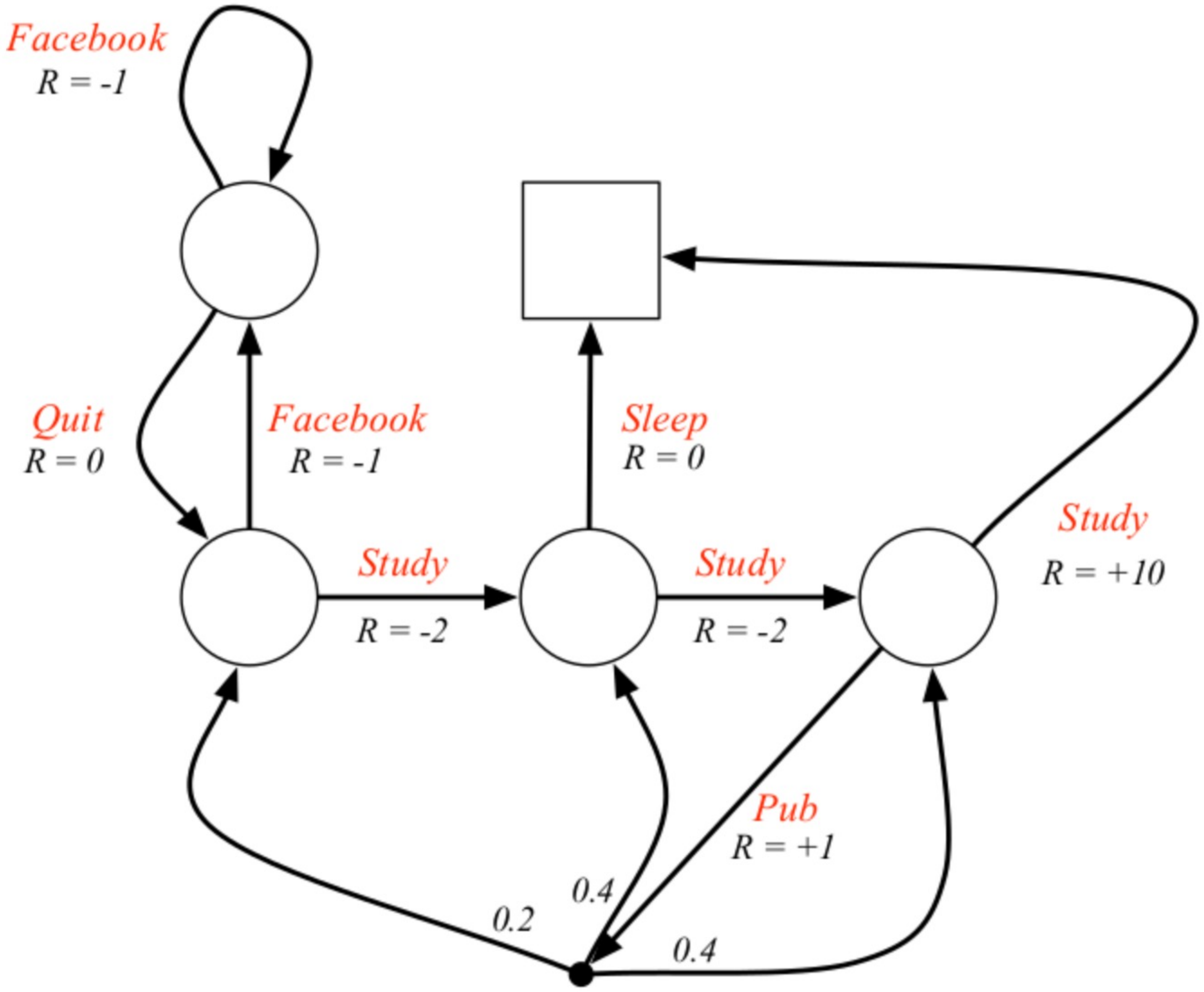
CSCI 3202: Intro to Artificial Intelligence

Lecture 3: Markov Decision Processes

Rhonda Hoenigman

Department of Computer Science

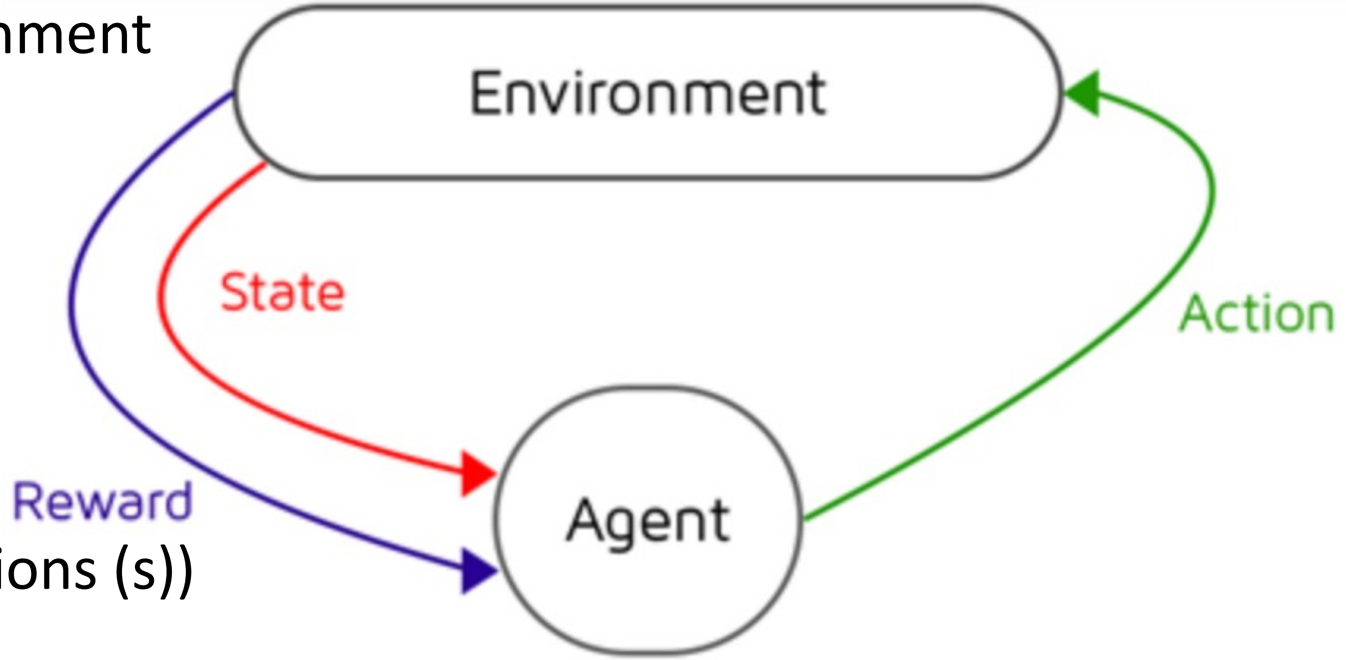


Student MDP with actions

Markov Decision Process – Overview

A Markov Decision Process (MDP):

- Sequential decision problem
- Fully observable, stochastic environment
- Markovian transition model
- Additive reward structure



Requires:

- States (call them s, with initial s_0)
- Actions available in each state (Actions (s))
- Transition model (P(s’ | s, a))
- Reward function R(s) (or R(s, a, s’))

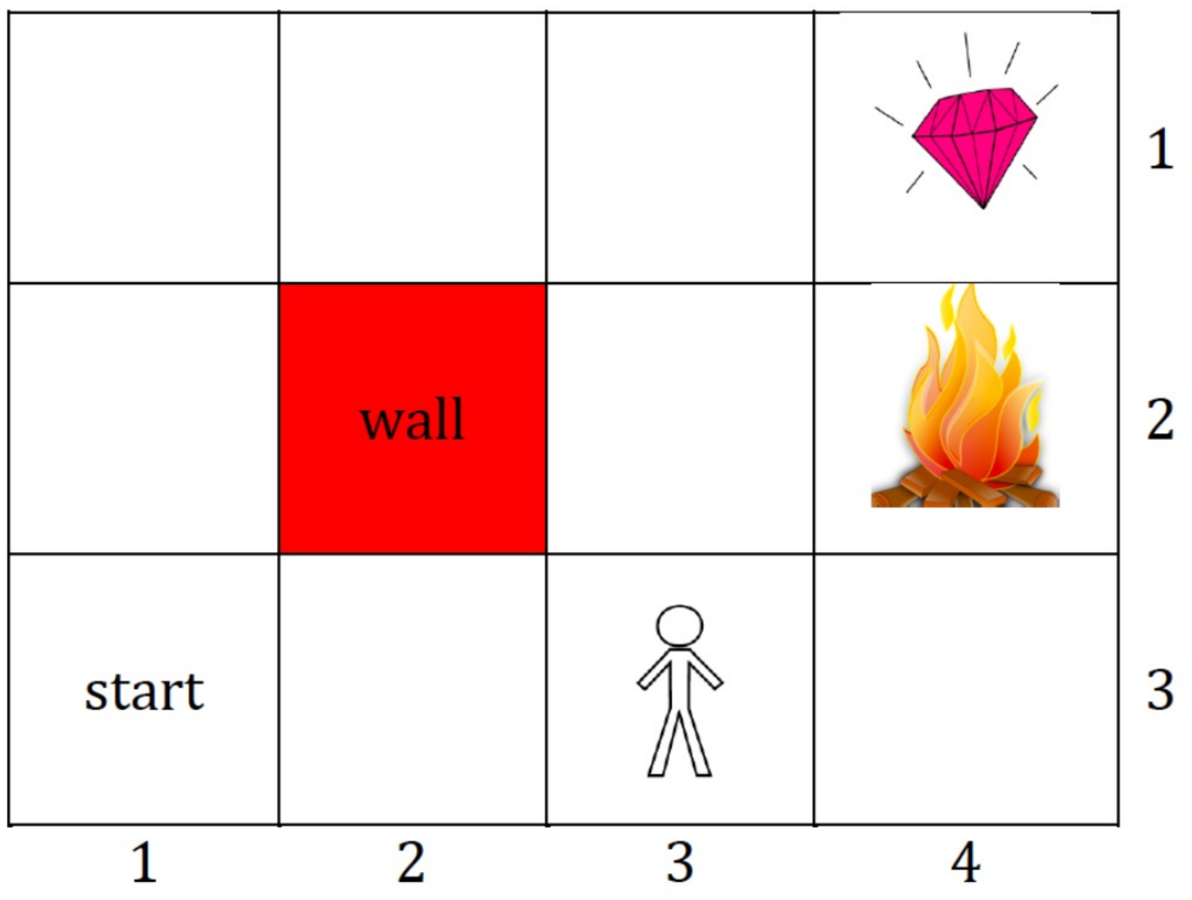
Markov Decision Process

Example: Move agent from "start" to the diamond without falling into the fire pit.

Deterministic – simple

Stochastic transition model:

- Desired direction: 0.8
- Either side: 0.1

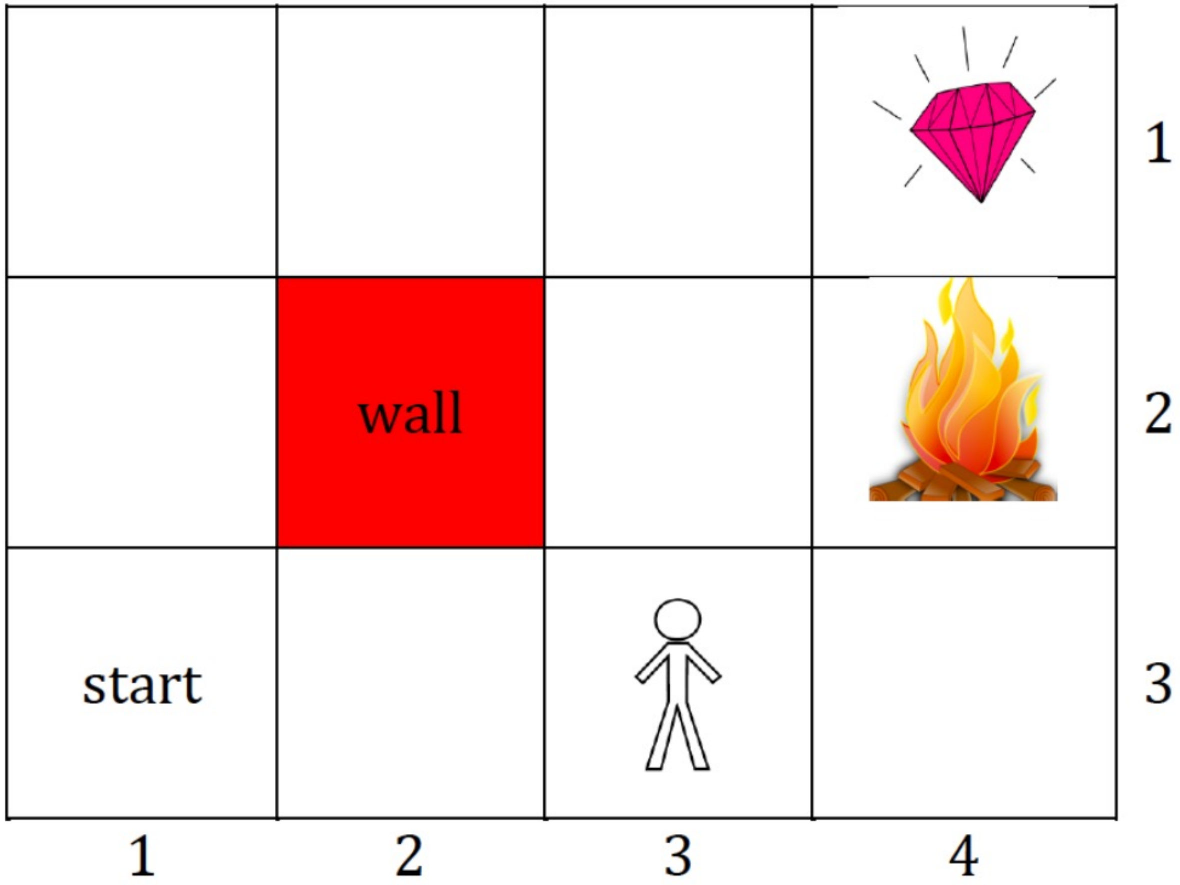


- *If agent goes into a wall, the agent just bounces off and stays in the same location.*

Markov Decision Process

Tell the agent what to do in each possible state s, so that the agent can reach the goal (treasure).

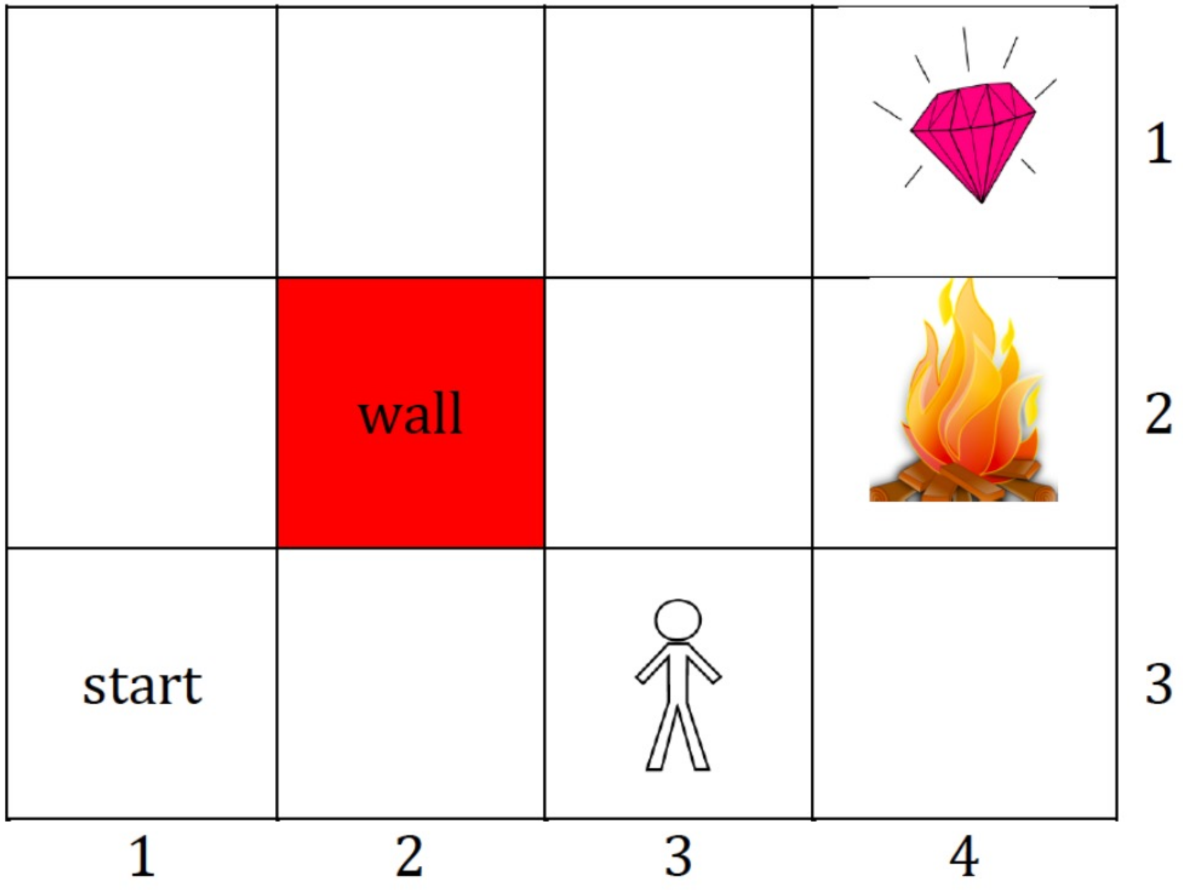
➢ Called **policies** in this context.



Markov Decision Process

Rewards - short-term gain

- Treasure: +1
- Fire pit: -1
- Each state: $R(s) = ?$

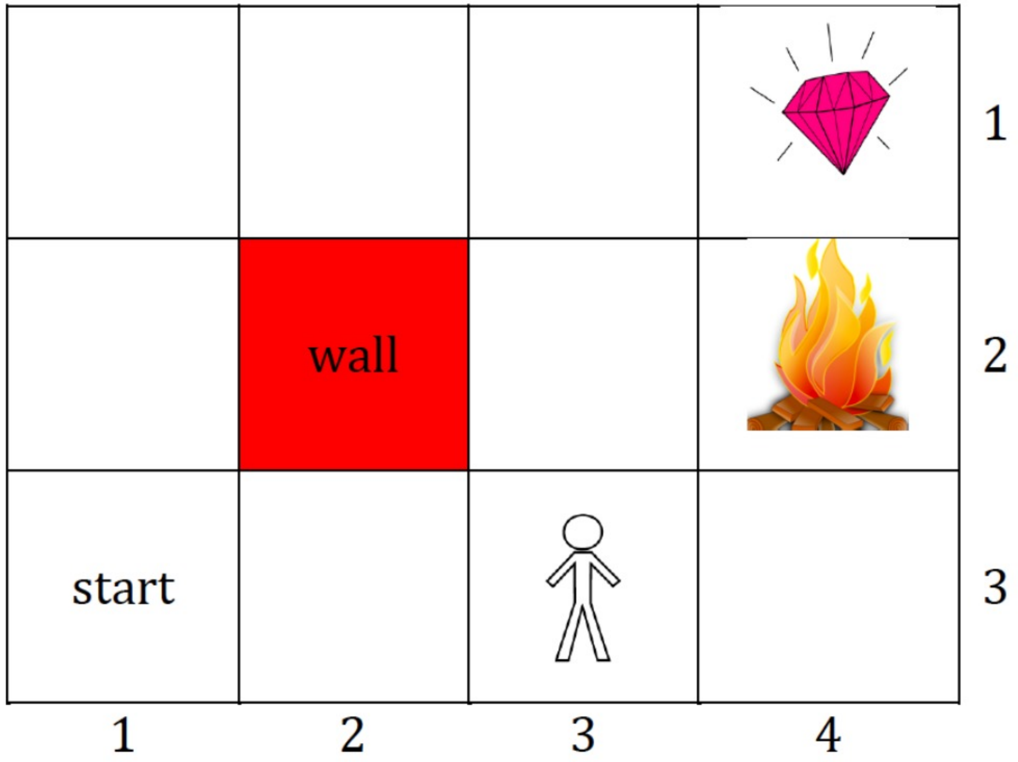


Markov Decision Process

Utility - long-term gain

- Depends on entire sequence of states
$$[s_0, s_1, \dots, s_{50}, s_{51}, \dots]$$

- Rough definition:
Utility = sum of rewards along the sequence



Markov Decision Process

Time Horizon

- **Finite Horizon:** after fixing some time N, nothing matters.

  Length of horizon could affect optimal move for a given state.

- **Infinite Horizon**: no reason to behave differently in the same state at different times.

Markov Decision Process

Discounting

- Preference for immediate reward as opposed to future rewards.

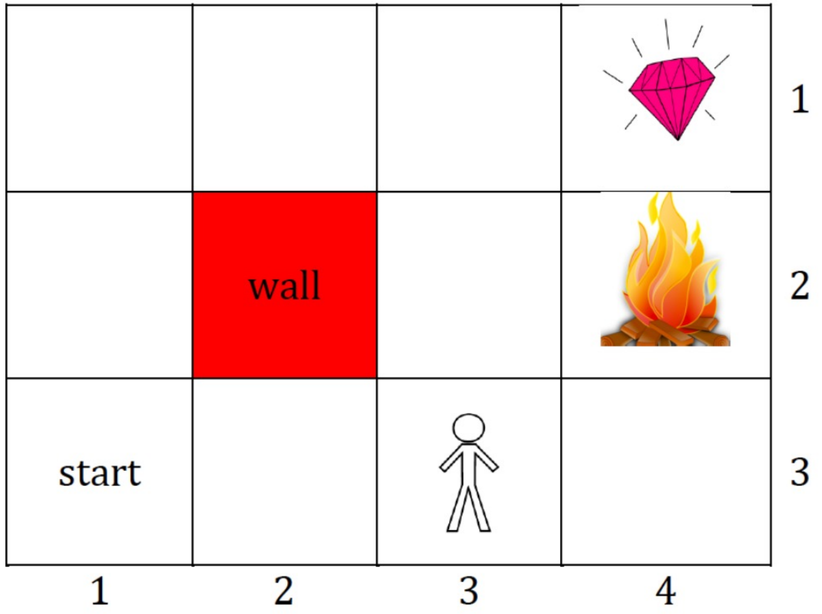
- Reduce future rewards by a discount factor γ

Markov Decision Process

Determine: optimal policy starting from some state s?

- Expected utility under policy π is:

$$U^{\pi}(s) = E\left[\sum_{t=0}^{\infty} \gamma^{t} R(S_t)\right]$$



Markov Decision Process

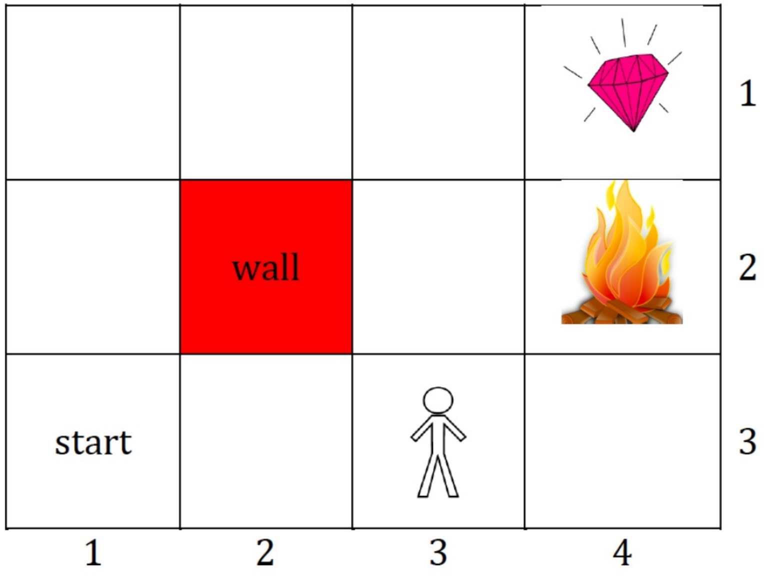
Maximize expected discounted utility

➢ Optimal policy starting from s

$$\pi_s^* = \arg\max_{\pi} U^{\pi}(s)$$

- But policy recommends action for any state, not just one. The optimal policy: π^*

- True utility of a state is $U^{\pi^*}(s)$ is the expected discounted sum of rewards if the agent executes an optimal policy from s…. just write this as $U(s)$



Markov Decision Process

If we know U(s), pick actions to maximize its expected value!

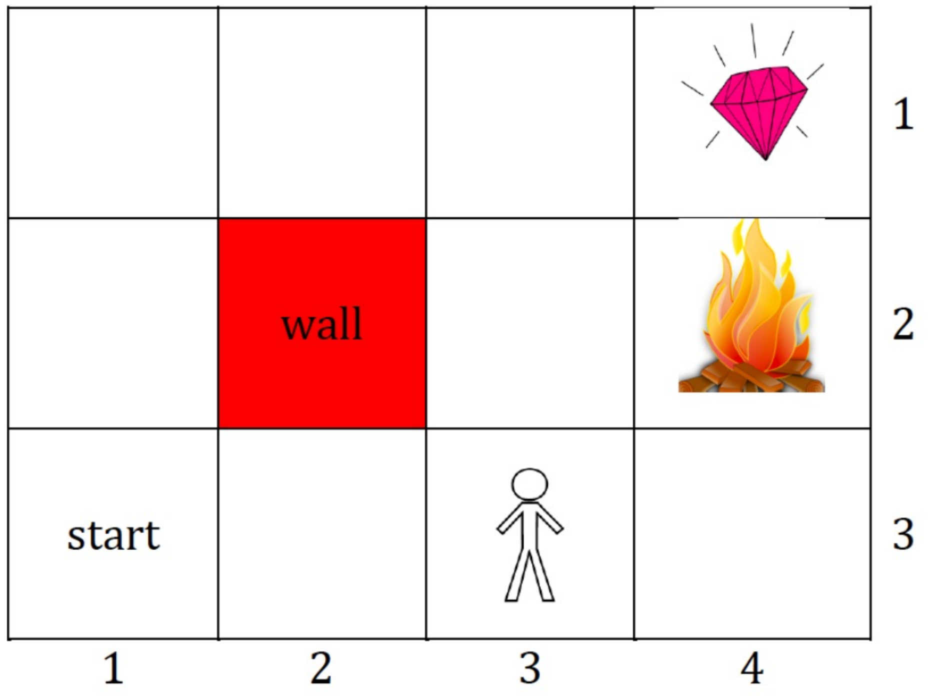
S'pose we are in state *s*

→ P(*s'* | *s*, *a*) = prob of going to state *s'* by action *a*

→ Expected utility is: $\sum_{s'} P(s' \mid s, a)\, U(s')$

→ Optimal policy in s is then:

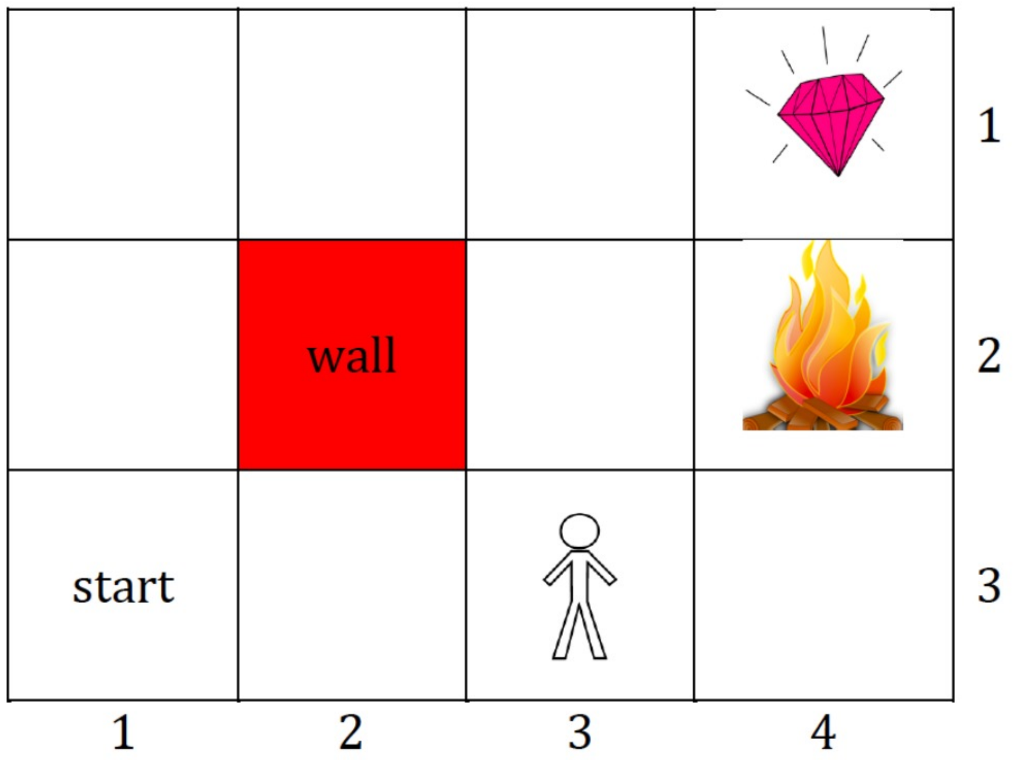
$$\pi^*(s) = \arg\max_{a \in A(s)} \sum_{s'} P(s' \mid s, a)\, U(s')$$



Markov Decision Process – Value Iteration

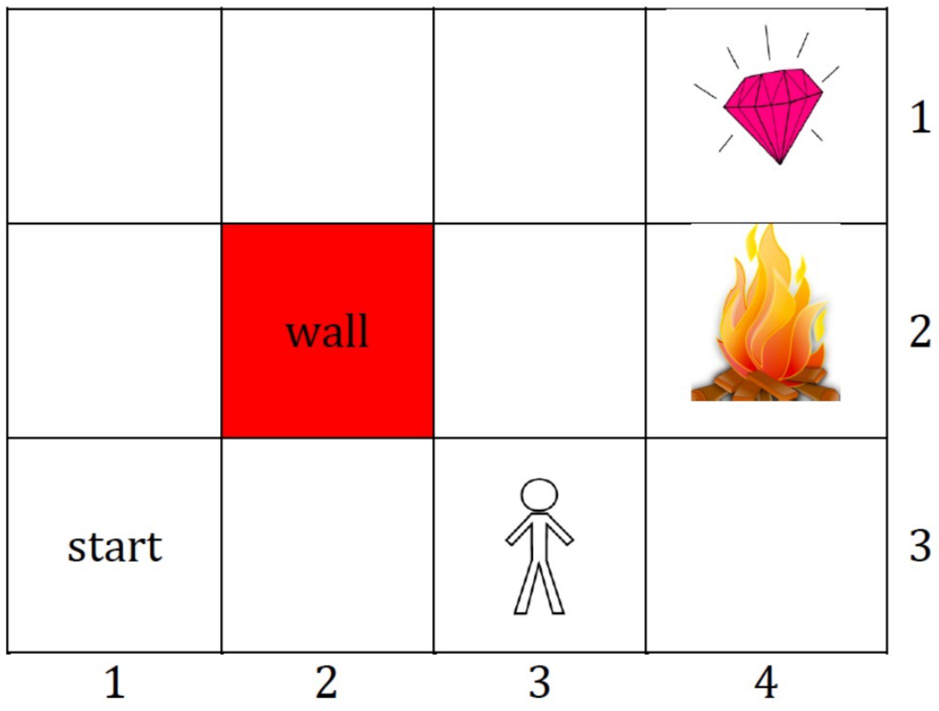
Calculate utilities iteratively using Value Iteration algorithm

- Terminal rewards: +1 for treasure, -1 for fire pit
- Living reward: -.03
- Discount factor: 0.9



Value Iteration

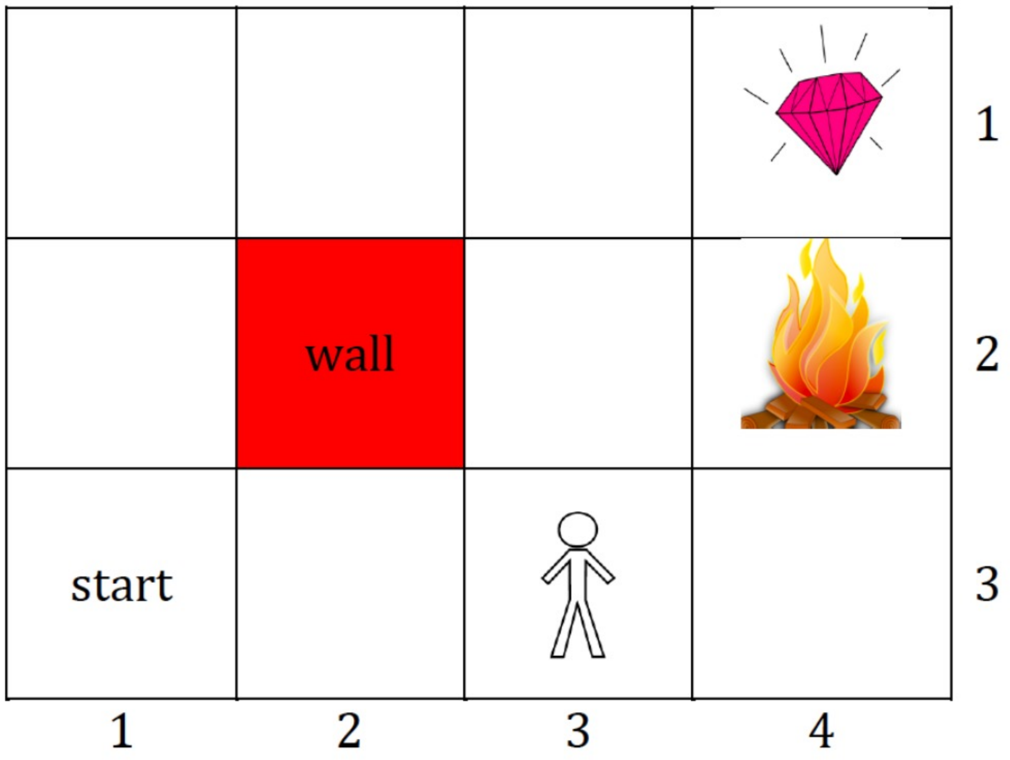
- To find the optimal **utilities**
- Start with some candidate set of utilities for each state. Do the following many times:
- For each state *s*:
  - What are all the actions available? (*a*)
  - For each action *a*:
    - What are the next states, and with what probabilities? ($P(s' \mid s, a)$)
    - Calculate expected utility w/ action *a*
  - Update utility of *s* to max of discounted expected utilities, plus reward of *s*



Value Iteration – Bellman Equations

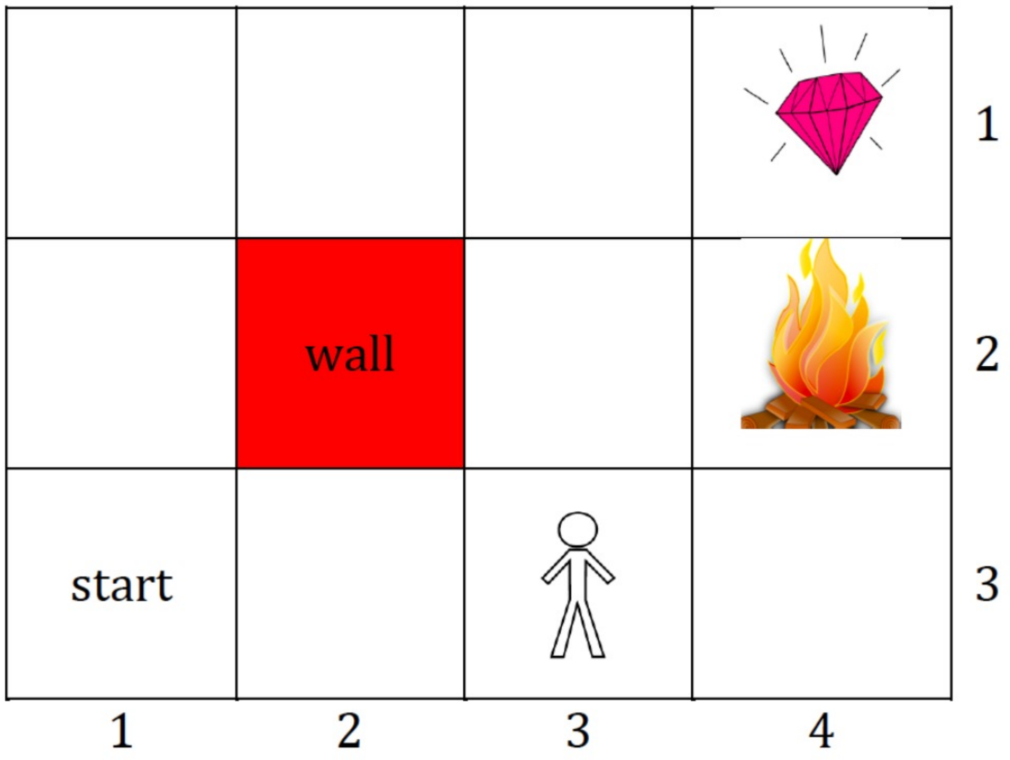
$$U_{i+1}(s) = R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s'|s,a)\, U_i(s')$$

Example: Find U_0 and U_1 for all states.



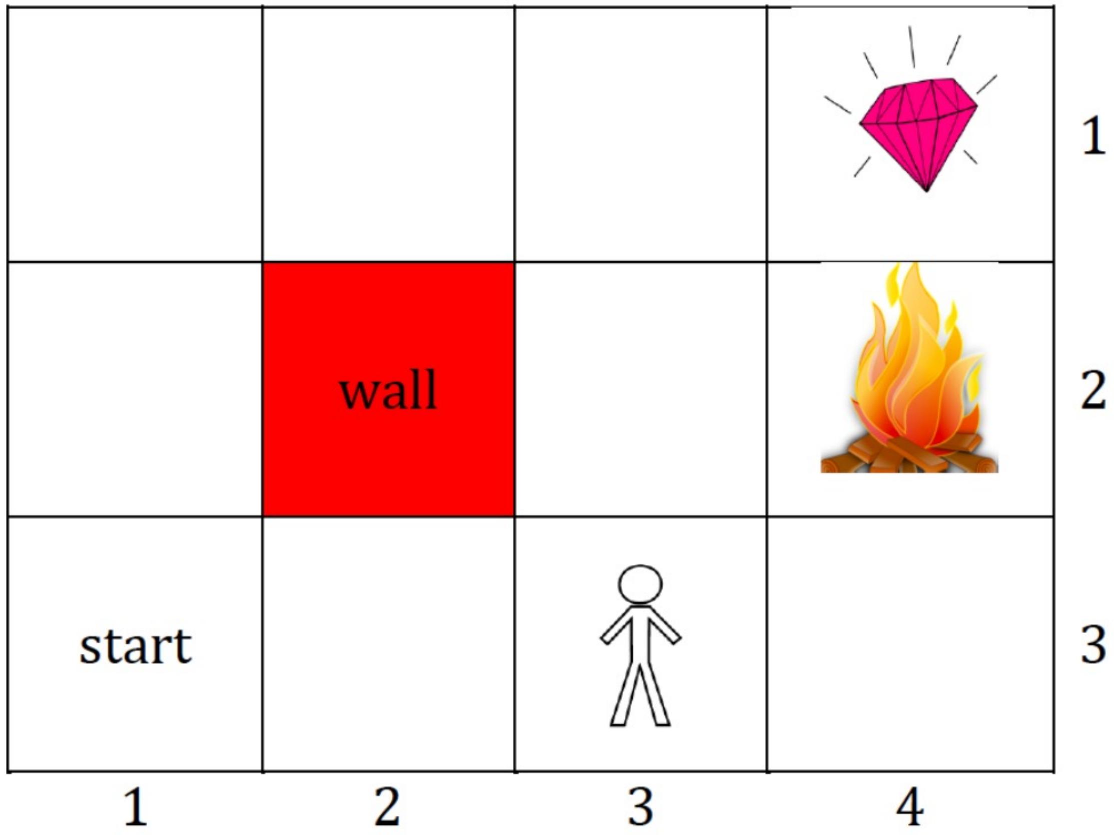
Value Iteration – Bellman Equations

$$U_{i+1}(s) = R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s'|s,a)\, U_i(s')$$



Value Iteration

Example: Find $U_2(s)$ for state s = (3, 1)



Value Iteration

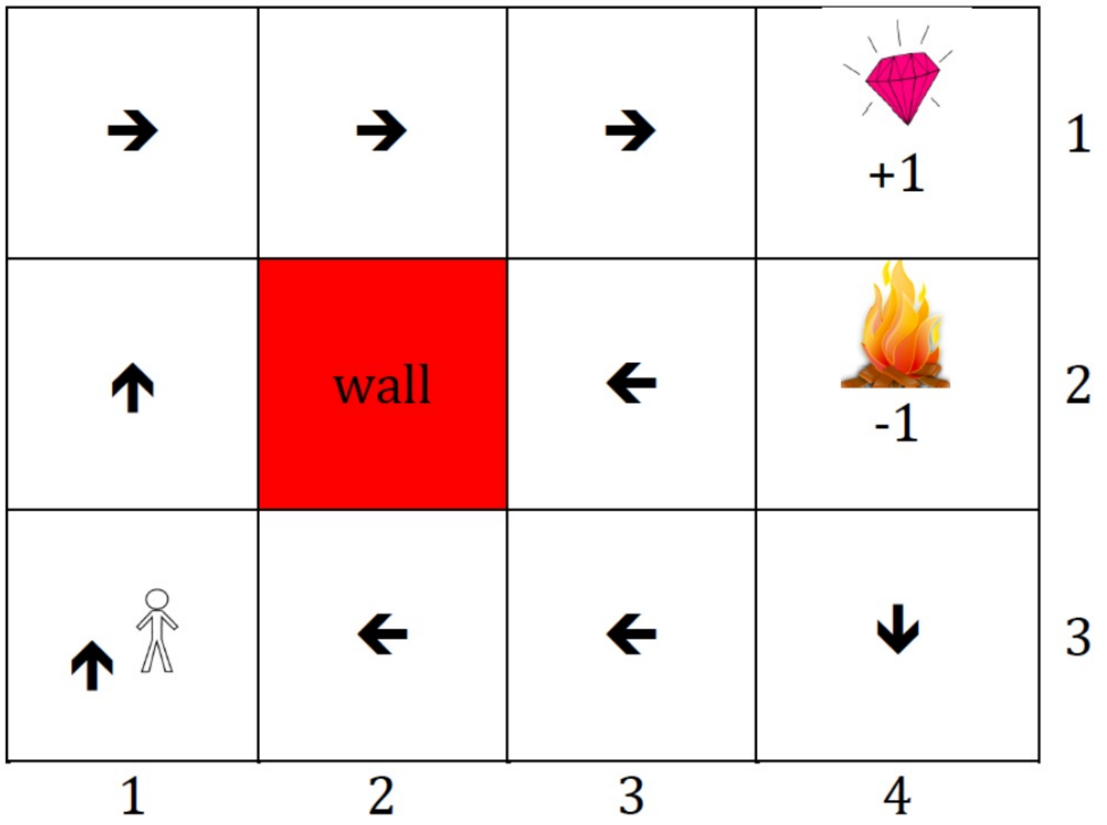
```
def value_iteration(mdp, tolerance):

    # initilize utility for all states

    # iterate:

        # make a copy of current utility, to be modified

        # initialize maximum change to 0

        # for each state s:

            # for each available action, what next states
            # are possible, and their probabilities?

            # calculate the maximum expected utility

            # new utility of s = reward(s) +
            #                    discounted max expected utility

            # update maximum change in utilities, if needed

        # if maximum change in utility from one iteration to the
        # next is less than some tolerance, break!

    return # final utility
```

Value Iteration

Markov Decision Process – examples of optimal policies

Environment:

- 80% of time action is successful, and 20% of time action is unsuccessful, goes left 10% and right 10%.
- R(s) = -0.01 - Slightly negative living reward.
- On each iteration, the living reward is deducted from the agent's utility.
- No discount factor



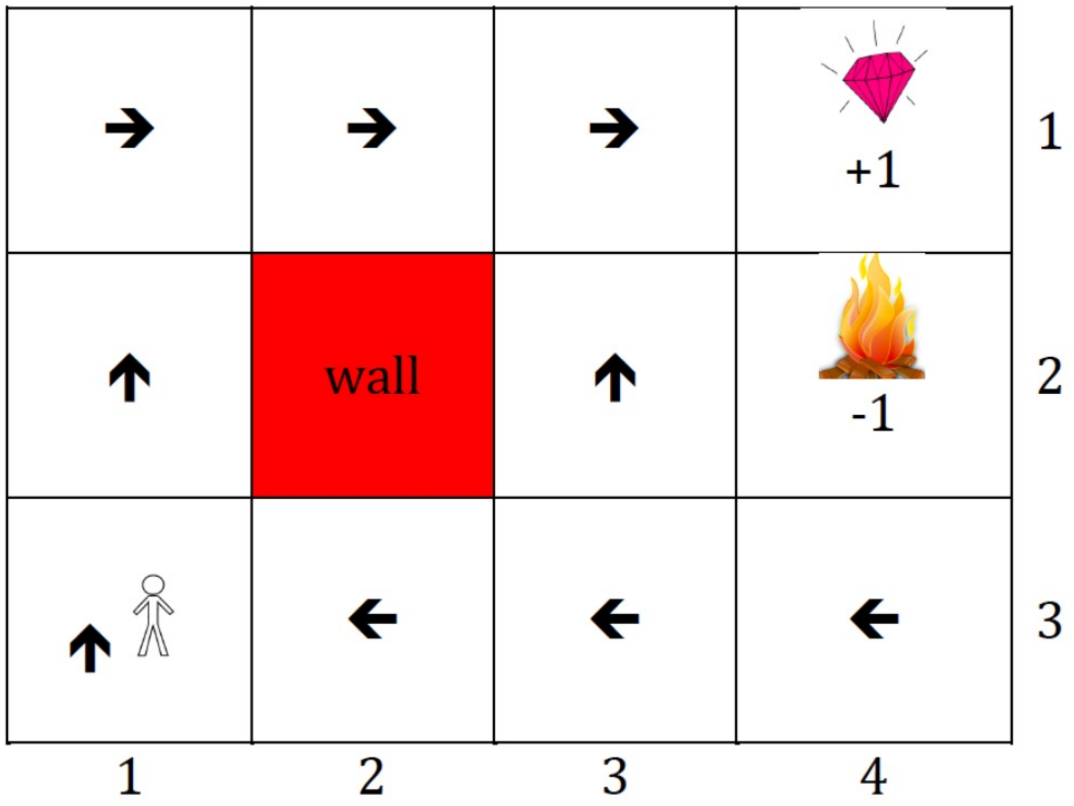
The optimal policy involves avoiding the fire pit at all costs. In the square next to the pit, run into the wall to avoid the risk of accidentally going into the pit. The living reward deduction is so small that the agent has time to hit the wall over and over.

Markov Decision Process – examples of optimal policies

Environment:

- 80% of time action is successful, and 20% of time action is unsuccessful, goes left 10% and right 10%.
- R(s) = -0.03
- No discount factor

It is no longer the best policy to run into the wall. The increased risk of dying in the fire is worth it to speed up the path to the diamond.

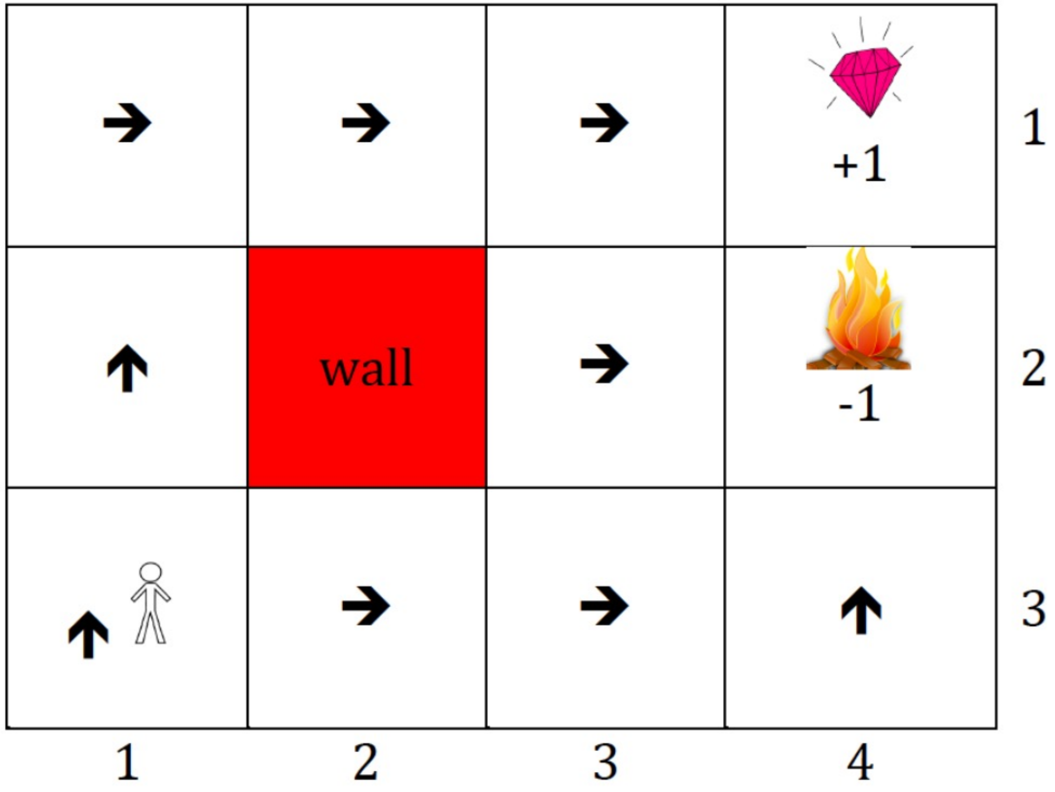


Markov Decision Process – examples of optimal policies

Environment:

- 80% of time action is successful, and 20% of time action is unsuccessful, goes left 10% and right 10%.
- R(s) = -2.0
- No discount factor

Jumping into the fire pit is preferable to living.



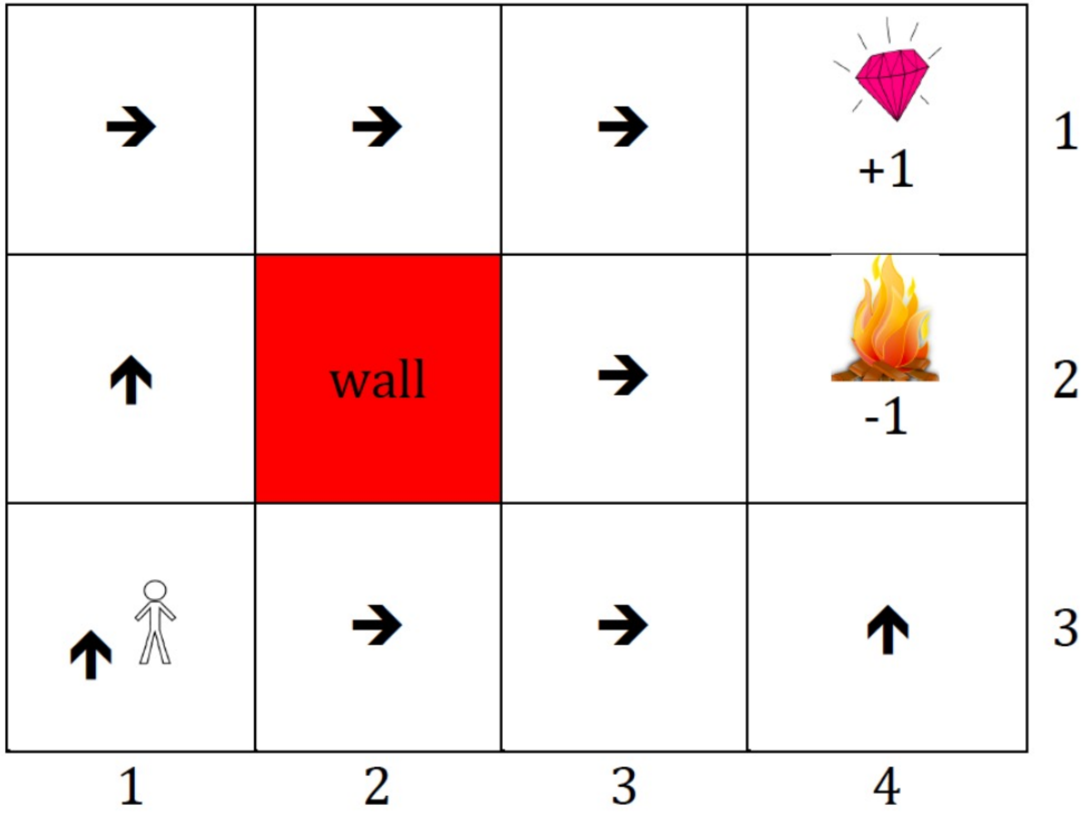
Policy Iteration

Changes to U(s) may not result in policy change.

Iterate through policies instead of utilities.

Want: A policy π that provides the best action in each state s.

$\pi(s)$ is the action in state s by policy π, e.g. up $\uparrow$



Policy Iteration

Policy Iteration: Uses policy evaluation and policy improvement, returns π

Two steps:

Policy Evaluation: given policy π_i

$U_i = U_i^{\pi}$ - the utility of each state if policy π_i were executed

- only one action, not all possible actions in state

$$U_i(s) = R(s) + \gamma \sum_{s'} P(s' \mid s, \pi_i(s))\, U_i(s')$$

Policy improvement:

Calculate the new policy π_{i+1} using π_i and U_i

- Iterate between **policy evaluation** and **policy improvement** steps
- Do this until the policy is unchanged by the evaluation/improvement

Policy Iteration

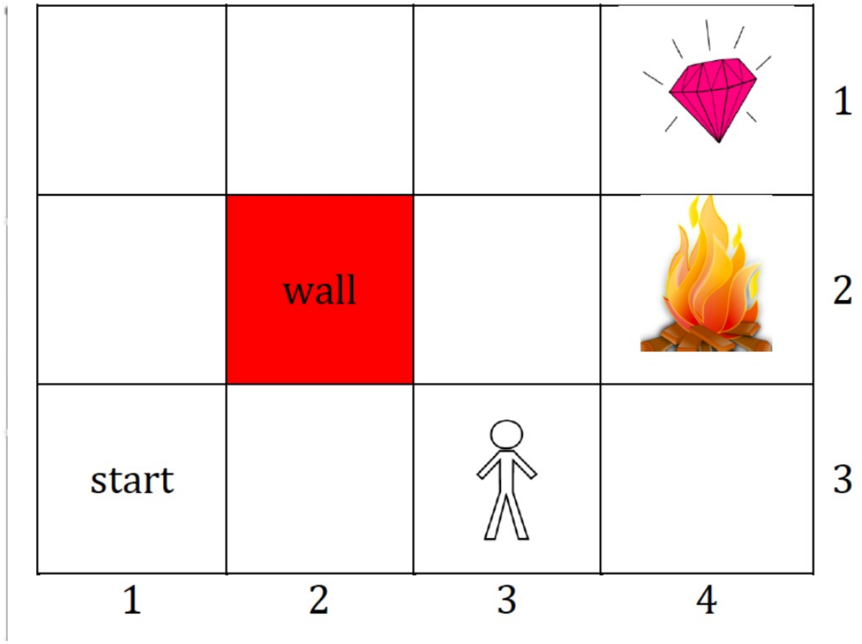
Policy evaluation: useful approximation

$$U_i(s) = R(s) + \gamma \sum_{s'} P(s' \mid s, \pi_i(s))\, U_i(s')$$

Remember the Bellman equations for value iteration?

$$U_{i+1}(s) = R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s' \mid s, a)\, U_i(s')$$

Instead of solving the linear system ($O(n^3)$), we can just iterate these a bunch of times to solve for an approximate utility for each state.



Policy Iteration

Policy improvement:

For each state, can we pick a better action?

$$\max_{a \in A(s)} \sum_{s'} P(s' \mid s, a)\, U(s') > \sum_{s'} P(s' \mid s, \pi(s))\, U(s')$$

If so, set π(s) = action that maximizes this expected utility

Policy Iteration

```
def policy_iteration(mdp):

    # initilize utility for all states

    # initialize a policy for each state, being a random action

    # iterate:

        # update utility, using policy evaluation and
        # current estimates of utility and policy

        # initialize unchanged = True

        # for each state s:

            # among the possible actions, which yields
            # the maximum expected utility?

            # if the best action choice is not currently
            # the policy for s, update it

        # if no policy values are changed, break!

    return # final policy (and/or utility)

def policy_evaluation(policy, utility, mdp, n_iter):
    # do a handful of value iteration updates of
    # the input utility, under the given policy
    return # updated utility
```