Decision Making Under Uncertainty

CARS Tutorial

Robert Moss

CS PhD Student, Stanford Intelligent Systems Laboratory

mossr@stanford.edu

December, 2022

What is this tutorial?

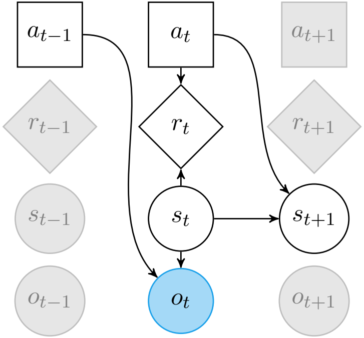


Figure: POMDP Sequence.

- A peek into the `POMDPs.jl` ecosystem of julia packages
- "But what *are* POMDPs?"
  - POMDPs are a *problem formulation* that enable optimal[1] sequential decisions to be made in uncertain environments.
- Teaching *by example* using interactive `Pluto.jl` notebooks
  - No prior knowledge of MDPs/POMDPs necessary—all are welcome!
  - Can also be used as a refresher on *decision making under uncertainty*.
  - Target audience is wide, but familiarity with Julia is helpful.

[1] or *approximately* optimal.

Topics covered in this course

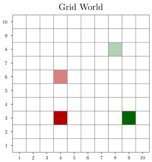
All topics highlight packages that adhere to the `POMDPs.jl` interface.

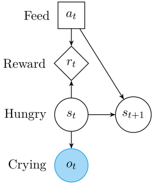
- **Sequential Decision Making**
  - *Markov decision processes* (MDPs)
  - *Partially observable Markov decision processes* (POMDPs)
- **Solution Methods**: Algorithms to solve MDPs/POMDPs
  - *Online* and *offline* solvers
  - *Value function approximation*
- **Simulations**
- **State Estimation using Particle Filters**
- **Reinforcement Learning**
- **Deep Reinforcement Learning**
- **Imitation Learning**

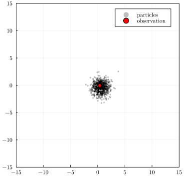
Example problems covered in this course

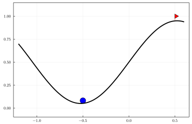
Common problems in the literature are used as running examples.

- $^{\text{(MDP)}}$ **Grid World**: Agent moving around a grid world, looking for rewards.
- $^{\text{(POMDP)}}$ **Crying Baby**: When to feed a baby, based on crying observations.
- $^{\text{(MDP)}}$ **1D Random Walk**: Agent moves around the number line.
- $^{\text{(POMDP)}}$ **2D Random Walk**: Estimating state of a moving agent based on observations.
- $^{\text{(MDP)}}$ **Mountain Car**: Reach a goal up a hill, starting in a valley.
- $^{\text{(MDP)}}$ **Swinging Pendulum**: Balance a swinging pendulum upright.









`POMDPs.jl` PACKAGE ECOSYSTEM

The `POMDPs.jl` package itself contains the interface to define problem definitions.

Other packages provide supporting tools that contain most of the functionality:[1]

- `QuickPOMDPs.jl`
- `POMDPModelTools.jl`
- `POMDPPolicies.jl`
- `POMDPSimulators.jl`
- `POMDPModels.jl`
- `POMDPGallery.jl`
- `BeliefUpdaters.jl`
- `ParticleFilters.jl`
- `POMDPModelChecking.jl`
- `POMDPStressTesting.jl`
- `DiscreteValueIteration.jl`
- `LocalApproximationValueIteration.jl`
- `GlobalApproximationValueIteration.jl`
- `MCTS.jl`
- `TabularTDLearning.jl`
- `DeepQLearning.jl`
- `Crux.jl`
- `QMDP.jl`
- `FIB.jl`
- `BeliefGridValueIteration.jl`
- `SARSOP.jl`
- `BasicPOMCP.jl`
- `ARDESPOT.jl`
- `MCVI.jl`
- `POMDPSolve.jl`
- `IncrementalPruning.jl`
- `POMCPOW.jl`
- `AEMS.jl`
- `PointBasedValueIteration.jl`

[1] *Key*: Tools, Extensions, MDP solvers, POMDP solvers.

Other resources

There are many *excellent* resources on MDPs/POMDPs and reinforcement learning:

- ***Algorithms for Decision Making*, Kochenderfer, Wheeler, & Wray**
  (https://algorithmsbook.com/)
- ***Reinforcement Learning: An Introduction*, Sutton & Barto**
  (http://incompleteideas.net/book/the-book.html)
- ***POMDPs.jl: A Framework for Sequential Decision Making under Uncertainty*, Egorov, Sunberg, et al., Journal of Machine Learning Research, 2017**
  (https://www.jmlr.org/papers/volume18/16-300/16-300.pdf)
- **Introduction to Reinforcement Learning with David Silver**
  (https://deepmind.com/learning-resources/-introduction-reinforcement-learning-david-silver)

Lecture breakdown

Each lecture has an associated `Pluto` notebook detailing the material.

1. **MDPs: Markov Decision Processes**
   - Includes: *planning, reinforcement learning, online/offline solvers, simulations*
2. **POMDPs: Partially Observable Markov Decision Processes**
3. **State Estimation using Particle Filtering**
4. **Approximate Methods for Continuous Spaces**
5. **Deep Reinforcement Learning**
6. **Imitation Learning: Learn from Demonstrations**

MDPs: Markov Decision Processes

CARS Tutorial: 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:

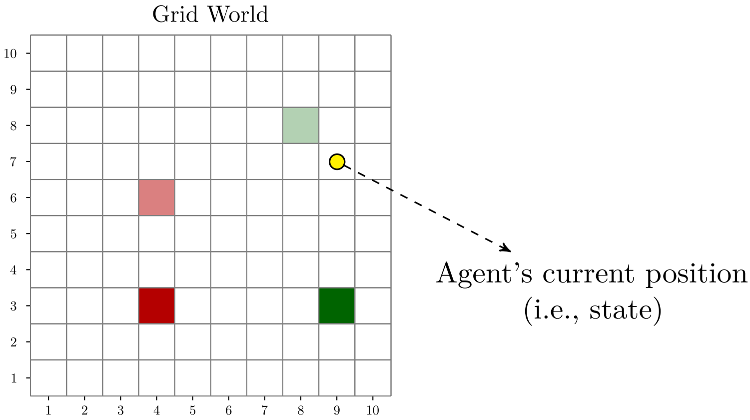
Table: MDP Problem Formulation: $\langle \mathcal{S}, \mathcal{A}, T, R, \gamma \rangle$

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

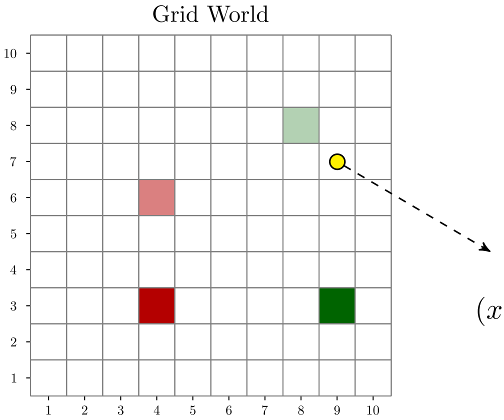
In the **Grid World** problem, an *agent* moves around a grid attempting to collect as much reward (**green cells**) as possible, avoiding negative rewards (**red cells**).



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 10×10 grid
(i.e., 100 discrete states)

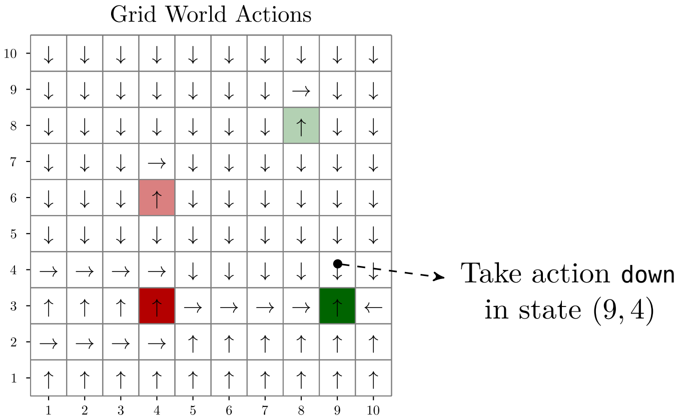


State
(x, y) of $(9, 7)$

MDP: Action space

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

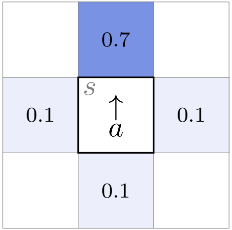
Grid World example:
The four (discrete) cardinal directions:
$[$`up`, `down`, `left`, `right`$]$



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]



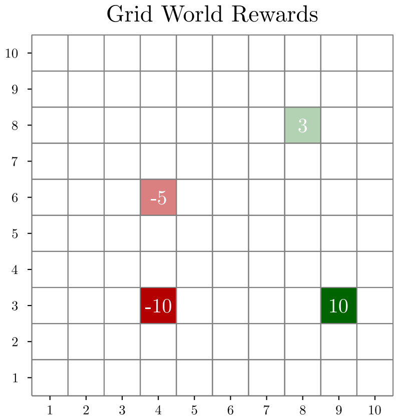
[1] Sometimes called the *transition model*.

[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**.

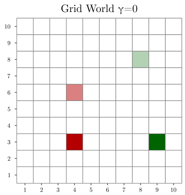


[1] Sometimes called the *reward model*.

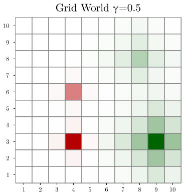
MDP: Discount factor

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

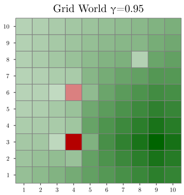
The ***discount factor*** controls how myopic (short-sighted) the agent is in its decision making (e.g., when $\gamma = 0$, the agent only cares about immediate rewards (myopic) and as $\gamma \to 1$, the agent takes in potential future information in its decision making process).



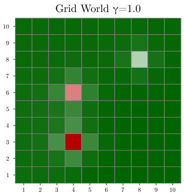
(a) Short-sighted (no reward spread)



(b) Some future reward[1] is spread



(c) Future reward is nicely spread



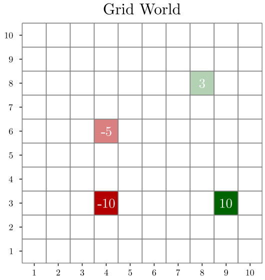
(d) Dominated by the future reward

[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-λ
`DeepQLearning.jl`	Continuous	Discrete	DQN, Double DQN, Dueling DQN, Recurrent Q-learning
`Crux.jl`	Discrete/Continuous	Discrete/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!

POMDPs: Partically Observable Markov Decision Processes

CARS Tutorial: POMDPs.jl

Decision Making Under Uncertainty

What is a POMDP?

Definition: POMDP. A *Partially observable Markov decision process* (POMDP) is an MDP with *state uncertainty*—meaning we cannot know the *true* state, only a *belief* about the true state using *observations*.

- Formally, a POMDP is defined by the following:

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

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

Remember, a POMDP is a ***problem formulation*** and ***not an algorithm***.

How are POMDPs different than MDPs?

- A POMDP[2] is an MDP with *state uncertainty*

$$\text{MDP: } \langle \mathcal{S}, \mathcal{A}, T, R, \gamma \rangle$$
$$\text{POMDP: } \langle \mathcal{S}, \mathcal{A}, \mathcal{O}, T, R, O, \gamma \rangle$$

- The agent receives an *observation* of the current state rather than the true state (potentially imperfect observations)
- Using past observations, the agent builds a *belief* of their underlying state
  - Which can be represented by a probability distribution over true states

[2] "Partially observable" is key in understanding beliefs.

Example POMDP: Crying Baby Problem

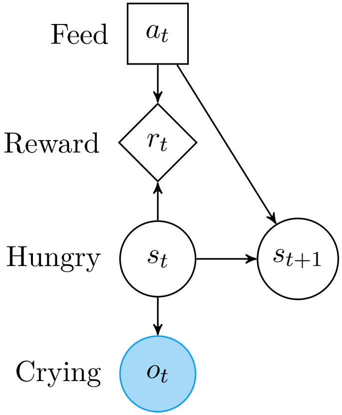


Figure: The crying baby POMDP.

- A simple POMDP with 2 states, 2 actions, and 2 observations:

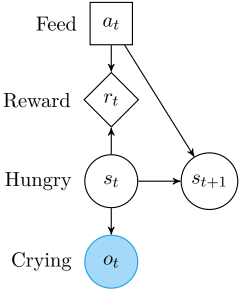
$$\mathcal{S} = \{\texttt{hungry}, \texttt{full}\}$$
$$\mathcal{A} = \{\texttt{feed}, \texttt{ignore}\}$$
$$\mathcal{O} = \{\texttt{crying}, \texttt{quiet}\}$$

- We cannot directly tell if the baby is truly `hungry`, but we can observe that it's `crying` and update our *belief* about the true state using this information.

QuickPOMDPs: Crying Baby

```julia
using POMDPs, POMDPModelTools, QuickPOMDPs

@enum State hungry full
@enum Action feed ignore
@enum Observation crying quiet

pomdp = QuickPOMDP(
    states       = [hungry, full],  # 𝒮
    actions      = [feed, ignore],  # 𝒜
    observations = [crying, quiet], # 𝒪
    initialstate = [full], # Deterministic
    discount     = 0.9, # γ

    transition = function T(s, a)
        if a == feed
            return SparseCat([hungry, full], [0, 1])
        elseif s == hungry && a == ignore
            return SparseCat([hungry, full], [1, 0])
        elseif s == full && a == ignore
            return SparseCat([hungry, full], [0.1, 0.9])
        end
    end,

    observation = function O(s, a, s′)
        if s′ == hungry
            return SparseCat([crying, quiet], [0.8, 0.2])
        elseif s′ == full
            return SparseCat([crying, quiet], [0.1, 0.9])
        end
    end,

    reward = (s,a)->(s == hungry ? -10 : 0) + (a == feed ? -5 : 0)
)
```

- This code[a] defines the entire *Crying Baby* POMDP 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!

POMDP solvers

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

Table: POMDP Solution Methods

Package	Online/Offline	State Spaces	Actions Spaces	Observation Spaces
`QMDP.jl`	Offline	Discrete	Discrete	Discrete
`FIB.jl`	Offline	Discrete	Discrete	Discrete
`BeliefGridValueIteration.jl`	Offline	Discrete	Discrete	Discrete
`SARSOP.jl`	Offline	Discrete	Discrete	Discrete
`BasicPOMCP.jl`	Online	Continuous	Discrete	Discrete
`ARDESPOT.jl`	Online	Continuous	Discrete	Discrete
`MCVI.jl`	Offline	Continuous	Discrete	Continuous
`POMDPSolve.jl`	Offline	Discrete	Discrete	Discrete
`IncrementalPruning.jl`	Offline	Discrete	Discrete	Discrete
`POMCPOW.jl`	Online	Continuous	Continuous	Continuous
`AEMS.jl`	Online	Discrete	Discrete	Discrete
`PointBasedValueIteration.jl`	Offline	Discrete	Discrete	Discrete

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