UNIT-IV

Markov Decision process: MDP formulation, utility theory, utility functions, value iteration, policy iteration and partially observable MDPs.

Uncertainty

- In knowledge representation
  - A → B , if A is true then B is true
  - Example: Toothache → Cavity
- A is true or not then we cannot express this statement, this situation is called uncertainty.
- Uncertainty means working with imperfect or incomplete information.
  - Toothache → Cavity V Gum-disease V wisdom teeth …
- So to represent uncertain knowledge, we need uncertain reasoning or probabilistic reasoning.

Uncertainty

- An agent must first have preferences between the different possible outcomes of the various plans.
- A particular outcome is a completely specified state.
- Using utility theory to represent and reason with preferences
- Utility is "the quality of being useful,"
- Utility theory says that every state has a degree and that the agent will prefer states with higher utility.

Decision theory

- Preferences, as expressed by utilities, are combined with probabilities in the general theory of rational decisions called decision theory
- Decision theory = probability theory + utility theory
- An agent is rational if and only if it chooses the action that yields the highest expected utility, averaged over all the possible outcomes of the action.
- This is called the principle of Maximum Expected Utility (MEU).

Design for a decision-theoretic agent

```
function DT-AGENT( percept) returns an action
  persistent: belief_state, probabilistic beliefs about the current state of the world
              action, the agent's action

  update belief_state based on action and percept
  calculate outcome probabilities for actions,
      given action descriptions and current belief_state
  select action with highest expected utility
      given probabilities of outcomes and utility information
  return action
```

Utility Theory

- Utility theory depends on
  - utility function
  - rational agent

Agents in AI

- An agent can be anything that perceive its environment through sensors and act upon that environment through actuators. An Agent runs in the cycle of perceiving, thinking, and acting.
- An agent can be:
  - Human-Agent: A human agent has eyes, ears, and other organs which work for sensors and hand, legs, vocal tract work for actuators.
  - Software Agent: Software agent can have keystrokes, file contents as sensory input and act on those inputs and display output on the screen.
  - Robotic Agent: A robotic agent can have cameras, infrared range finder, NLP for sensors and various motors for actuators.

Intelligent Agents

- An intelligent agent is an autonomous entity which act upon an environment using sensors and actuators for achieving goals.
- Following are the main four rules for an AI agent:
  - Rule 1: An AI agent must have the ability to perceive the environment.
  - Rule 2: The observation must be used to make decisions.
  - Rule 3: Decision should result in an action.
  - Rule 4: The action taken by an AI agent must be a rational action.

Rational Agent:

- A rational agent is an agent which has clear preference, models uncertainty, and acts in a way to maximize its performance measure with all possible actions.
- A rational agent is said to perform the right things. AI is about creating rational agents to use for game theory and decision theory for various real-world scenarios.
- in AI reinforcement learning algorithm, for each best possible action, agent gets the positive reward and for each wrong action, an agent gets a negative reward.

Utility Theory

- We are designing rational agents that maximize expected utility.
- Probability theory is a tool for dealing with degrees of belief
- The agent‘s preferences are captured by a utility function, U(s), which assigns a single number to express desirability of a state.
- The expected utility of an action given the evidence is just the average value of outcomes, weighted bytheir probabilities
  - EU(a|e) = ∑ s P(Result(a)=s|a,e) U(s)

Cont..

- A rational agent should choose the action that maximizes the agent's expected utility (MEU)
  - action = argmax a EU(a|e)
- The MEU principle formalizes the general notion that the agent should "do the right thing", but we need make it operational.

Rational preferences

- An agent chooses among prizes (A, B, etc.)
- Notation:
  - $A \succ B$ The Agent preferred A over B
  - $A \sim B$ The agent is indifference between A and B
  - $A \succ\sim B$ The agent preferred A over B or is indifferent between them.
- Lottery L = [p, A; (1 − p), B], i.e., situations with uncertain prizes
- An example of lottery (food in airplanes) Chicken or pasta?
  - [0.8, juicy chicken; 0.2, overcooked chicken]
  - [0.7, delicious pasta; 0.3, congealed pasta]

Axioms of the utility theory

- Orderability: Given any two states, the a rational agent prefers one of them, else the two as equally preferable.
  - $(A \succ B) \vee (B \succ A) \vee (A \sim B)$
- Transitivity: Given any three states, if an agent prefers A to B and prefers B to C, agent must prefer A to C.
  - $(A \succ B) \wedge (B \succ C) \hat{O}\Rightarrow (A \succ C)$
- Continuity: If some state B is between A and C in preference, then there is a p for which the rational agent will be indifferent between state B and the lottery in which A comes with probability p, C with probability (1-p).
  - $A \succ B \succ C \hat{O}\Rightarrow \exists\, p\ [p, A; 1 - p, C] \sim B$

- Substitutability: If an agent is indifferent between two lotteries, A and B, then there is a more complex lottery in which A can be substituted with B.
  - $(A \sim B) \Rightarrow [p : A ; (1 - p) : C] \sim [p : B ; (1 - p) : C]$
- Monotonicity: If an agent prefers A to B, then the agent must prefer the lottery in which A occurs with a higher probability
  - $(A \succ B) \Rightarrow (p \geq q \Leftrightarrow [p, A; 1 - p, B] \succ \sim [q, A; 1 - q, B])$
- Decomposability: Compound lotteries can be reduced to simpler lotteries using the laws of probability.
  - $[p : A ; (1 - p) : [q : B ; (1 - q) : C]] \Rightarrow [p : A ; (1 - p) q : B ; (1 - p)(1 - q) : C]$

Utility

- Problem: infinite lifetimes $\Longrightarrow$ additive utilities are infinite
- 1) **Finite horizon**: termination at a *fixed time* T
  $\Longrightarrow$ **nonstationary** policy: $\pi(s)$ depends on time left
- 2) **Absorbing state(s)**: w/ prob. 1, agent eventually "dies" for any π
  $\Longrightarrow$ expected utility of every state is finite
- 3) **Discounting**: assuming $\gamma < 1$, $R(s) \leq R_{\max}$,

$$U([s_0, \ldots s_\infty]) = \sum_{t=0}^{\infty} \gamma^t R(s_t) \leq R_{\max}/(1-\gamma)$$

  Smaller $\gamma \Rightarrow$ shorter horizon
- 4) Maximize **system gain** = average reward per time step
  Theorem: optimal policy has constant gain after initial transient
  E.g., taxi driver's daily scheme cruising for passengers

Bellman Equation

- Definition of utility of states leads to a simple relationship among utilities of neighboring states:

- **Expected sum of rewards**
  = current reward
  + $\gamma \times$ expected sum of rewards after taking best action

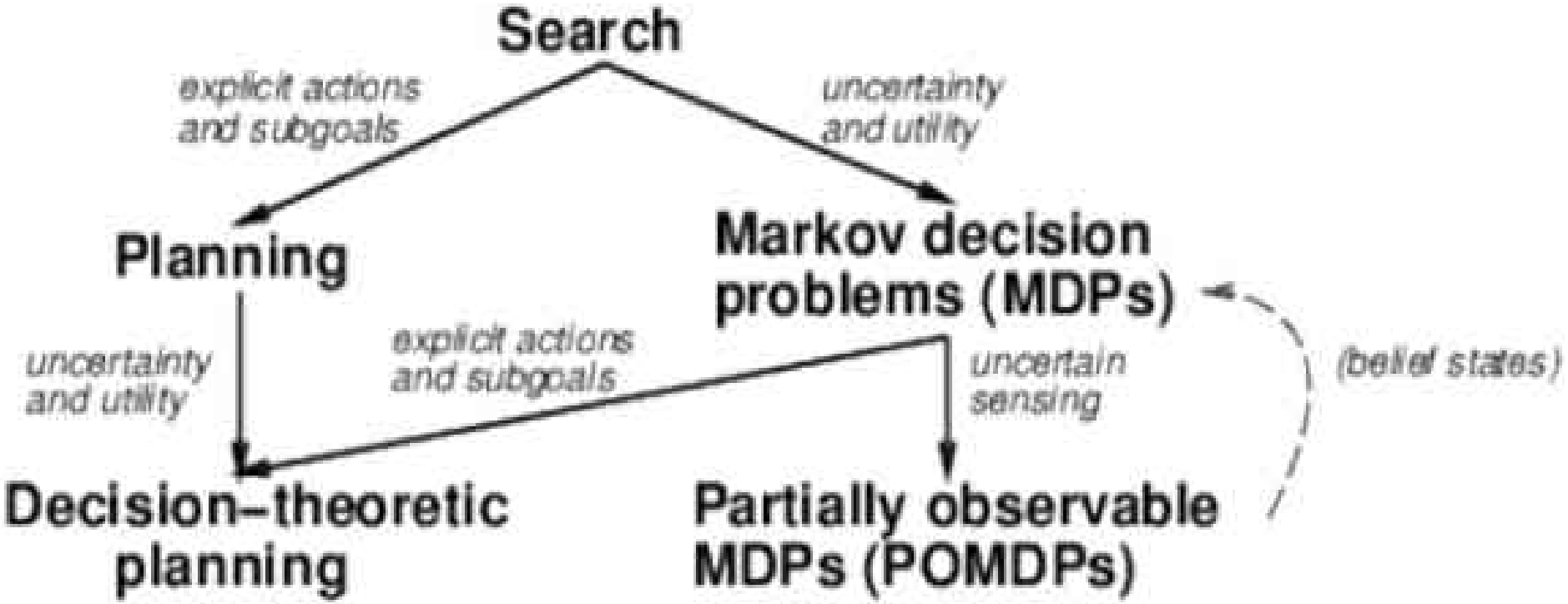
- Bellman equation (1957):

$$U(s) = R(s) + \gamma \max_a \sum_{s'} U(s')T(s,a,s')$$

- $U(1,1) = -0.04$
  $+ \gamma \max\{0.8U(1,2) + 0.1U(2,1) + 0.1U(1,1),$ *up*
  $0.9U(1,1) + 0.1U(1,2)$ *left*
  $0.9U(1,1) + 0.1U(2,1)$ *down*
  $0.8U(2,1) + 0.1U(1,2) + 0.1U(1,1)\}$ *right*

- One equation per state = n **nonlinear** equations in n unknowns

Sequential Decision Problems

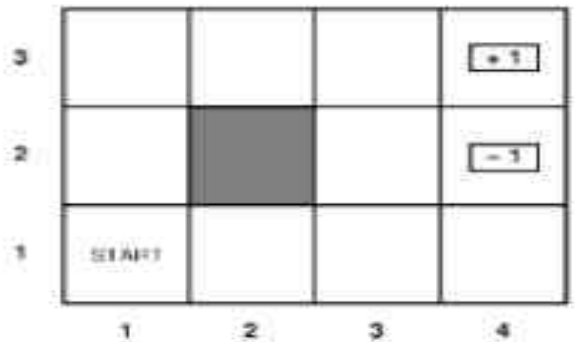


Markov Decision Process

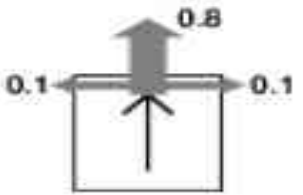
- A sequential decision problem for a fully observable, stochastic environment with a Markovian transition model and additive rewards is called a Markov decision process,or MDP,
- It consists of a set of states(S) a set ACTIONS(A) of actions in each state, a transition model P(s' a, s); and a reward function R(s,a)

Example Markov Decision Process

State Map



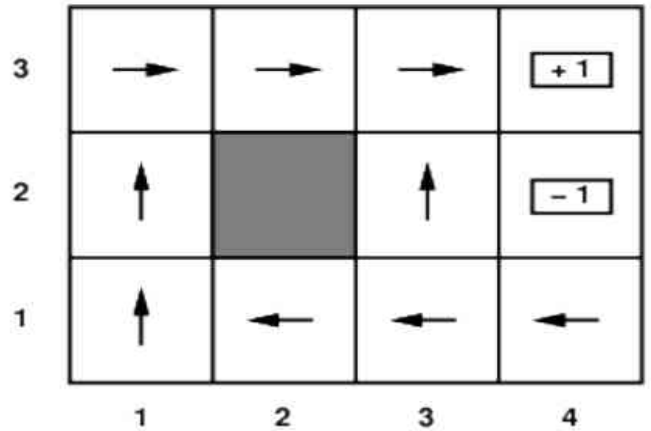
Stochastic Movement



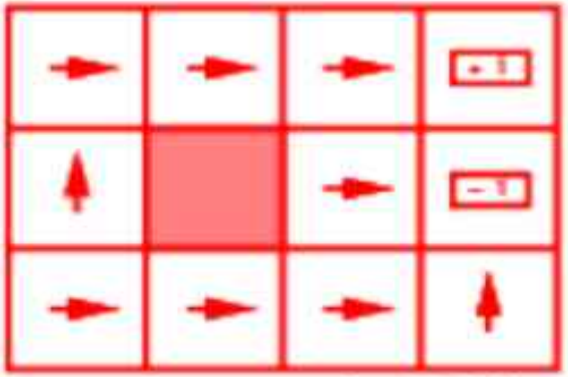
- States $s \in S$, actions $a \in A$
- Model $T(s, a, s') \equiv P(s'|s, a)$ = probability that a in s leads to s'
- Reward function $R(s)$ (or $R(s, a)$, $R(s, a, s')$)

$$= \begin{cases} -0.04 & \text{(small penalty) for nonterminal states} \\ \pm 1 & \text{for terminal states} \end{cases}$$

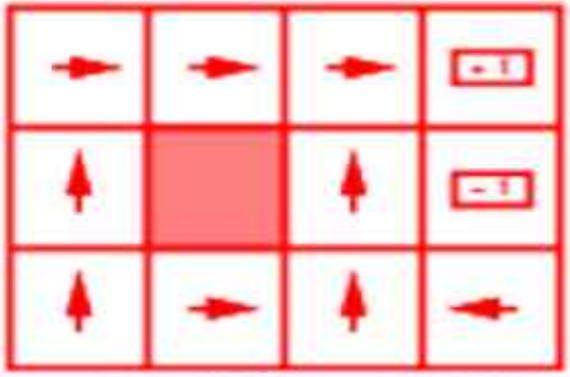
policy



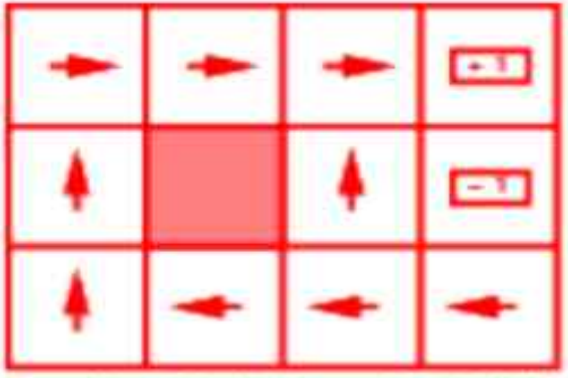
Risk and Reward



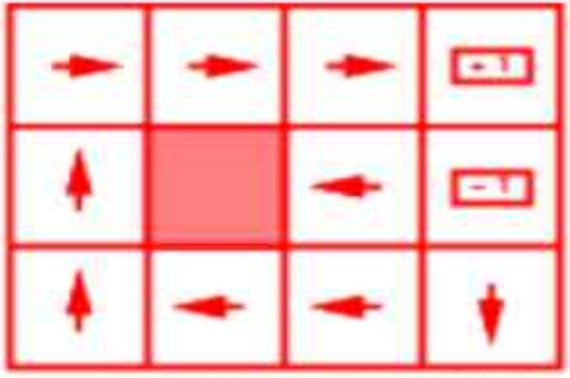
$r = [-\infty : -1.6284]$



$r = [-0.4278 : -0.0850]$



$r = [-0.0480 : -0.0274]$



$r = [-0.0218 : 0.0000]$

Utility of State Sequences

- Need to understand preferences between sequences of states
- Typically consider stationary preferences on reward sequences:

$[r, r0 , r1 , r2 , . . .] \succ [r, r0' , r1' , r2' , . . .] \Leftrightarrow [r0 , r1 , r2 , . . .] \succ [r0' , r1' , r2' , . . .]$

- There are two ways to combine rewards over time
  - Additive utility function:

  $U([s 0 , s 1 , s 2 , . . .]) = R(s 0) + R(s 1) + R(s 2) + \cdots$

  - Discounted utility function:

  $U([s 0 , s 1 , s 2 , . . .]) = R(s 0) + \gamma R(s 1) + \gamma 2 R(s 2) + \cdots$

where γ is the discount factor

Utility of a state

- Utility of a state (a.k.a. its value) is defined to be

U (s) = expected (discounted) sum of rewards (until termination) assuming optimal actions

- Given the utilities of the states, choosing the best action is just MEU: maximize the expected utility of the immediate successors

Bellman Equation

- Definition of utility of states leads to a simple relationship among utilities of neighboring states:

- **Expected sum of rewards**
  = current reward
  + $\gamma \times$ expected sum of rewards after taking best action

- Bellman equation (1957):

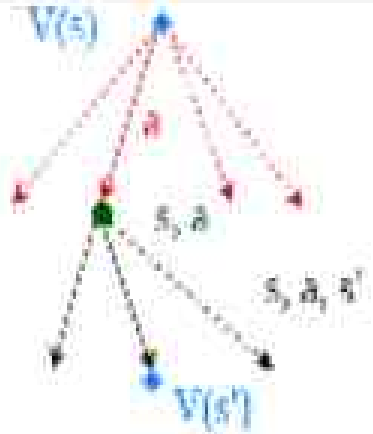
$$U(s) = R(s) + \gamma \max_a \sum_{s'} U(s')T(s,a,s')$$

- $U(1,1) = -0.04$
  $+ \gamma \max\{0.8U(1,2) + 0.1U(2,1) + 0.1U(1,1),$ *up*
  $0.9U(1,1) + 0.1U(1,2)$ *left*
  $0.9U(1,1) + 0.1U(2,1)$ *down*
  $0.8U(2,1) + 0.1U(1,2) + 0.1U(1,1)\}$ *right*

- One equation per state = n **nonlinear** equations in n unknowns

Value Iteration

- In value iteration, we compute the optimal state value function by iteratively updating the estimate v(s):
- Value iteration learns the value of the states from the Bellman Update directly.
- Learning a policy may be more direct than learning a value.
- Learning a value may take an infinite amount of time

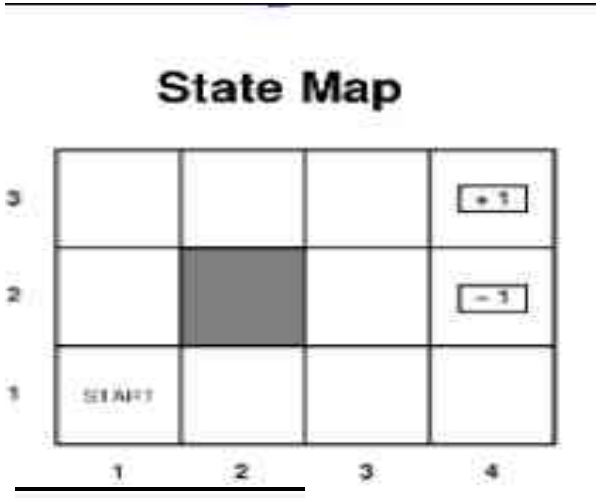


We start with a random value function $V(s)$. At each step we update it:

$$V(s) \leftarrow \max_a \sum_{s',r} p(s',r|s,a)[r + \gamma V(s')]$$

Hence, we look-ahead one step and go over all possible actions at each iteration to find the maximum

Example



K=0			
0.0	0.0	0.0	0.0
0.0		0.0	0.0
0.0	0.0	0.0	0.0

K=1			
0.0	0.0	0.0	1
0.0		0.0	-1
0.0	0.0	0.0	0.0

V(3,4)=+1

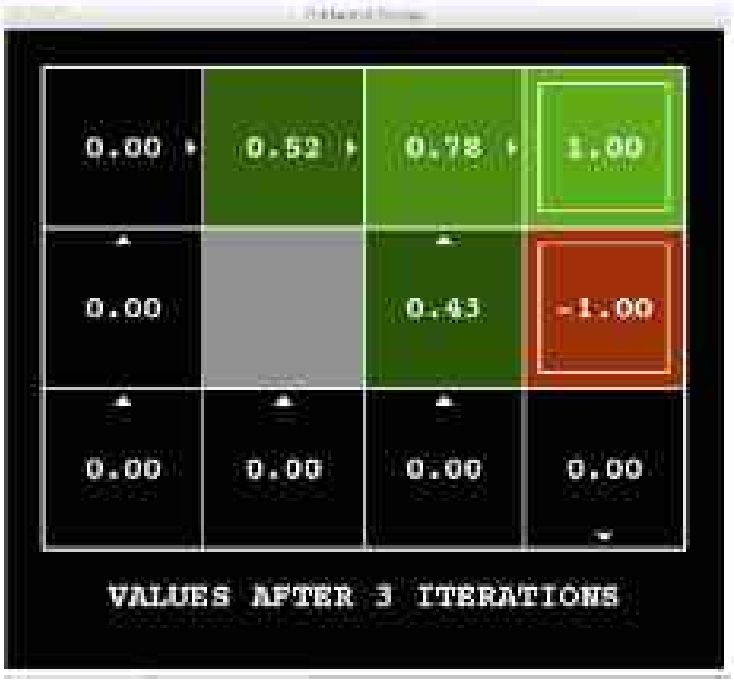
V(2,4)= -1

K=2			
0.0	0.0	0.72	1
0.0		0.0	-1
0.0	0.0	0.0	0.0

$V(3,3)=P(S`|(3,3), right)[R((3,3),right,S`)+\gamma(V(S`)$

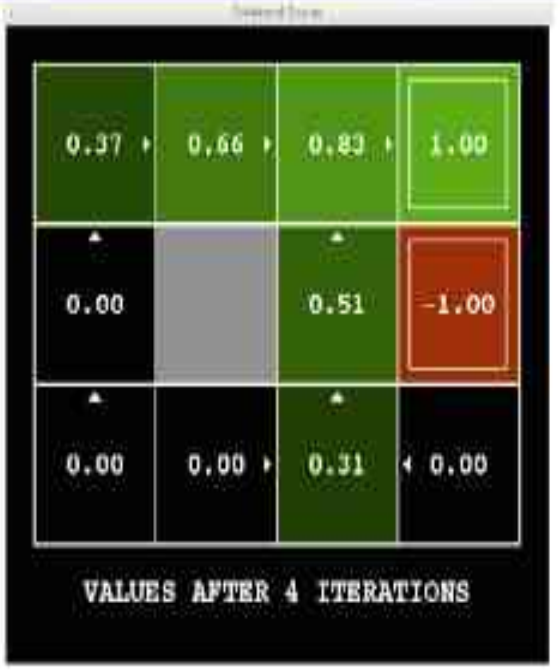
$V(3,3)=0.8(0+0.9*1)=0.72$

k=3



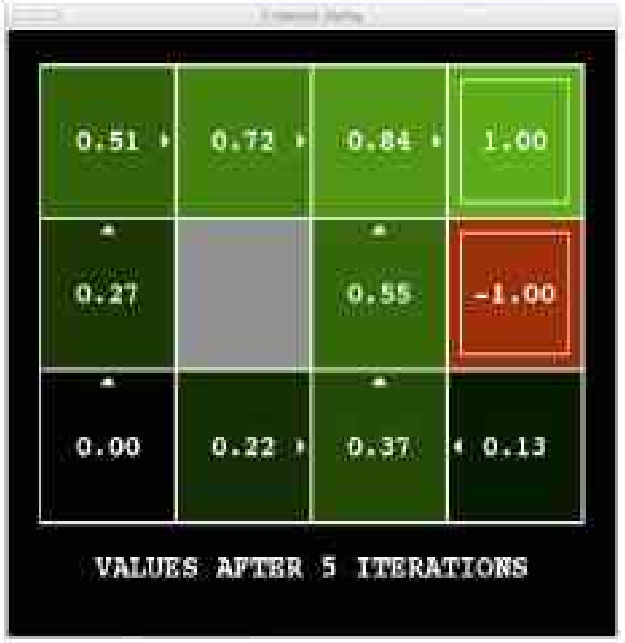
Noise = 0.2
Discount = 0.9
Living reward = 0

k=4



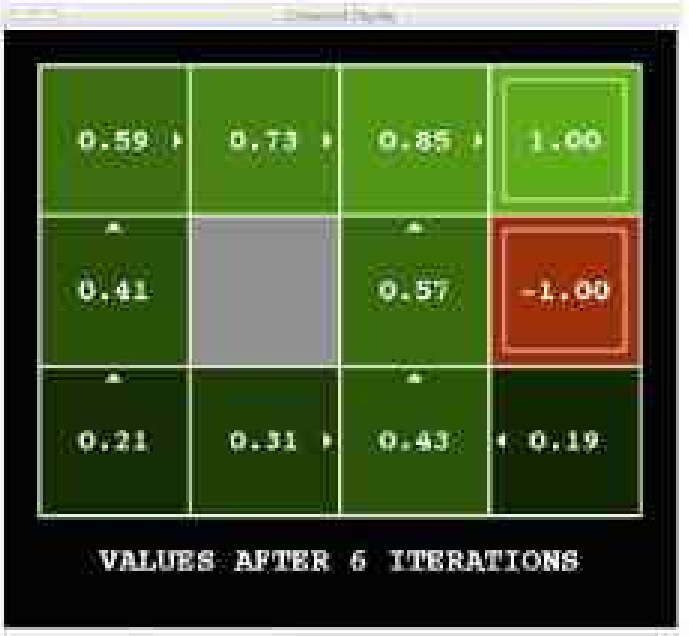
Noise = 0.2
Discount = 0.9
Living reward = 0

k=5



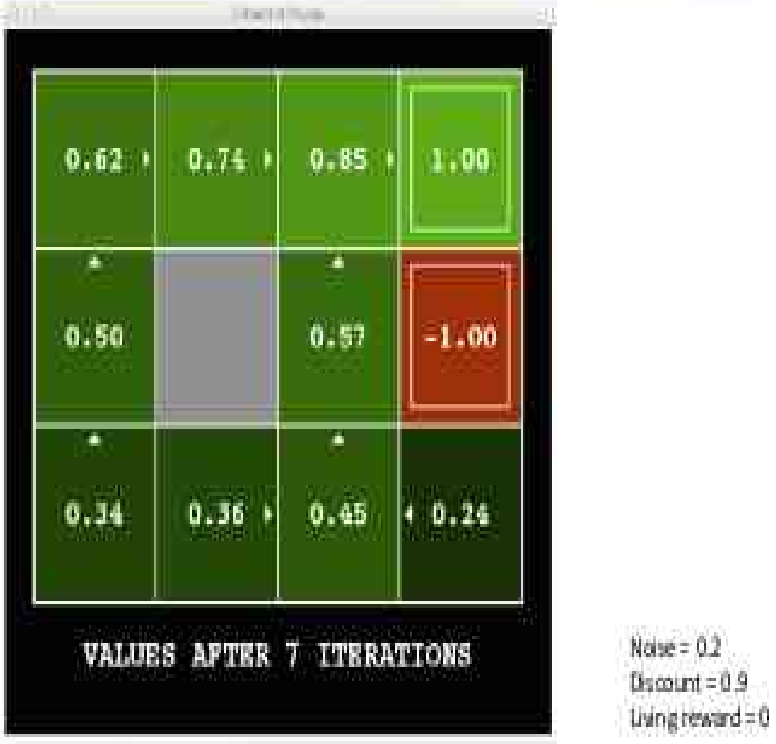
Noise = 0.2
Discount = 0.9
Living reward = 0

k=6



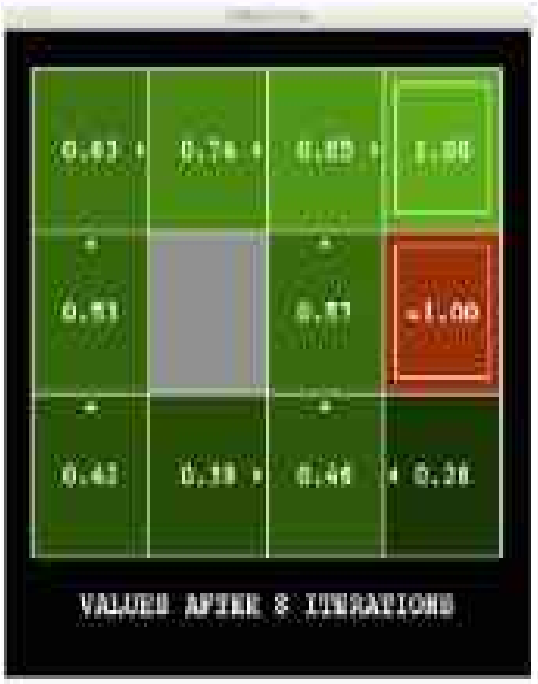
Noise = 0.2
Discount = 0.9
Living reward = 0

k=7

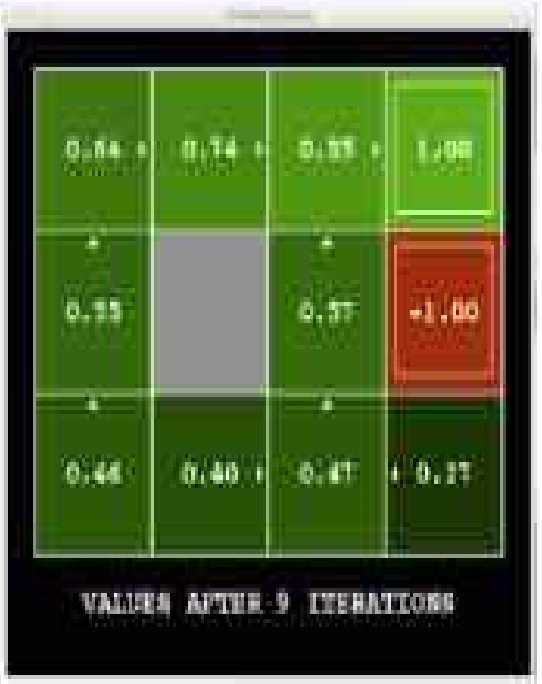


Noise = 0.2
Discount = 0.9
Living reward = 0

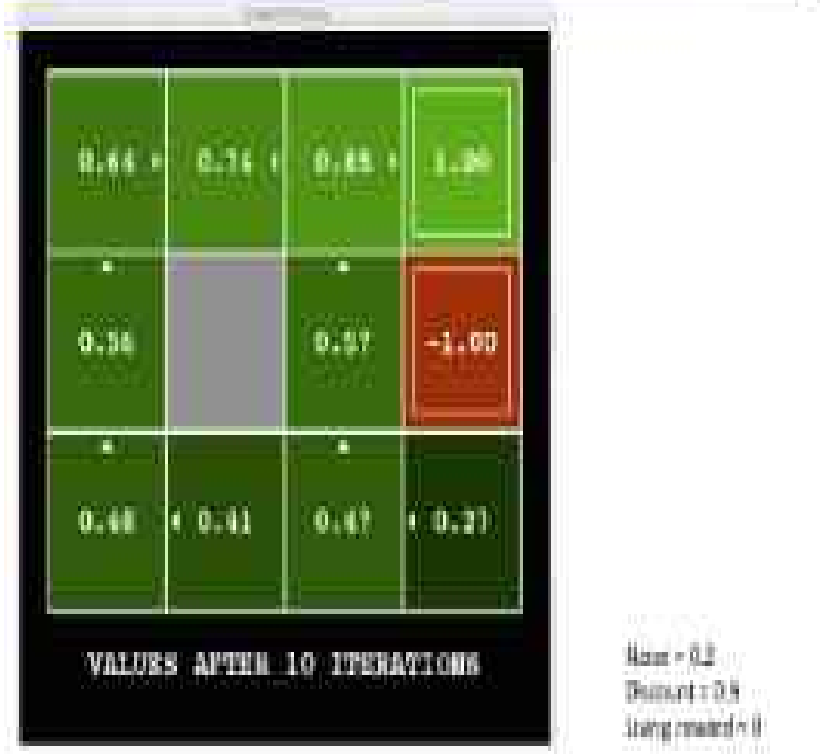
k=8


0.83
0.76
0.85
1.00
0.51
0.57
-1.00
0.43
0.38
0.48
0.28
VALUES AFTER 8 ITERATIONS

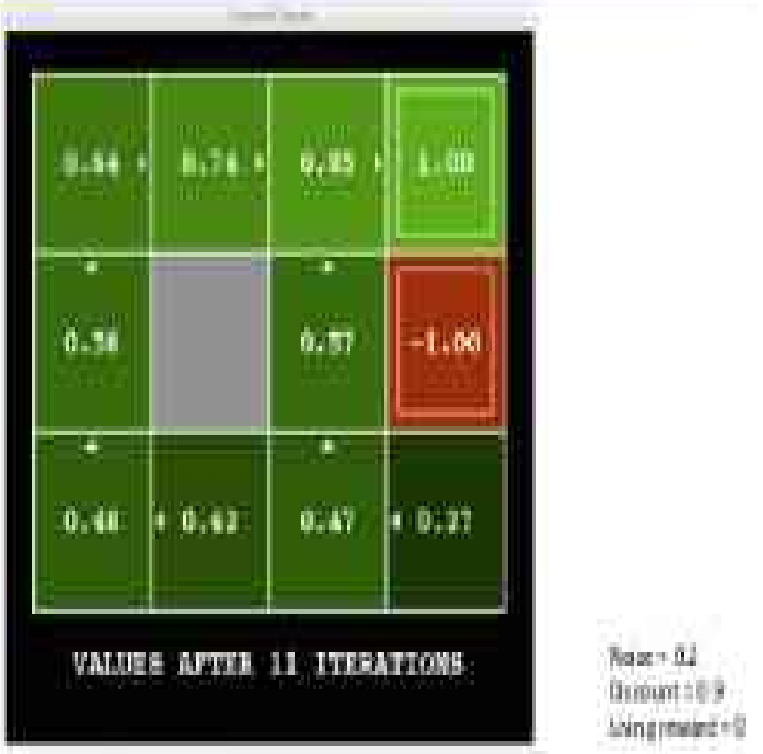
k=9


0.84
0.74
0.85
1.00
0.55
0.57
-1.00
0.46
0.40
0.47
0.27
VALUES AFTER 9 ITERATIONS

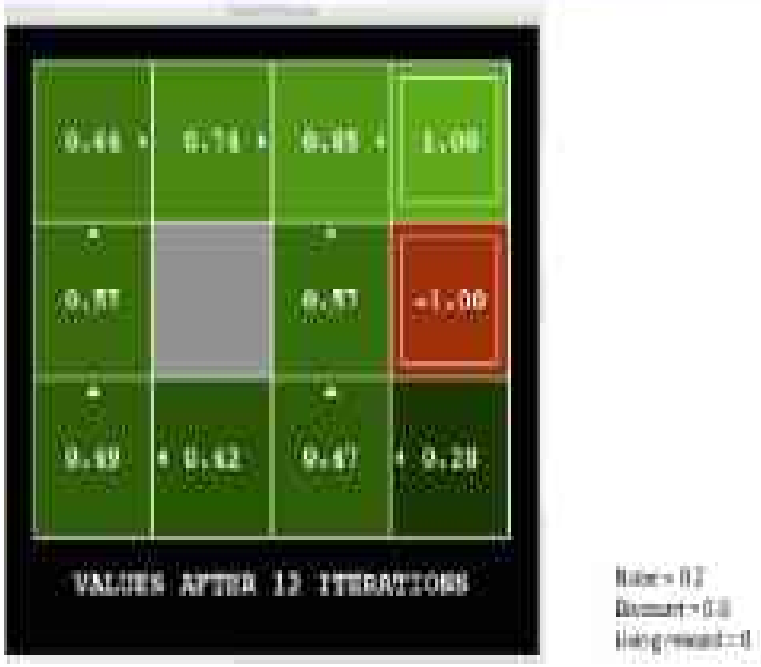
k=10


0.64
0.74
0.85
1.00
0.56
0.67
-1.00
0.48
0.41
0.47
0.27
VALUES AFTER 10 ITERATIONS

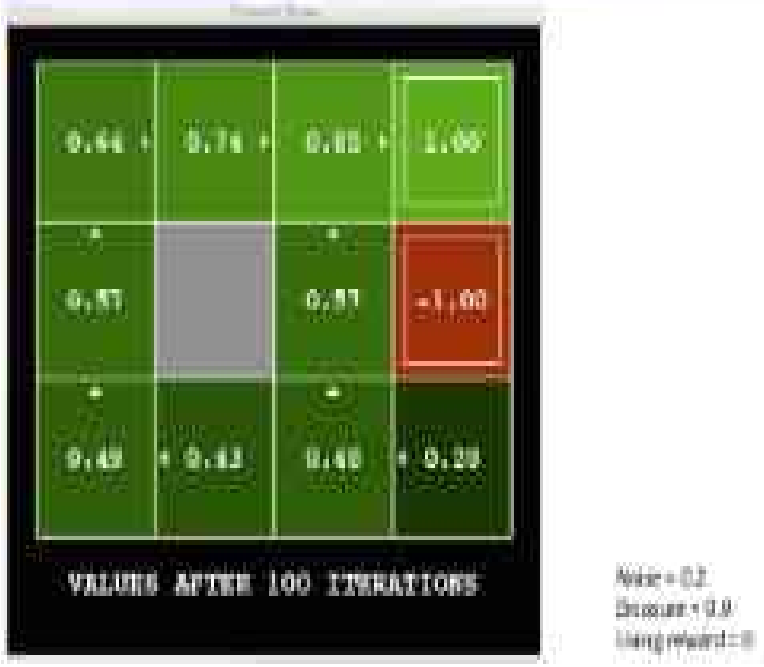
k=11


0.64
0.74
0.85
1.00
0.56
0.67
-1.00
0.48
0.43
0.47
0.27
VALUES AFTER 11 ITERATIONS

k=12



k=100



Value Iteration Algorithm

With this, we can construct the value iteration algorithm as follows:

1. Let $V_0(s)$ be any function $V_0 : S \to \mathbb{R}$ [Note: **not stage 0, but iteration 0**]
2. Apply the **principle of optimality** so that given V_i at iteration i, we compute

$$V_{i+1}(s) = \max_{a \in A} r(s,a) + \gamma \mathbb{E}_{s' \sim P(\cdot|s,a)} V_i(s')$$

3. Terminate when V_i stops improving, e.g. when $\max_s |V_{i+1}(s) - V_i(s)|$ is small.
4. Return the greedy policy:

$$\pi_K(s) = \arg\max_{a \in A} r(s,a) + \gamma \mathbb{E}_{s' \sim P(\cdot|s,a)} V_K(s')$$

Algorithm 1 Finite Horizon Value Iteration

for $t = T-1, T-2, \ldots, 0$ **do**
  for $s \in \mathcal{S}$ **do**
    $\pi_t(s), V_t(s) = \text{maximize}_a \, \mathbb{E}[r_t + V_{t+1}(s_{t+1})]$
  end for
end for

Algorithm 2 Infinite-Horizon Value Iteration

Initialize $V^{(0)}$ arbitrarily.
for $n = 0, 1, 2, \ldots$ until termination condition **do**
  for $s \in \mathcal{S}$ **do**

    $\pi^{(n+1)}(s), V^{(n+1)}(s) = \text{maximize}_a \, E_{s_T}[r_{T-1} + \gamma V^{(n)}(s_T)]$
  end for
end for

Note that $V^{(n)}$ is exactly V_0 in a finite-horizon problem with n timesteps.

Algorithm 2: Value Iteration Algorithm

Data: θ: a small number
Result: π: a deterministic policy s.t. $\pi \approx \pi_*$
Function *ValueIteration* **is**
    /* Initialization */
    Initialize $V(s)$ arbitrarily, except $V(terminal)$;
    $V(terminal) \leftarrow 0$;
    /* Loop until convergence */
    $\Delta \leftarrow 0$;
    while $\Delta < \theta$ **do**
        for *each* $s \in S$ **do**
            $v \leftarrow V(s)$;
            $V(s) \leftarrow \max_a \sum_{s',r'} p(s', r|s, a)[r + \gamma V(s')]$;
            $\Delta \leftarrow \max(\Delta, |v - V(s)|)$;
        end
    end
    /* Return optimal policy */
    return π s.t. $\pi(s) = arg\max_a \sum_{s',r'} p(s', r|s, a)[r + \gamma V(s')]$;
end

Policy Iteration

- In policy iteration, we start by choosing an arbitrary policy Π. Then, we iteratively evaluate and improve the policy until convergence:
- Policy learning incrementally looks at the current values and extracts a policy.
- action space is finite
- it can converge faster than Value Iteration.
- There are two steps to Policy Iteration.
  - Policy extraction
  - Policy Evaluation

- Start with some policy $\pi_0(s_i)$.
- Such policy transforms the MDP into a plain Markov system with rewards.
- Compute the values of the states according to current policy.
- Update policy:

$$\pi_1(s_i) = \text{argmax}_a\{r_i + \gamma \sum_j p_{ij}^a V^{\pi_0}(s_j)\}$$

- Keep computing
- Stop when $\pi_{k+1} = \pi_k$.

Policy Evaluation

$$v_\pi(s) = \sum_a \pi(a \mid s) \sum_{s',r} p(s', r \mid s, a) \left[r + \gamma v_\pi(s') \right]$$

Equation 1: Bellman expectation equation giving the state value under policy π

where:

$\pi(a|s)$ is the probability of taking action a in state s.
$p(s',r|s,a)$ is the probability of moving to the next state s' and getting reward r when starting in state s and taking action a.
r is the reward received after taking this action.
γ is the discount factor.
$v(s')$ is the value of the next state.

Policy Iteration

- Howard, 1960: search for optimal policy and utility values simultaneously

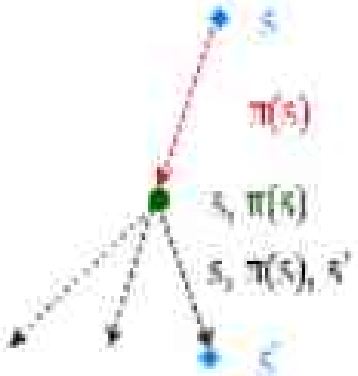
- Algorithm:

  $\pi \leftarrow$ an arbitrary initial policy
  repeat until no change in π
    compute utilities given π
    update π as if utilities were correct (i.e., local MEU)

- To compute utilities given a fixed π (value determination):

$$U(s) = R(s) + \gamma \sum_{s'} U(s')T(s, \pi(s), s') \qquad \text{for all } s$$

- i.e., n simultaneous linear equations in n unknowns, solve in $O(n^3)$



We evaluate a policy $\pi(s)$ by calculating the state value function $V(s)$:

$$V(s) = \sum_{s',r} p(s',r|s,\pi(s))[r + \gamma V(s')]$$

Then, we calculate the improved policy by using one-step look-ahead to replace the initial policy $\pi(s)$:

$$\pi(s) = \arg\max_a \sum_{s',r} p(s',r|s,a)[r + \gamma V(s')]$$

Here, r is the reward generated by taking the action a, γ is a discount factor for future rewards and p is the transition probability.

In the beginning, we don’t care about the initial policy Π being optimal or not.

During the execution, we concentrate on improving it on every iteration by repeating policy evaluation and policy improvement steps.

Using this algorithm we produce a chain of policies, where each policy is an improvement over the previous one:

Algorithm 1: Policy Iteration Algorithm

Data: θ: a small number
Result: V: a value function s.t. $V \approx v_*$, π: a deterministic policy s.t. $\pi \approx \pi_*$
Function *PolicyIteration* **is**
    /* Initialization */
    Initialize $V(s)$ arbitrarily;
    Randomly initialize policy $\pi(s)$;
    /* Policy Evaluation */
    $\Delta \leftarrow 0$;
    while $\Delta < \theta$ **do**
        for *each* $s \in S$ **do**
            $v \leftarrow V(s)$;
            $V(s) \leftarrow \sum_{s',r'} p(s', r|s, \pi(s))[r + \gamma V(s')]$;
            $\Delta \leftarrow \max(\Delta, |v - V(s)|)$;
        end
    end
    /* Policy Improvement */
    policy-stable $\leftarrow$ *true*;
    for *each* $s \in S$ **do**
        old-action $\leftarrow \pi(s)$;
        $\pi(s) \leftarrow arg\max_a \sum_{s',r'} p(s', r|s, a)[r + \gamma V(s')]$;
        if *old-action* != $\pi(s)$ **then**
            policy-stable $\leftarrow$ *false*;
        end
    end
    if $policy - stable$ **then**
        return $V \approx v_*$ and $\pi \approx \pi_*$;
    else
        go to Policy Evaluation;
    end
end

- Alternate between
  1. Evaluate policy $\pi \Rightarrow V^{\pi}$
  2. Set new policy to be *greedy* policy for V^{π}

$$\pi(s) = \arg\max_{a} \mathbb{E}_{s' \mid s,a} \left[r + \gamma V^{\pi}(s') \right]$$

- Guaranteed to converge to optimal policy and value function in a finite number of iterations, when $\gamma < 1$
- Value function converges faster than in value iteration[1]

Algorithm 4 Policy Iteration

Initialize $\pi^{(0)}$.
for $n = 1, 2, \ldots$ **do**
  $V^{(n-1)} = \text{Solve}[V = \mathcal{T}^{\pi^{(n-1)}} V]$
  $\pi^{(n)} = \mathcal{G} V^{\pi^{(n-1)}}$
end for

Modified Policy Iteration

- Policy iteration often converges in few iterations, but each is expensive
- Idea: use a few steps of value iteration (but with π fixed) starting from the value function produced the last time to produce an approximate value determination step.
- Often converges much faster than pure VI or PI
- Leads to much more general algorithms where Bellman value updates and Howard policy updates can be performed locally in any order
- Reinforcement learning algorithms operate by performing such updates based on the observed transitions made in an initially unknown environment

Modified Policy Iteration

- Policy iteration often converges in few iterations, but each is expensive
- Idea: use a few steps of value iteration (but with π fixed) starting from the value function produced the last time to produce an approximate value determination step.
- Often converges much faster than pure VI or PI
- Leads to much more general algorithms where Bellman value updates and Howard policy updates can be performed locally in any order
- Reinforcement learning algorithms operate by performing such updates based on the observed transitions made in an initially unknown environment

- Update π to be the greedy policy, then value function with k backups (k-step lookahead)

Algorithm 5 Modified Policy Iteration

Initialize $V^{(0)}$.
for $n = 1, 2, \ldots$ **do**
  $\pi^{(n+1)} = GV^{(n)}$
  $V^{(n+1)} = (\mathcal{T}^{\pi^{(n+1)}})^k V^{(n)}$, for integer $k \geq 1$
end for

- $k = 1$: value iteration
- $k = \infty$: policy iteration

Policy Iteration	Value Iteration
Starts with a random policy	Starts with a random value function
Algorithm is more complex	Algorithm is simpler
Guaranteed to converge	Guaranteed to converge
Cheaper to compute	More expensive to compute
Requires few iterations to converge	Requires more iterations to converge
Faster	Slower

Partial Observability

- System state can not always be determined

  ⇒ a Partially Observable MDP (POMDP)

  - Action outcomes are not fully observable
  - Add a set of observations O to the model
  - Add an observation distribution U(s,o) for each state
  - Add an initial state distribution I

POMDP to MDP Conversion

Belief state Pr(x) can be updated to Pr(x'|o) using Bayes' rule:

$$\Pr(s'|s,o) = \Pr(o|s,s')\Pr(s'|s) / \Pr(o|s)$$

$$= U(s',o)\ T(s',a,s) \text{ normalized}$$

$$\Pr(s'|o) = \Pr(s'|s,o)\Pr(s)$$

A POMDP is Markovian and fully observable relative to the belief state.

⇒ a POMDP can be treated as a continuous state MDP