# MDPs: Markov Decision Processes

## Julia Academy: POMDPs.jl

Decision Making Under Uncertainty

# What is an MDP?

**Definition: MDP.** A *Markov decision process* (MDP) is a *problem formulation* that defines how an agent takes sequential *actions* from *states* in its environment, guided by *rewards*—using uncertainty in how it *transitions* from state to state.

- Formally, an MDP is defined by the following:

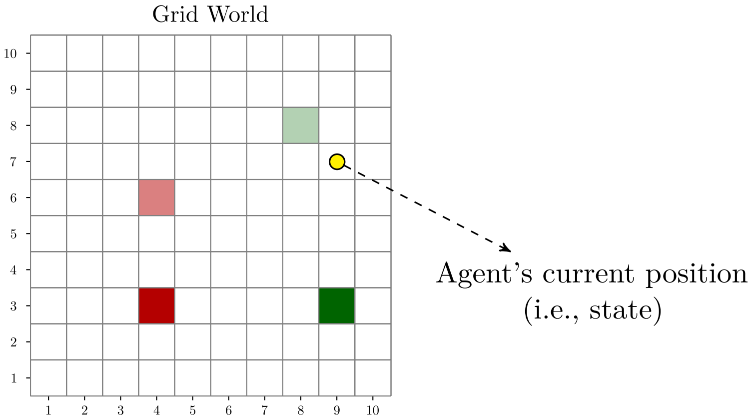
Table: MDP Problem Formulation: $\langle \mathcal{S}, \mathcal{A}, T, R, \gamma \rangle$

| Variable | Description | `POMDPs` Interface |
|---|---|---|
| $\mathcal{S}$ | State space | `POMDPs.states` |
| $\mathcal{A}$ | Action space | `POMDPs.actions` |
| $T(s' \mid s, a)$ | Transition function | `POMDPs.transition` |
| $R(s, a)$ | Reward function | `POMDPs.reward` |
| $\gamma \in [0, 1]$ | Discount factor | `POMDPs.discount` |

Remember, an MDP is a ***problem formulation*** and ***not an algorithm***.
An MDP formulation enables the use of solution methods, i.e. algorithms.

# MDP Example: Grid World

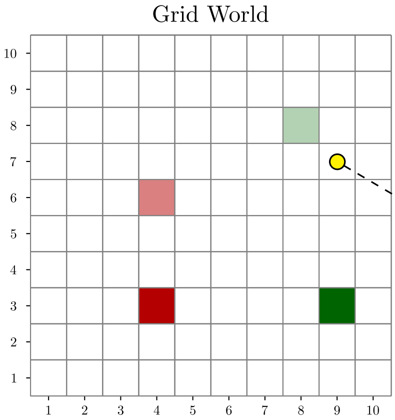
In the **Grid World** problem, an *agent* moves around a grid attempting to collect as much reward (**green cells**) as possible, avoiding negative rewards (**red cells**).



# MDP: State space

**Definition: State space $\mathcal{S}$.**
A set of all possible ***states*** an agent can be in (discrete or continuous).

**Grid World example**:
All possible $(x, y)$
cells in a $10 \times 10$ grid
(i.e., 100 discrete states)



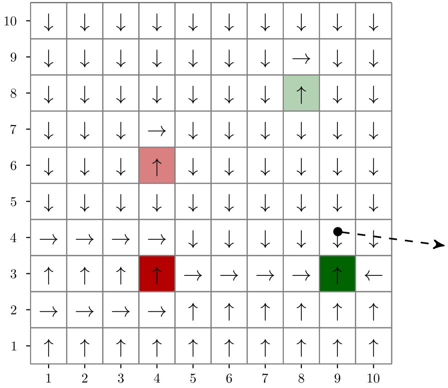

**State**
$(x, y)$ of $(9, 7)$

# MDP: Action space

**Definition: Action space $\mathcal{A}$.**
A set of all possible ***actions*** an agent can take (discrete or continuous).

**Grid World example**:
The four (discrete) cardinal directions: $[$`up`, `down`, `left`, `right`$]$

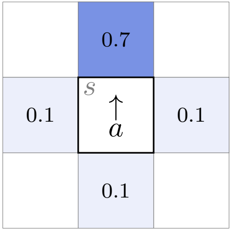
Grid World Actions



# MDP: Transition function

**Definition: Transition function**[1] $T(s' \mid s, a)$.
Defines how the agent ***transitions*** from the current state $s$ to the next state $s'$ when taking action $a$.
Returns a ***probability distribution*** over all possible next states $s'$ given $(s, a)$.

**Grid World example**:
Stochastic transitions (incorporates randomness/uncertainty).
Action $a =$ `up` from state $s$.
70% chance of transitioning correctly.
30% chance (10% $\times$ 3) of transitioning incorrectly.[2]




[1] Sometimes called the *transition model*.

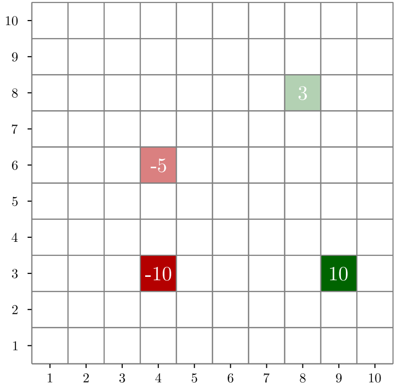
[2] i.e., a different action is taken.

# MDP: Reward function

**Definition: Reward function**[1] $R(s, a)$**.**
A defines the ***reward*** an agent receives when taking action $a$ from state $s$.

**Grid World example**:
Two cells contain **positive rewards**
and two cells contain **negative rewards**,
all others are **zero**.

Grid World Rewards

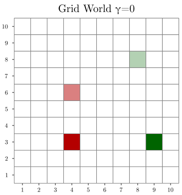



[1] Sometimes called the *reward model*.

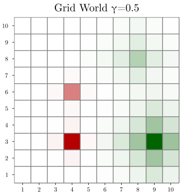
# MDP: Discount factor

**Definition: Discount factor $\gamma \in [0, 1]$.**

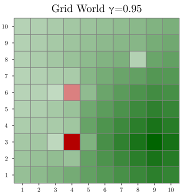
The ***discount factor*** controls how myopic (short-sighted) the agent is in its decision making (e.g., when $\gamma = 0$, the agent only cares about immediate rewards (myopic) and as $\gamma \to 1$, the agent takes in potential future information in its decision making process).



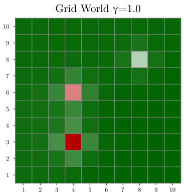

(a) Short-sighted (no reward spread)




(b) Some future reward[1] is spread




(c) Future reward is nicely spread



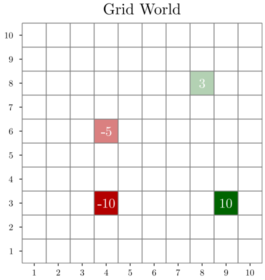

(d) Dominated by the future reward

[1] The sum of the *discounted future rewards* is called the *utility* $U(s)$ or the *value* $V(s)$ of a state.

# QuickPOMDPs: Grid World

```julia
using POMDPs, POMDPModelTools, QuickPOMDPs

struct State; x::Int; y::Int end # State definition
@enum Action UP DOWN LEFT RIGHT # Action definition

𝒮 = [[State(x,y) for x=1:10, y=1:10]..., State(-1,-1)] # State-space
𝒜 = [UP, DOWN, LEFT, RIGHT] # Action-space

const MOVEMENTS = Dict(UP=>State(0,1), DOWN=>State(0,-1), LEFT=>State(-1,0), RIGHT=>State(1,0))
Base.:+(s1::State, s2::State) = State(s1.x + s2.x, s1.y + s2.y) # Helper for applying actions

function T(s, a) # Transition function
    R(s) != 0 && return Deterministic(State(-1,-1))
    Nₐ = length(𝒜)
    next_states = Vector{State}(undef, Nₐ + 1)
    probabilities = zeros(Nₐ + 1)
    for (i, a′) in enumerate(𝒜)
        prob = (a′ == a) ? 0.7 : (1 - 0.7) / (Nₐ - 1)
        destination = s + MOVEMENTS[a′]
        next_states[i+1] = destination
        if 1 ≤ destination.x ≤ 10 && 1 ≤ destination.y ≤ 10
            probabilities[i+1] += prob
        end
    end
    (next_states[1], probabilities[1]) = (s, 1 - sum(probabilities))
    return SparseCat(next_states, probabilities)
end

function R(s, a=missing) # Reward function
    if s == State(4,3)
        return -10
    elseif s == State(4,6)
        return -5
    elseif s == State(9,3)
        return 10
    elseif s == State(8,8)
        return 3
    end
    return 0
end

abstract type GridWorld <: MDP{State, Action} end

mdp = QuickMDP(GridWorld,
    states     = 𝒮,
    actions    = 𝒜,
    transition = T,
    reward     = R,
    discount   = 0.95,
    isterminal = s->s==State(-1,-1));
```

- This code[a] defines the entire *Grid World* problem using `QuickPOMDPs.jl`
    - Just a sneak-peek: we'll walk through this in detail in the `Pluto` notebooks




[a] Yes, this is self-contained—copy and paste it into a notebook or REPL!

# MDP SOLVERS

A number of ways to solve MDPs are implemented in the following packages.

Table: MDP Solution Methods

| **Package** | **Online/Offline** | **State Spaces** | **Actions Spaces** |
|---|---|---|---|
| `DiscreteValueIteration.jl` | Offline | Discrete | Discrete |
| `LocalApproximationValueIteration.jl` | Offline | Continuous | Discrete |
| `GlobalApproximationValueIteration.jl` | Offline | Continuous | Discrete |
| `MCTS.jl`* | Online | Continuous | Continuous |

* Monte Carlo Tree Search.

When defining your problem, the ***type*** of state and action space is very important!

# Reinforcement learning solvers

Certain problems are better suited in the *reinforcement learning* (RL) domain. Several RL solvers that adhere to the `POMDPs.jl` interface are implemented in the following packages.

Table: Reinforcement Learning Solution Methods

| **Package** | **State Spaces** | **Actions Spaces** | **Algorithms Implemented** |
|---|---|---|---|
| `TabularTDLearning.jl` | Discrete | Discrete | Q-learning, SARSA, SARSA-$\lambda$ |
| `DeepQLearning.jl` | Continuous | Discrete | DQN, Double DQN, Dueling DQN, Recurrent Q-learning |
| `Crux.jl` | Continuous | Continuous | DQN, REINFORCE, PPO, A2C, DDPG, TD3, SAC, Behavior Cloning, GAIL, AdVIL, AdRIL, SQIL, ASAF |

When defining your problem, the ***type*** of state, action, and observation space is very important!