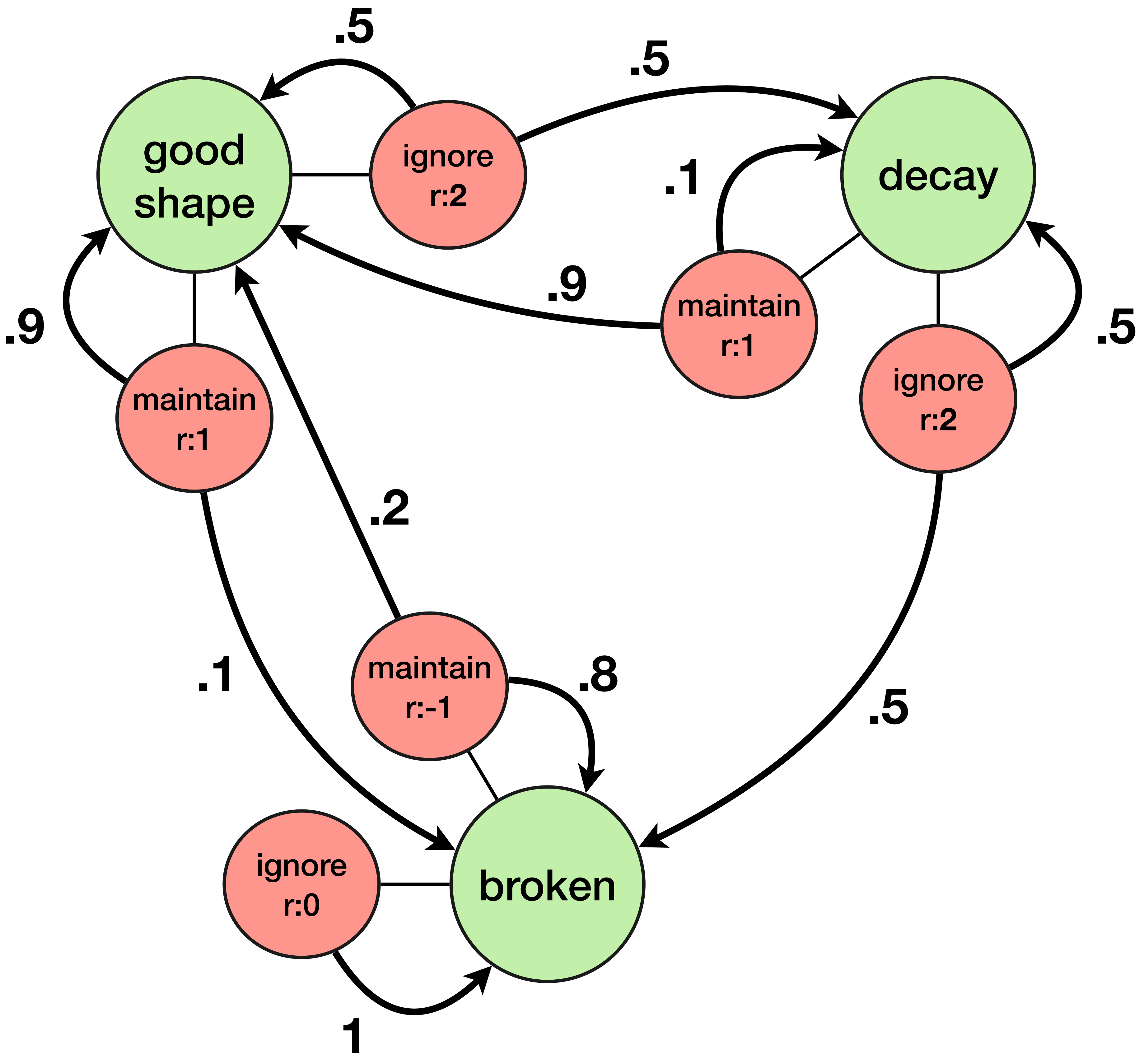
t = 0 : (good shape, ignore, good shape, 2)
t = 1 : (good shape, ignore, decay, 2)
t = 2 : (decay, ignore, broken, 2)
t = 3 : (broken, maintain, broken, -1) ...

R = 2 + 2 +2 -1 ...

Markov Decision Processes

Process evolution

- At time step $t = 0$, given policy π and initial state s_0
- For $t = 0$ until done:
  - Agent select $a_t \leftarrow \pi[s_t]$
  - Get target state $s'_t \sim T[s_t, a_t, *]$
  - Get reward $r_t \leftarrow R[s_t, a_t]$
  - Agent collects **experience tuple** (s_t, a_t, s'_t, r_t)

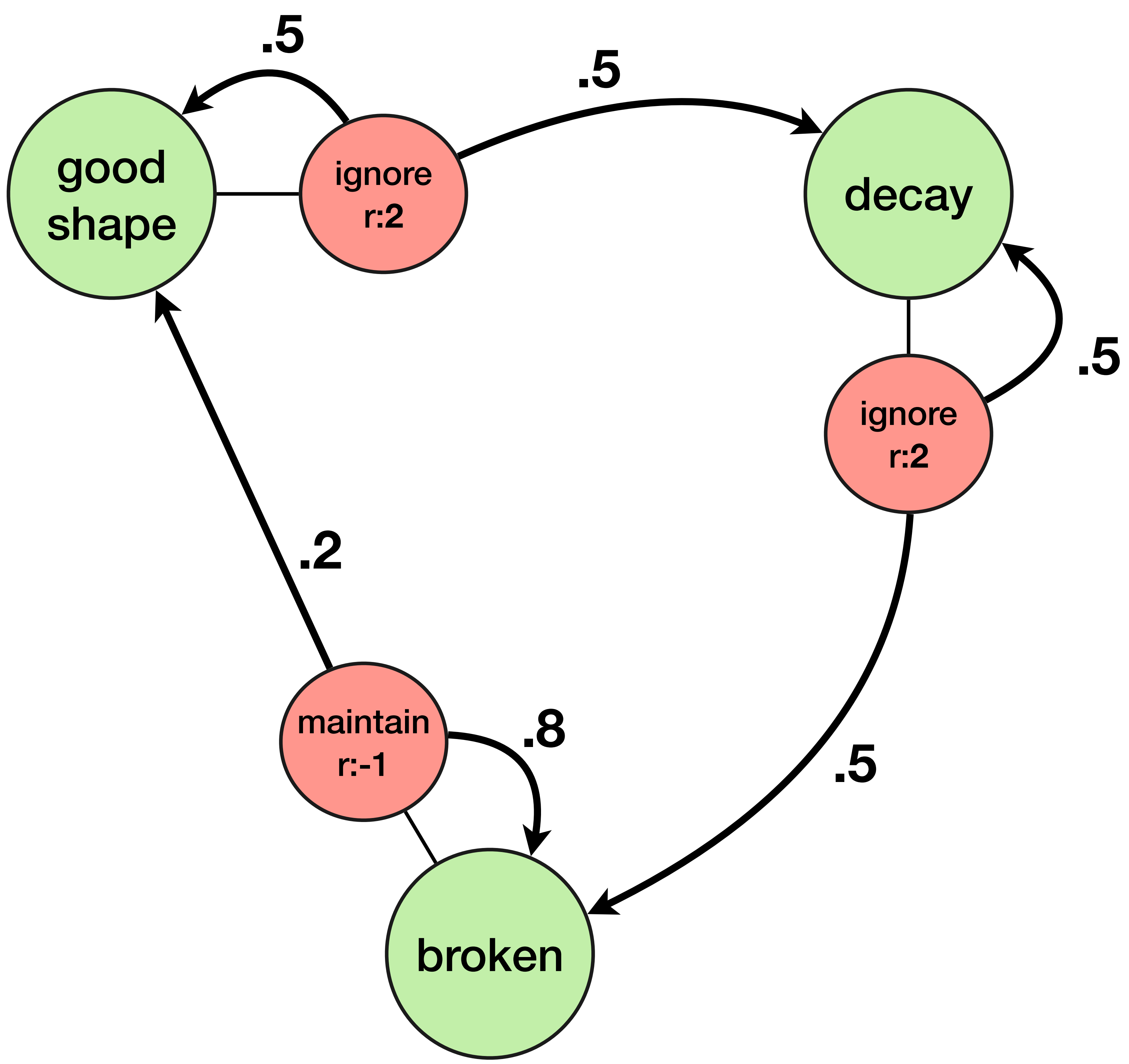


$\pi =$ [**ignore, ignore, maintain**] $s_0 =$ **good shape**

$$T = \begin{bmatrix} 0.9 & 0 & 0.1 \\ 0.9 & 0.1 & 0 \\ 0.2 & 0 & 0.8 \end{bmatrix} \begin{bmatrix} 0.5 & 0.5 & 0 \\ 0 & 0.5 & 0.5 \\ 0 & 0 & 1 \end{bmatrix} \quad R = \begin{array}{cl} \begin{bmatrix} +1 & +2 \\ +1 & +2 \\ -1 & 0 \end{bmatrix} & \begin{array}{l} \textbf{good shape} \\ \textbf{decay} \\ \textbf{broken} \end{array} \end{array}$$

$$T_\pi = \begin{bmatrix} 0.5 & 0.5 & 0 \\ 0 & 0.5 & 0.5 \\ 0.2 & 0 & 0.8 \end{bmatrix}$$

Markov Decision Processes



Process evolution

- At time step $t = 0$, given policy π and initial state s_0
- For $t = 0$ until done:
  - Agent select $a_t \leftarrow \pi[s_t]$
  - Get target state $s_t' \sim T[s_t, a_t, *\,]$
  - Get reward $r_t \leftarrow R[s_t, a_t]$
  - Agent collects **experience tuple** (s_t, a_t, s_t', r_t)

$\pi =$ [**ignore, ignore, maintain**] $s_0 =$ **good shape**

$$T = \begin{bmatrix} 0.9 & 0 & 0.1 \\ 0.9 & 0.1 & 0 \\ 0.2 & 0 & 0.8 \end{bmatrix} \begin{bmatrix} 0.5 & 0.5 & 0 \\ 0 & 0.5 & 0.5 \\ 0 & 0 & 1 \end{bmatrix} \quad R = \begin{array}{c} \begin{bmatrix} +1 & +2 \\ +1 & +2 \\ -1 & 0 \end{bmatrix} \end{array} \begin{array}{l} \textbf{good shape} \\ \textbf{decay} \\ \textbf{broken} \end{array}$$

$$T_\pi = \begin{bmatrix} 0.5 & 0.5 & 0 \\ 0 & 0.5 & 0.5 \\ 0.2 & 0 & 0.8 \end{bmatrix}$$

Markov Decision Processes

- For every MDP there exists an optimal policy (***proven***)
- The optimal policy is the one that **maximises the expected sum of rewards** (cumulative reward)

$$\pi^* = \arg\max_{\pi} \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r_t \middle| \pi\right]$$

How good is a state?

The **value function** at state s is the expected cumulative reward from following the policy from state s

It obeys the Bellman equation, relating the value function to itself as:

$$V^{\pi}(s) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r_t \middle| s_0 = s, \pi\right] = \underbrace{R[s, \pi[s]]}_{\text{immediate reward}} + \underbrace{\sum_{s' \in S} T[s, \pi[s], s'] \cdot \gamma \cdot V^{\pi}(s')}_{\text{expected cumulative discounted reward}}$$

Optimal value function

$$V^*(s) = \max_{a \in A} R[s, a] + \sum_{s' \in S} T[s, a, s'] \cdot \gamma \cdot V^*(s')$$

Value Iteration Algorithm

- **set** $V[s] = 0 \; \forall s \in S$
- **while** true:
  - **for** each $s \in S$:
    - $V[s] = \max_{a \in A} R[s,a] + \sum_{s' \in S} T[s,a,s'] \cdot \gamma \cdot V[s']$
  - **exit** if convergence (small change from previous and actual $V[s] \; \forall s$)

Optimal policy! $\pi^*[s] = \arg\max_{a \in A} R[s,a] + \sum_{s' \in S} T[s,a,s'] \cdot \gamma \cdot V[s']$

Value Iteration Algorithm

$$T = \begin{bmatrix} 0.9 & 0 & 0.1 \\ 0.9 & 0.1 & 0 \\ 0.2 & 0 & 0.8 \end{bmatrix} \begin{bmatrix} 0.5 & 0.5 & 0 \\ 0 & 0.5 & 0.5 \\ 0 & 0 & 1 \end{bmatrix} \quad R = \begin{bmatrix} +1 & +2 \\ +1 & +2 \\ -1 & 0 \end{bmatrix} \begin{array}{l} \text{good shape} \\ \text{decay} \\ \text{broken} \end{array}$$

$$\gamma = 0.9$$

- set $V[s] = 0 \ \forall s \in S$
- while true:
  - for each $s \in S$:
    - $V[s] = \max_{a \in A} R[s,a] + \sum_{s' \in S} T[s,a,s'] \cdot \gamma \cdot V[s']$
  - exit if convergence (small change from previous and actual $V[s]$ $\forall s$)

Optimal policy! $\pi^*[s] = \arg\max_{a \in A} R[s,a] + \sum_{s' \in S} T[s,a,s'] \cdot \gamma \cdot V[s']$

While iter 1

- $V = [0,0,0]$

  state 0 – good shape : $V[0] = \max\{\ a_0 : 1 + (0.9 \cdot \gamma \cdot 0) + (0 \cdot \gamma \cdot 0) + (0.1 \cdot \gamma \cdot 0);$

  $a_1 : 2 + (0.5 \cdot \gamma \cdot 0) + (0.5 \cdot \gamma \cdot 0) + (0 \cdot \gamma \cdot 0)\} = \max\{1,2\}$

- $V = [2,0,0]$

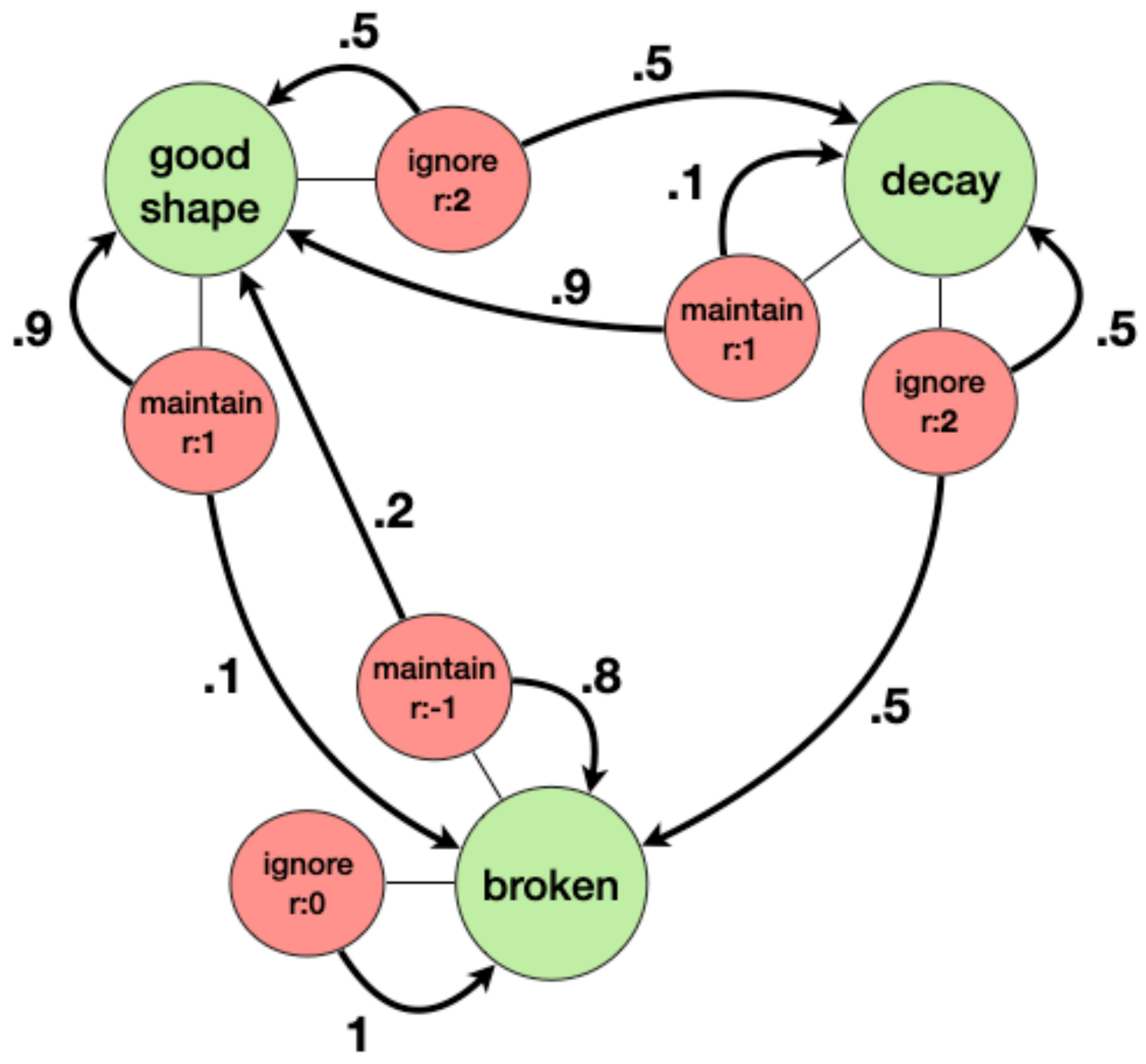
  state 1 –decay : $V[1] = \max\{\ a_0 : 1 + (0.9 \cdot \gamma \cdot 2) + (0.1 \cdot \gamma \cdot 0) + (0 \cdot \gamma \cdot 0);$

  $a_1 : 2 + (0 \cdot \gamma \cdot 2) + (0.5 \cdot \gamma \cdot 0) + (0.5 \cdot \gamma \cdot 0)\} = \max\{2.62,\ 2\}$

- $V = [2, 2.62, 0]$

  state 2 –broken : $V[2] = \max\{\ a_0 : -1 + (0.2 \cdot \gamma \cdot 2) + (0 \cdot \gamma \cdot 2.68) + (0.8 \cdot \gamma \cdot 0);$

  $a_1 : 0 + (0 \cdot \gamma \cdot 2) + (0 \cdot \gamma \cdot 2.68) + (1 \cdot \gamma \cdot 0)\} = \max\{-0.64,\ 0\}$



Value Iteration Algorithm

$$T = \begin{bmatrix} 0.9 & 0 & 0.1 \\ 0.9 & 0.1 & 0 \\ 0.2 & 0 & 0.8 \end{bmatrix} \begin{bmatrix} 0.5 & 0.5 & 0 \\ 0 & 0.5 & 0.5 \\ 0 & 0 & 1 \end{bmatrix} \quad R = \begin{bmatrix} +1 & +2 \\ +1 & +2 \\ -1 & 0 \end{bmatrix} \begin{matrix} \text{good shape} \\ \text{decay} \\ \text{broken} \end{matrix}$$

$$\gamma = 0.9$$

- set $V[s] = 0 \ \forall s \in S$
- while true:
  - for each $s \in S$:
    - $\ast \ V[s] = \max_{a \in A} R[s,a] + \sum_{s' \in S} T[s,a,s'] \cdot \gamma \cdot V[s']$
  - exit if convergence (small change from previous and actual $V[s] \ \forall s$)

Optimal policy! $\pi^*[s] = \arg\max_{a \in A} R[s,a] + \sum_{s' \in S} T[s,a,s'] \cdot \gamma \cdot V[s']$

While iter 2

- $V = [2, 2.62, 0]$

  state 0 — good shape : $V[0] = \max\{\ a_0 : 1 + (0.9 \cdot \gamma \cdot 2) + (0 \cdot \gamma \cdot 2.68) + (0.1 \cdot \gamma \cdot 0);$

  $a_1 : 2 + (0.5 \cdot \gamma \cdot 2) + (0.5 \cdot \gamma \cdot 2.62) + (0 \cdot \gamma \cdot 0)\} = \max\{2.62, 4.08\}$

- $V = [4.08, 2.62, 0]$

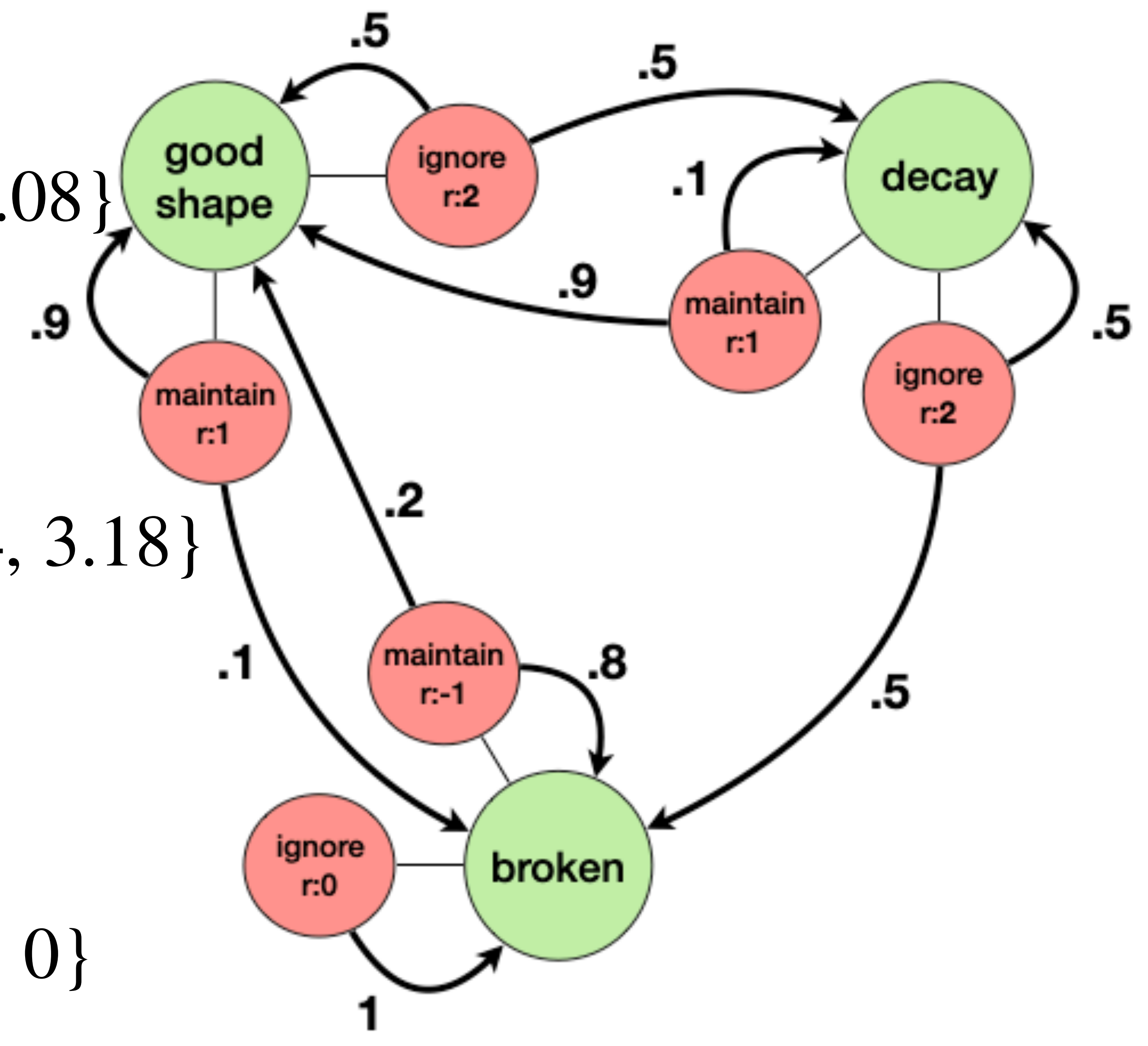
  state 1 —decay : $V[1] = \max\{\ a_0 : 1 + (0.9 \cdot \gamma \cdot 4.08) + (0.1 \cdot \gamma \cdot 2.62) + (0 \cdot \gamma \cdot 0);$

  $a_1 : 2 + (0 \cdot \gamma \cdot 4.08) + (0.5 \cdot \gamma \cdot 2.62) + (0.5 \cdot \gamma \cdot 0)\} = \max\{4.54,\ 3.18\}$

- $V = [4.08, 4.54, 0]$

  state 2 —broken : $V[2] = \max\{\ a_0 : -1 + (0.2 \cdot \gamma \cdot 4.08) + (0 \cdot \gamma \cdot 4.54) + (0.8 \cdot \gamma \cdot 0);$

  $a_1 : 0 + (0 \cdot \gamma \cdot 4.08) + (0 \cdot \gamma \cdot 4.54) + (1 \cdot \gamma \cdot 0)\} = \max\{-0.27,\ 0\}$



Value Iteration Algorithm

$$T = \begin{bmatrix} 0.9 & 0 & 0.1 \\ 0.9 & 0.1 & 0 \\ 0.2 & 0 & 0.8 \end{bmatrix} \begin{bmatrix} 0.5 & 0.5 & 0 \\ 0 & 0.5 & 0.5 \\ 0 & 0 & 1 \end{bmatrix} \quad R = \begin{bmatrix} +1 & +2 \\ +1 & +2 \\ -1 & 0 \end{bmatrix}$$

good shape
decay
broken

$\gamma = 0.9$

- set $V[s] = 0 \ \forall s \in S$
- while true:
  - for each $s \in S$:
    - $V[s] = \max_{a \in A} R[s,a] + \sum_{s' \in S} T[s,a,s'] \cdot \gamma \cdot V[s']$
  - exit if convergence (small change from previous and actual $V[s] \ \forall s$)

Optimal policy! $\pi^*[s] = \arg\max_{a \in A} R[s,a] + \sum_{s' \in S} T[s,a,s'] \cdot \gamma \cdot V[s']$

Converged at iter. 50! Now it is time to find the optimal policy

- $V = [16.69, 15.95, 7.15]$

state 0 — good shape : $\pi^*[0] = \arg\max\{\ a_0 : 1 + (0.9 \cdot \gamma \cdot 16.69) + (0 \cdot \gamma \cdot 15.95) + (0.1 \cdot \gamma \cdot 7.15);$
$a_1 : 2 + (0.5 \cdot \gamma \cdot 16.69) + (0.5 \cdot \gamma \cdot 15.95) + (0 \cdot \gamma \cdot 7.15)\} = \arg\max\{15.16, 16.68\}$ = 1 **ignore**

state 1 — decay: $\pi^*[1] = \arg\max\{\ a_0 : 1 + (0.9 \cdot \gamma \cdot 16.69) + (0.1 \cdot \gamma \cdot 15.95) + (0 \cdot \gamma \cdot 7.15);$
$a_1 : 2 + (0 \cdot \gamma \cdot 16.69) + (0.5 \cdot \gamma \cdot 15.95) + (0.5 \cdot \gamma \cdot 7.15)\} = \arg\max\{15.95, 12.4\}$ = 0 **maintain**

state 2 — broken: $\pi^*[2] = \arg\max\{\ a_0 : -1 + (0.2 \cdot \gamma \cdot 16.69) + (0 \cdot \gamma \cdot 15.95) + (0.8 \cdot \gamma \cdot 7.15);$
$a_1 : 0 + (0 \cdot \gamma \cdot 16.69) + (0 \cdot \gamma \cdot 15.95) + (1 \cdot \gamma \cdot 7.15)\} = \arg\max\{7.15, 6.44\}$ = 0 **maintain**

$\pi^* =$ [ignore, maintain, maintain]

Policy Iteration Algorithm

- Value iteration is slow $\mathcal{O}(S^2A)$

Initialisation

- **set** $V[s]$ random value and $\pi[s]$ random action $\forall s \in S$

Policy evaluation

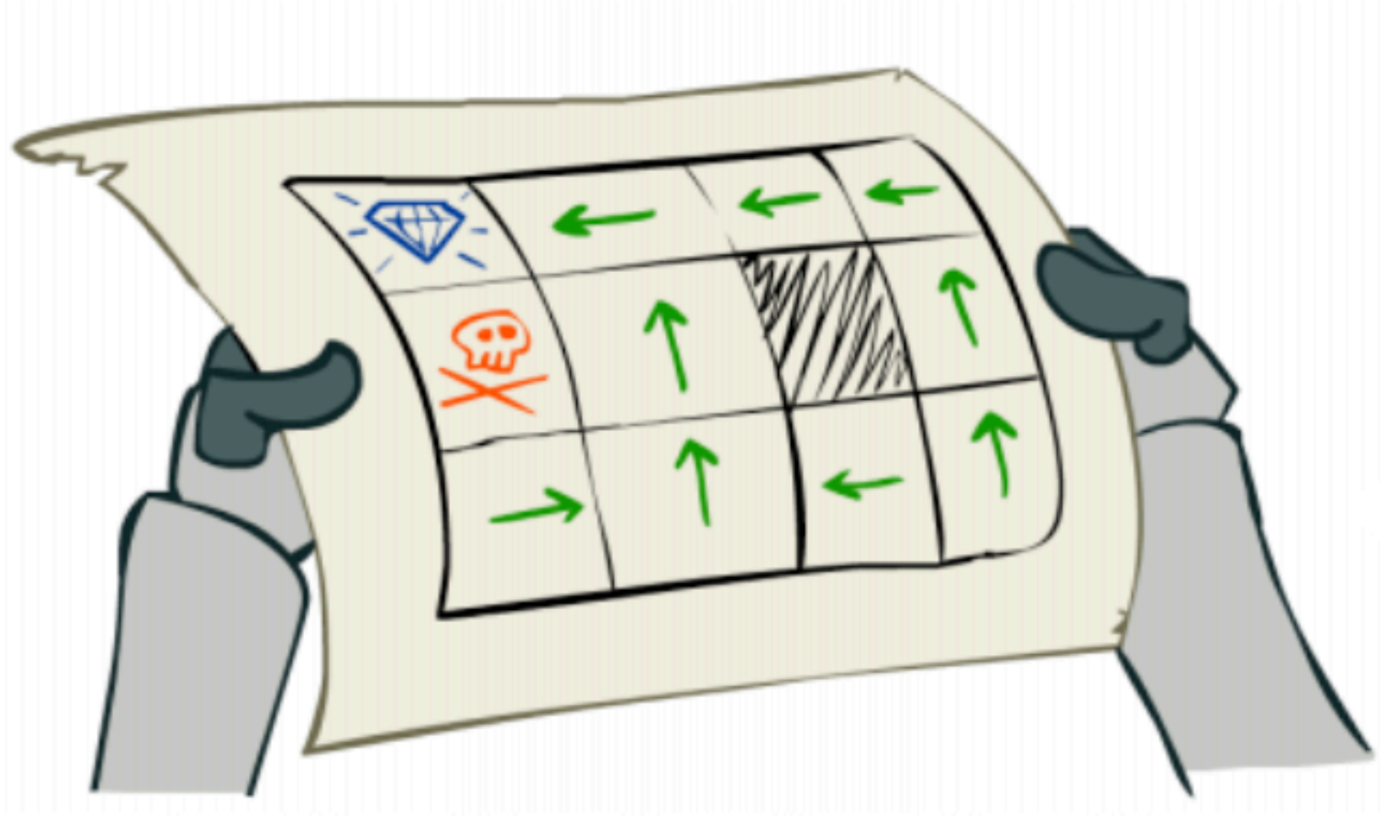
- **while** true:
  - **for** each $s \in S$:
    - $V[s] = R[s, \pi[s]] + \sum_{s' \in S} T[s, \pi[s], s'] \cdot \gamma \cdot V[s']$
  - **exit** if convergence (small change from previous and actual $V[s]$ $\forall s$)

Policy update

- **for** each $s \in S$:
  - $\pi^*[s] = \arg\max_{a \in A} R[s, a] + \sum_{s' \in S} T[s, a, s'] \cdot \gamma \cdot V[s']$
- **if** the policy did not change after the last loop, return it; else go back to **policy evaluation**

Two Reinforcement Learning Ways

- What we have seen so far was **model *based* reinforcement learning**, use Value Iteration and Policy Iteration algorithms for finding an optimal policy
- What if you **don't have full knowledge of the environment** and you want to discover the optimal policy anyways. You **don't know T and R**, at least not completely
- Another reinforcement learning class of problems is called **model *free* reinforcement learning**
- A robot discovering an *unknown* environment and learning what to do, from the *unknown* feedback it gets from it
- That's when **Q-learning** comes in



Q-Learning

- Remember value function:

Optimal value function

$$V^*(s) = \max_{a \in A} R[s, a] + \sum_{s' \in S} T[s, a, s'] \cdot \gamma \cdot V^*(s')$$

- Let Q be a function mapping a **state** and an **action** to the quality of that pair, it satisfies the Bellman equation as:

$$Q^*(s, a) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r_t \,\middle|\, s_0 = s, a_0 = a, \pi\right] = \mathbb{E}\left[r + \gamma \max_{a' \in A} Q^*(s', a') \,\middle|\, s_0 = s, a_0 = a, \pi\right]$$

Optimal policy

$$\pi^*(s) = \arg\max_{a \in A} Q^*(s, a)$$

Q-Learning Algorithm

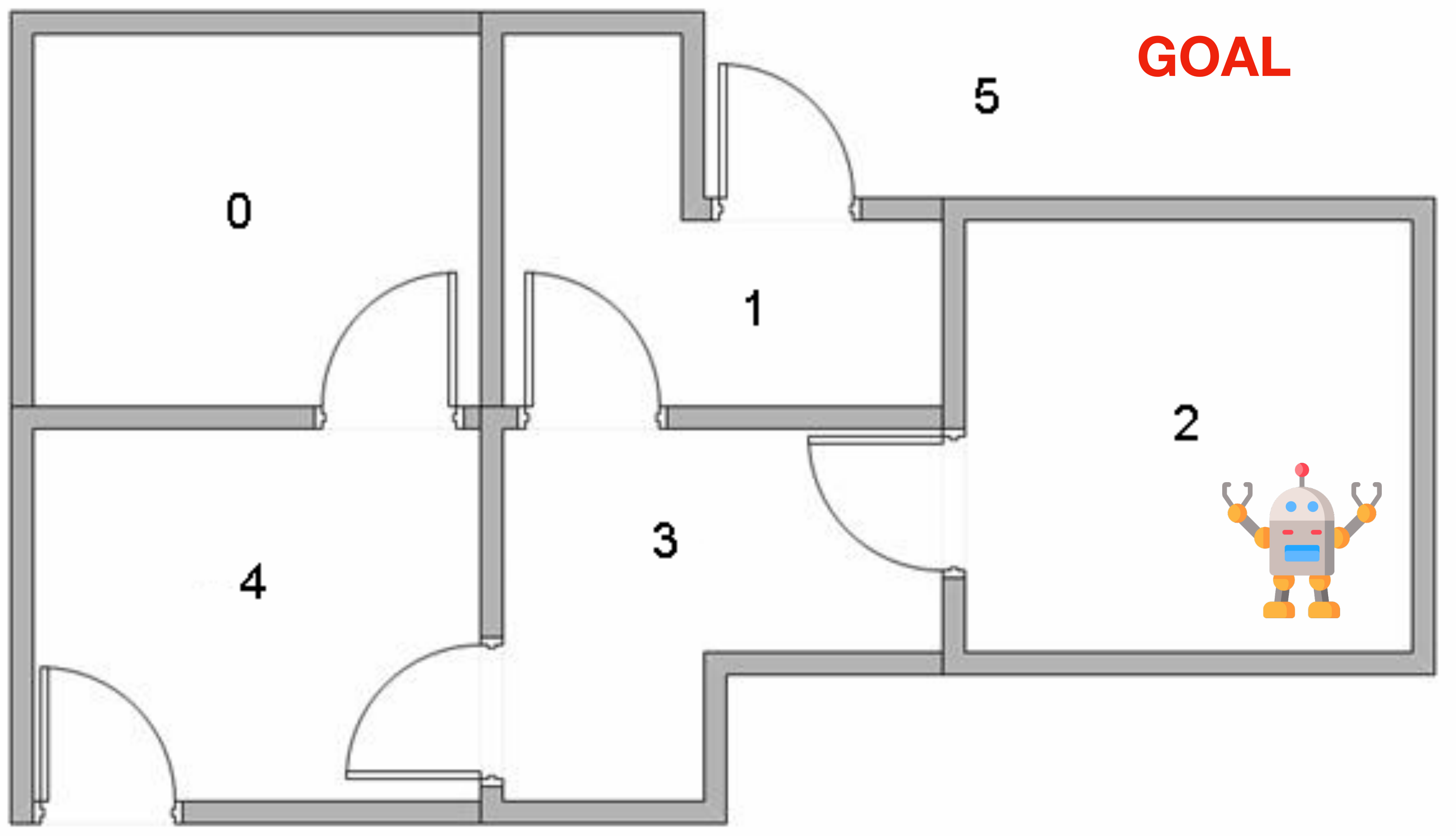
- **set** $Q[s, a] = 0 \ \forall s \in S, a \in A$
- **while** true:
  - from state s **do** until done state
    - ✳ choose action *initially random* with decaying probability, then $a = \arg\max_{a' \in A} Q(s, a')$
    - ✳ take action and observe s', r, **set** $s = s'$
    - ✳ $Q[s, a] = (1 - \alpha)Q[s, a] + \alpha(r + \gamma \max_{a' \in A} Q[s', a'])$
  - **exit** when converged

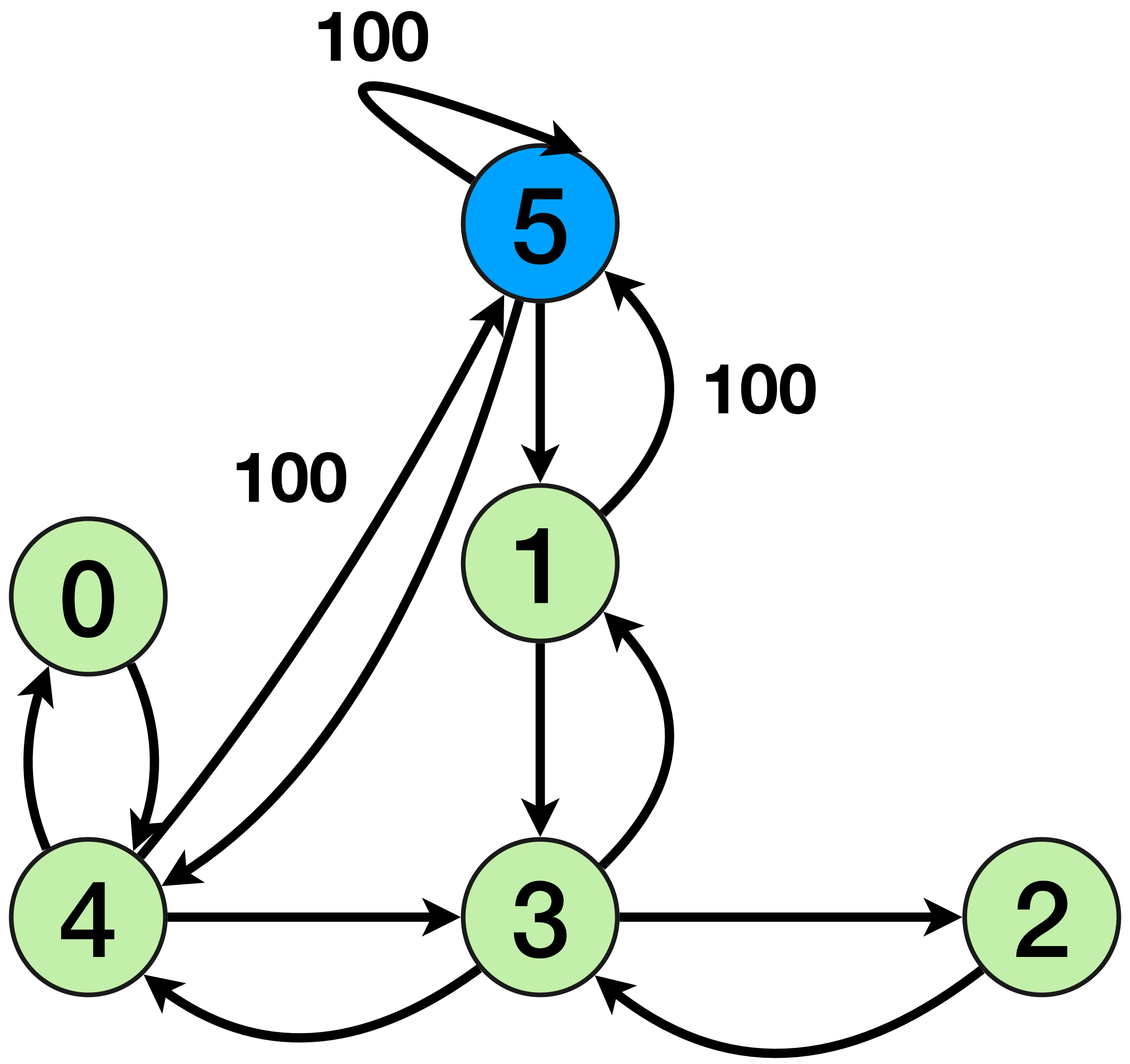
Optimal policy $\pi^*(s) = \arg\max_{a \in A} Q(s, a)$

- success depends on **exploration** and **experience replay**

Q-Learning Example

- There are 5 rooms (0 to 4) in a building connected by doors. Doors 1 and 4 lead into the building from the outside. Let us teach to an agent the **best policy to go out**.





Rewards map

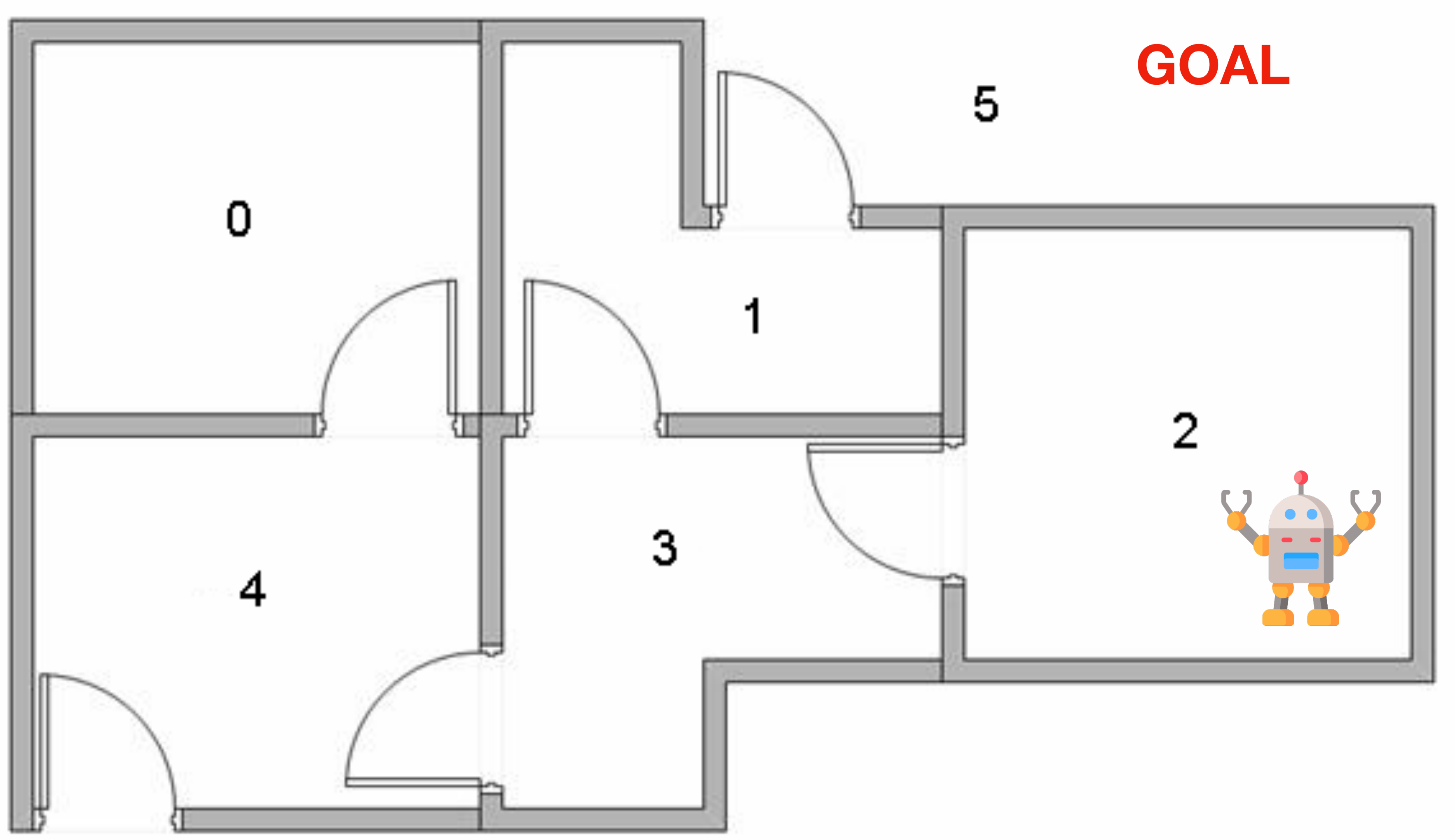
$$R = \begin{array}{c} \text{State} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \text{Action} \\ \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} -1 & -1 & -1 & -1 & 0 & -1 \\ -1 & -1 & -1 & 0 & -1 & 100 \\ -1 & -1 & -1 & 0 & -1 & -1 \\ -1 & 0 & 0 & -1 & 0 & -1 \\ 0 & -1 & -1 & 0 & -1 & 100 \\ -1 & 0 & -1 & -1 & 0 & 100 \end{bmatrix} \end{array}$$

- R is unknown to the agent, it just perceives those feedbacks online. -1 represent null rewards.

http://mnemstudio.org/path-finding-q-learning-tutorial.htm

Q-Learning Example

- Exploit a Q table as representing the memory of what the agent has learned through experience. The **rows** of matrix Q represent **the current state** of the agent, and the **columns** represent the **possible actions** leading to the next state



$$R = \begin{array}{c} \text{State} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \text{Action} \\ \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} -1 & -1 & -1 & -1 & 0 & -1 \\ -1 & -1 & -1 & 0 & -1 & 100 \\ -1 & -1 & -1 & 0 & -1 & -1 \\ -1 & 0 & 0 & -1 & 0 & -1 \\ 0 & -1 & -1 & 0 & -1 & 100 \\ -1 & 0 & -1 & -1 & 0 & 100 \end{bmatrix} \end{array}$$

$$Q = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

- We'll start by setting the value of the learning parameter $\gamma = 0.8, \alpha = 1$, and the initial state as 1.

Q-Learning Example

While iter 1 (epoch)

state room 1

- choose action randomly from **go to room {5, 3} -> 5**
- received reward 100
- $s = 5$
- $Q[1{,}5] = R[1{,}5] + 0.8 \cdot \max\{Q[5{,}1],\ Q[5{,}4],\ Q[5{,}5]\} = 100 + 0.8 \cdot 0 = 100$

While iter 2 (epoch)

state room 3 (chosen randomly)

- choose action randomly from **go to room {1, 2, 4} -> 1**
- received reward 100
- $s = 1$
- $Q[3{,}1] = R[3{,}1] + 0.8 \cdot \max\{Q[1{,}3],\ Q[1{,}5]\} = 0 + 0.8 \cdot 100 = 80$

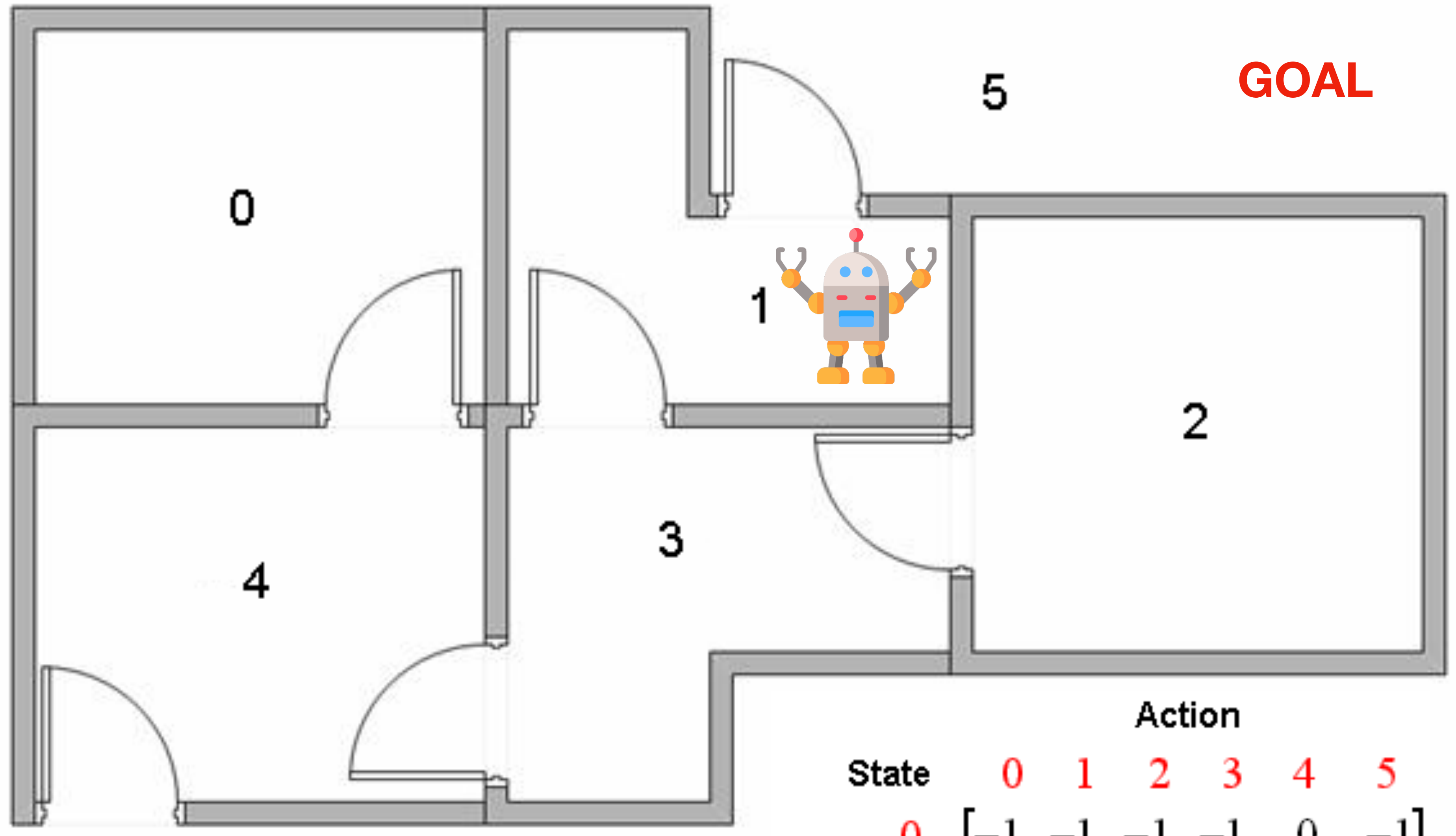
State 5 is a "done state" thus a new epoch should start

$$Q = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 100 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 80 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

set $Q[s,a] = 0\ \forall s \in S, a \in A$
while true:

- from state s **do** until done state
  - choose action *initially random* with decaying probability, then $a = \arg\max_{a' \in A} Q(s,a')$
  - take action and observe s', r, **set** $s = s'$
  - $Q[s,a] = (1-\alpha)Q[s,a] + \alpha(r + \gamma \max_{a' \in A} Q[s',a'])$
- **exit** when performance of target metric converged

Optimal policy $\pi^*(s) = \arg\max_{a \in A} Q(s,a)$



$$R = \begin{array}{c} \text{State} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \text{Action} \\ \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} -1 & -1 & -1 & -1 & 0 & -1 \\ -1 & -1 & -1 & 0 & -1 & 100 \\ -1 & -1 & -1 & 0 & -1 & -1 \\ -1 & 0 & 0 & -1 & 0 & -1 \\ 0 & -1 & -1 & 0 & -1 & 100 \\ -1 & 0 & -1 & -1 & 0 & 100 \end{bmatrix} \end{array}$$

Q-Learning Example

While iter 2 (epoch)

state room 1

- choose action randomly from **go to room {5, 3} -> 3**
- received reward 0
- $s = 3$
- $Q[1,3] = R[1,3] + 0.8 \cdot \max\{Q[3,1], Q[3,2], Q[3,4]\} = 0 + 0.8 \cdot 80 = 64$

state room 3

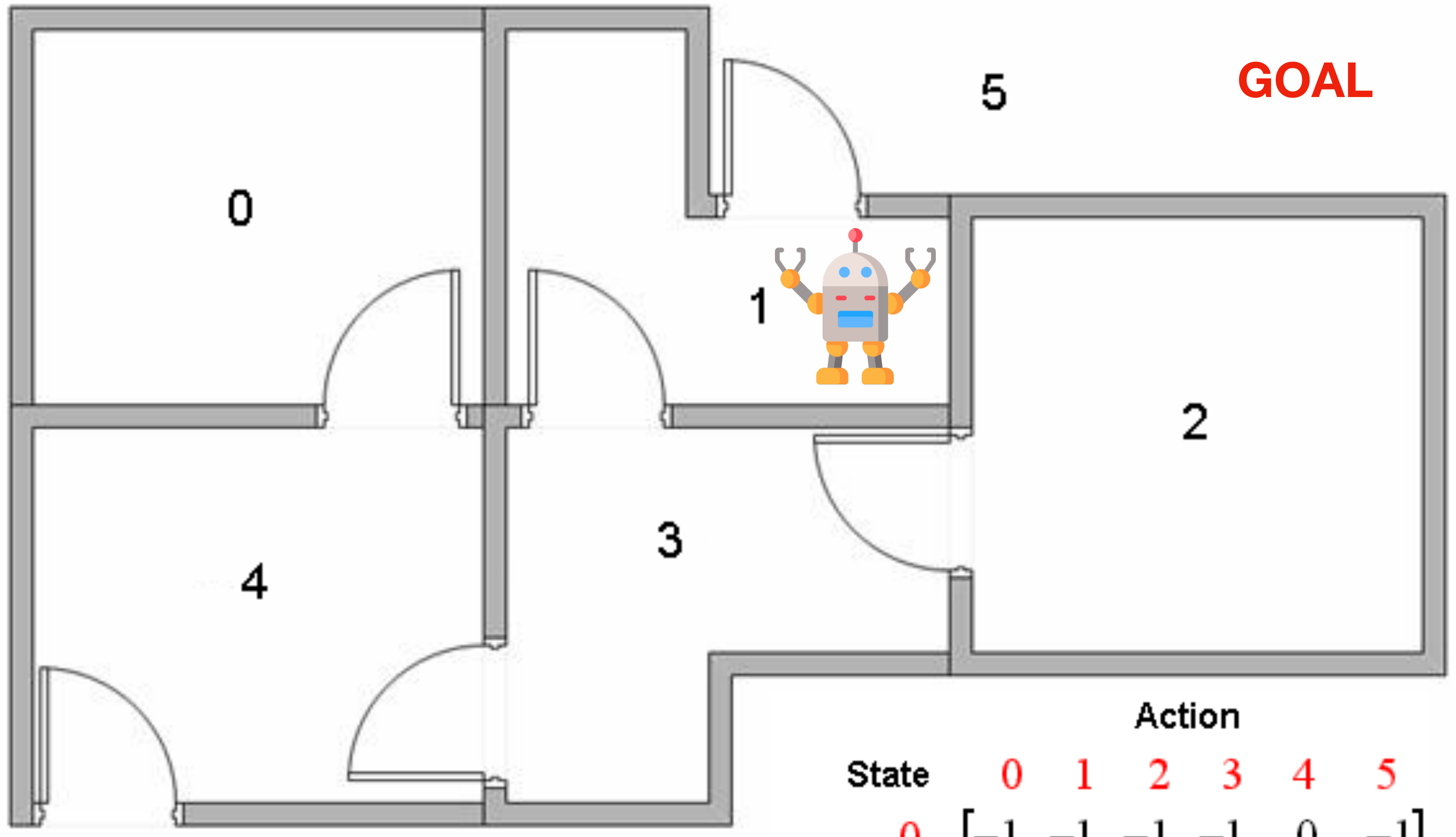
- choose action randomly from **go to room {1, 2, 4} -> 2**
- received reward 0
- $s = 2$
- $Q[3,2] = R[3,2] + 0.8 \cdot \max\{Q[2,3]\} = 0 + 0.8 \cdot 0 = 0$

$$Q = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 64 & 0 & 100 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 80 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

set $Q[s,a] = 0 \; \forall s \in S, a \in A$
while true:

- from state s **do** until done state
  - choose action *initially random* with decaying probability, then $a = \arg\max_{a' \in A} Q(s, a')$
  - take action and observe s', r, **set** $s = s'$
  - $Q[s,a] = (1 - \alpha) Q[s,a] + \alpha (r + \gamma \max_{a' \in A} Q[s', a'])$
- **exit** when performance of target metric converged

Optimal policy $\pi^*(s) = \arg\max_{a \in A} Q(s, a)$



Action

$$R = \begin{array}{c} \text{State} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} -1 & -1 & -1 & -1 & 0 & -1 \\ -1 & -1 & -1 & 0 & -1 & 100 \\ -1 & -1 & -1 & 0 & -1 & -1 \\ -1 & 0 & 0 & -1 & 0 & -1 \\ 0 & -1 & -1 & 0 & -1 & 100 \\ -1 & 0 & -1 & -1 & 0 & 100 \end{bmatrix} \end{array}$$

Q-Learning Example

- Once Q converged, normalise it for convenience and find the optimal policy

$$Q = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 & 400 & 0 \\ 0 & 0 & 0 & 320 & 0 & 500 \\ 0 & 0 & 0 & 320 & 0 & 0 \\ 0 & 400 & 256 & 0 & 400 & 0 \\ 320 & 0 & 0 & 320 & 0 & 500 \\ 0 & 400 & 0 & 0 & 400 & 500 \end{bmatrix} \end{array}$$

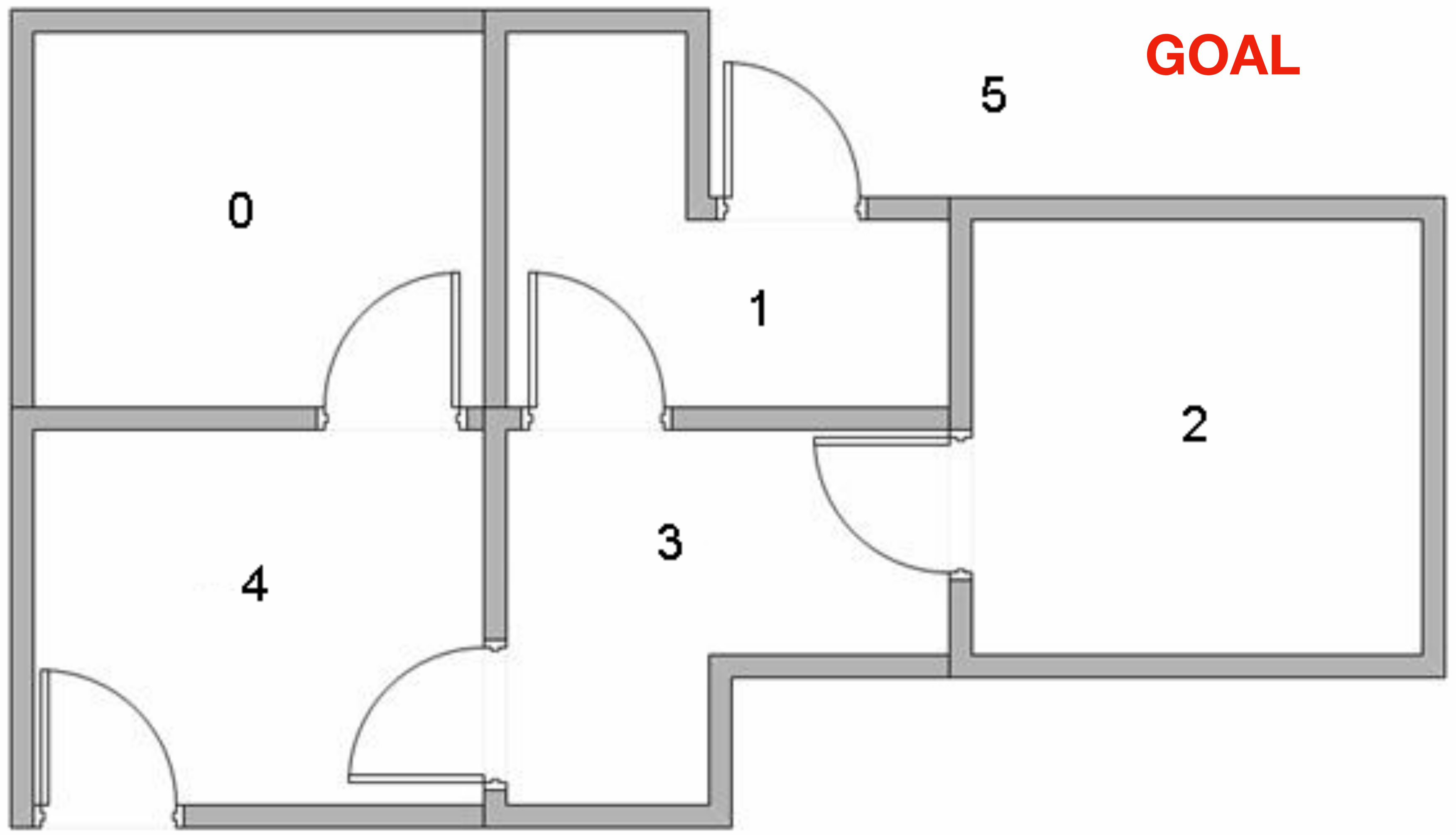
Normalize

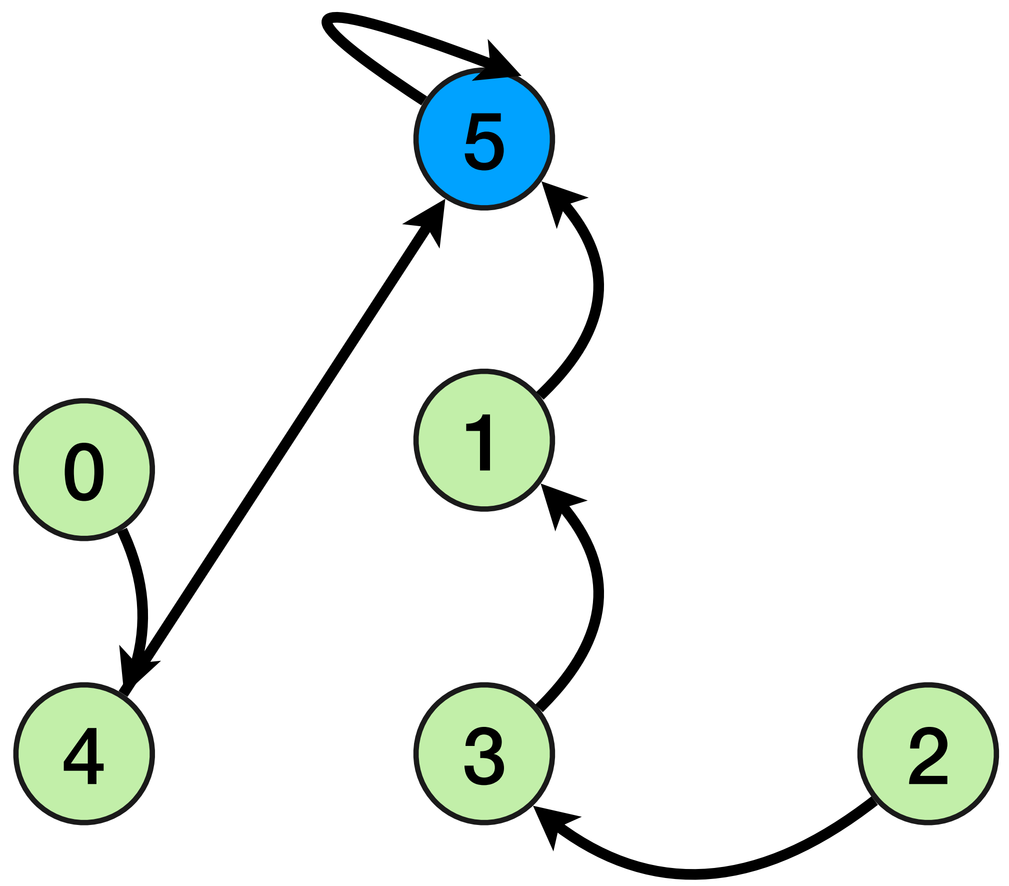
$$Q = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 & 80 & 0 \\ 0 & 0 & 0 & 64 & 0 & 100 \\ 0 & 0 & 0 & 64 & 0 & 0 \\ 0 & 80 & 51 & 0 & 80 & 0 \\ 64 & 0 & 0 & 64 & 0 & 100 \\ 0 & 80 & 0 & 0 & 80 & 100 \end{bmatrix} \end{array}$$

Optimal policy

$$\pi^*(s) = \arg\max_{a \in A} Q(s, a)$$

$$\pi^* = [4, 5, 3, 1, 5, 5]$$

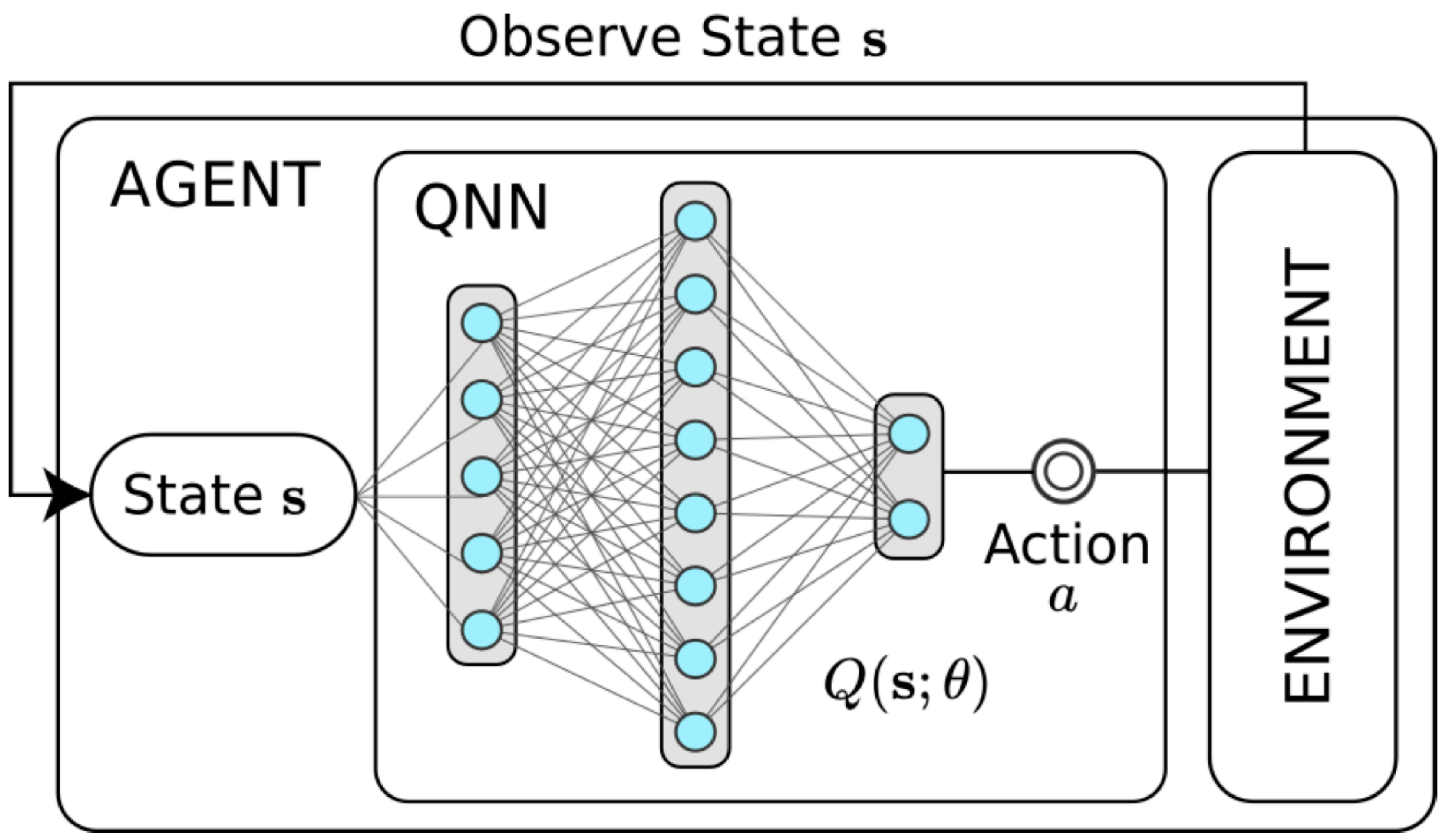




Deep Reinforcement Learning

- What's the problem with Q-learning?
- We need to keep in memory a table of $|S| \cdot |A|$ entries. State space is generally **HUGE**!
- If you need a dense state space, exploit **Deep Reinforcement Learning (DRL)**
- A neural network (a function approximator) can predict Q values (one for each action) given a state as input. Just learn to minimise this loss function

$$L(\theta) = \Big(\underbrace{r + \gamma \max_{a'} Q(\hat{\mathbf{s}};\theta)[a']}_{\text{target}} - \underbrace{Q(\mathbf{s};\theta)[a]}_{\text{prediction}}\Big)^2$$



- Read more about it, or ask: **Mnih, Volodymyr, et al.** "Playing atari with deep reinforcement learning."

Go Code Some RL Algorithm

- Go to https://github.com/matteoprata/markov-processes-class
- To get these slides and a Jupyter Notebook with **Value Iteration**, **Policy Iteration** and **Q-Learning** algorithms implemented on the two examples showed in class